`

Chapter 5

Dimension Reduction and Latent Variable Methods (Part 1)

It is not uncommon to have hundreds of process relevant variables being measured at manufacturing facilities. If you are looking to build a machine learning model using these process variables, then the high dimensionality (number of model variables) will present you several unique challenges related to algorithmic issues (due to collinearity among variables), difficulty in visualizing data, large computational costs, and slow model training. However, this problem, referred to as the *curse of dimensionality*, should not dampen your spirits! Conservation laws such as mass balances, thermodynamics constraints, enforced product specifications, and other operational restrictions induce correlations among the process variables and make it appear as if the measured variables are all derived from a small number of hidden (un-measured) variables. Latent variable-based methods reduce process dimensionality by finding these hidden latent variables.

PCA and PLS are among the most popular latent variable-based statistical tools and have been used successfully in several process monitoring and soft sensing applications. This chapter provides a comprehensive exposition of the PCA and PLS techniques and teaches you how to apply them on process data. Specifically, the following topics are covered

- Introduction to PCA and PLS
- Process modeling and monitoring via PCA and PLS
- Fault diagnosis for root cause analysis
- Nonlinear and dynamic variants of linear PCA and PLS

`

5.1 PCA: An Introduction

Principal component analysis (PCA) is a multivariate technique that transforms a high-dimensional set of correlated variables into a low-dimensional set of uncorrelated (latent) variables with minimum loss of information. Consider the 3-dimensional data in Figure 5.1. It is apparent that although the data is three dimensional, the data-points mostly lie along a 2-D plane; and even in this plane, the spread is much higher along a particular direction. PCA converts the original *(x,y,z)* space into a 2-D principal component (PC) space where the 1st PC (PC1) corresponds to the direction of maximum spread/variance and the 2nd PC (PC2) corresponds to the direction with highest variance among all directions orthogonal to 1st PC. Depending upon modeling requirements, even the 2nd PC may be discarded, essentially obtaining a 1-D data while losing out some information. Also, as we will see soon, it is straightforward to recover original data from data in PC space.

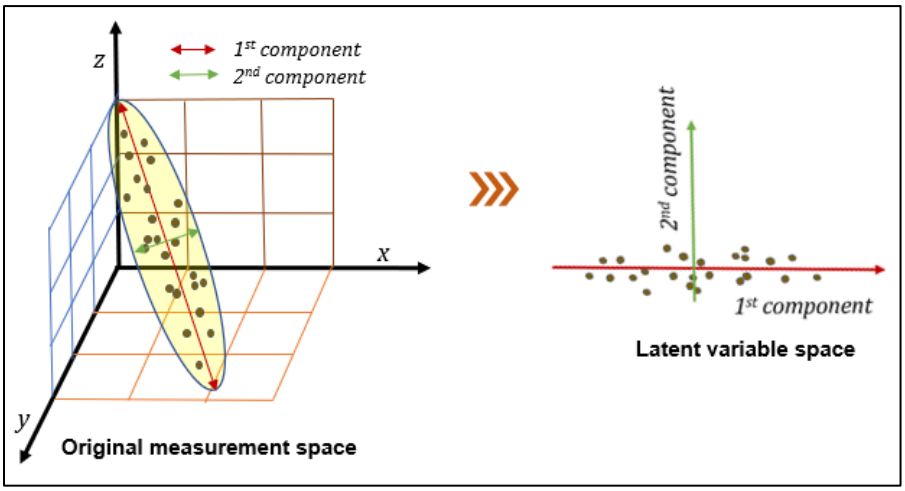


Figure 5.1: PCA illustration

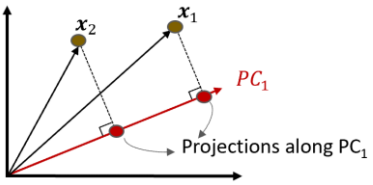
In ML world, it is common to find applications of classification and clustering techniques in the PC space. In process industry, process modeling (via principal component regression (PCR)) and monitoring are common application of PCA[17]. In PCR, the process outputs are regressed onto the principal components of the input data. By doing so, the problem of ill-conditioning frequently encountered in classical multiple linear regression (MLR) due to high input space dimensionality and high degree of correlation among input variables is circumvented. PCA is also frequently utilized for process visualization. For many applications, two or three PCs are adequate for capturing most of the variability in process data and therefore, the compressed process data can be visualized within a single plot. Plant operators and engineers use this single plot to find past and current patterns in process data.

[17] The popularity of latent-variable techniques for process control and monitoring arose from the pioneering work by John McGregor at McMaster University.

`

Mathematical background

Consider a data matrix $\boldsymbol{X} \in \mathbb{R}^{N\times m}$ consisting of *N* samples of *m* process variables where each row represents a data-point in the original measurement space. It is assumed that each column is normalized to zero mean and unit variance. Let $\boldsymbol{v} \in \mathbb{R}^{m}$ represent the 'loading' vector that projects data-points along PC1; it can be found by solving the following optimization problem



$$\max_{\boldsymbol{v}\neq 0} \frac{(\boldsymbol{Xv})^{\mathrm{T}}\boldsymbol{Xv}}{\boldsymbol{v}^{T}\boldsymbol{v}} \qquad \text{eq. 1}$$

It is apparent that eq. 1 is trying to maximize the variance of the projected data-points along PC1. Loading vectors for other PCs are found by solving the same problem with the added constraint of orthogonality to previously computed loading vectors. Alternatively, loading vectors can also be computed from eigenvalue decomposition of covariance matrix (***S***) of ***X***

$$\frac{1}{N-1}\boldsymbol{X}^{\boldsymbol{T}}\mathbf{X} = \boldsymbol{S} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{\boldsymbol{T}} \qquad \text{eq. 2}$$

Above is the form you will find commonly in PCA literature. The columns of eigenvector matrix $\boldsymbol{V} \in \mathbb{R}^{m\times m}$ are the loading vectors that we need. The diagonal eigenvalue matrix $\boldsymbol{\Lambda}$ equals $diag\{\lambda_1, \lambda_2, \ldots, \lambda_m\}$, where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_m$ are the eigenvalues. Infact, λ_j is equal to the variance along the j^{th} PC. If there is significant correlation among process variables in original data, only the first few eigenvalues will be significant. Let's assume that *k* PCs are retained, then the first *k* columns of ***V*** (which corresponds to the first *k* λ_s) are taken to form the loading matrix $\boldsymbol{P} \in \mathbb{R}^{m\times \mathrm{k}}$. Transformed data in the PC space can now be obtained

Projected values along j^{th} PC ⟵ $\boldsymbol{t}_j = \boldsymbol{X}\boldsymbol{p}_j \; or \; \boldsymbol{T} = \boldsymbol{XP}$ eq. 3

The m dimensional i^{th} row of ***X*** has been transformed into *k* (< *m*) dimensional i^{th} row of ***T***. $\boldsymbol{T} \in \mathbb{R}^{N\times k}$ is called score matrix and the j^{th} column of ***T*** ($\boldsymbol{t}_j$) contains the (score) values along the j^{th} PC. The scores can be projected back to the original measurement space as follows

$$\hat{\boldsymbol{X}} = \boldsymbol{T}\boldsymbol{P}^{T} \qquad \text{eq. 4}$$

Note that because we discarded the loading vectors corresponding to insignificant λ_s, $\hat{\boldsymbol{X}} \neq \boldsymbol{X}$. The difference $\boldsymbol{E} = \boldsymbol{X} - \hat{\boldsymbol{X}}$ is referred to as residual matrix as each row is the residual or error vector for a data-point. Overall, the PC space captures the systematic trends in process data and the residual space primarily describes the noise in data.

`

Dimensionality reduction for Polymer Manufacturing Process

Let us now see the powerful dimensionality reduction capability of PCA in action. We will use data from a polymer manufacturing facility. The dataset contains 33 variables and 92 hourly samples (Figure 5.2).

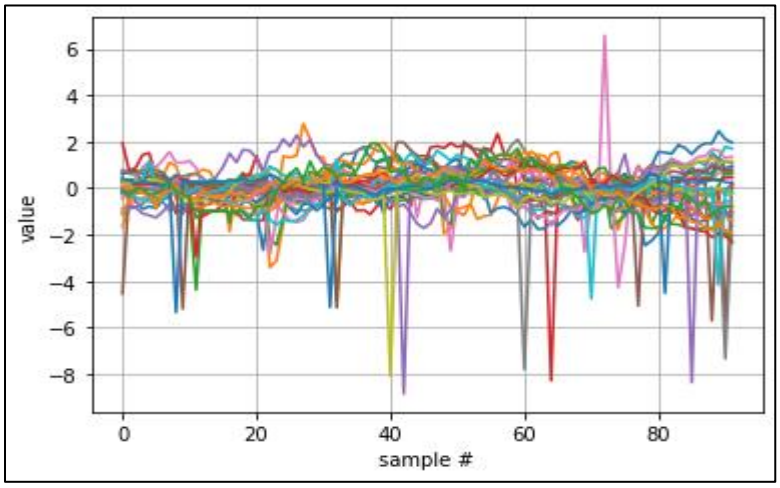


Figure 5.2: Process data from a polymer manufacturing plant. Each colored curve corresponds to a process variable

For this dataset, it is reported that the process started behaving abnormally around sample 70 and eventually had to be shut down. Therefore, we use samples 1 to 69 for training the PCA model using the code below. The rest of the data will be utilized for process monitoring illustration later.

```
# import requisite libraries
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# fetch data and separate training data
data = pd.read_excel('proc1a.xls', skiprows = 1, usecols = 'C:AI')
data_train = data.iloc[0:69,]

# normalize data
scaler = StandardScaler()
data_train_normal = scaler.fit_transform(data_train)

# PCA
pca = PCA()
score_train = pca.fit_transform(data_train_normal)
```

`

After training the PCA model, loading vectors/principal components can be accessed from transpose of the components_ attribute of pca model. Note that we have not accomplished any dimensionality reduction yet. PCA has simply provided us an uncorrelated dataset in score_train. To confirm this, we can compute the correlation coefficients among the columns of score_train. Only the diagonal values are 1 while the rest of the coefficients are 0!

```
# confirm no correlation
corr_coef = np.corrcoef(score_train, rowvar = False)
>>> print('Correlation matrix: \n', corr_coef[0:3,0:3]) # printing only a portion

Correlation matrix:
 [[ 1.00000000e+00  8.24652750e-16 -1.88830953e-16]
 [ 8.24652750e-16  1.00000000e+00  2.36966153e-16]
 [-1.88830953e-16  2.36966153e-16  1.00000000e+00]]
```

For dimensionality reduction we will need to study the variance along each PC. Note that the sum of variance along the *m* PCs equals the sum of variance along the *m* original dimensions. Therefore, the variance along each PC is also called explained variance. The attribute explained_variance_ratio gives the fraction of variance explained by each PC and Figure 5.3 clearly shows that not all 33 components are needed to capture all the information in data. Most of the information is captured in the first few PCs itself.

```
# visualize explained variance
import matplotlib.pyplot as plt

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

plt.figure()
plt.plot(cum_explained_variance, 'r+', label = 'cumulative % variance explained')
plt.plot(explained_variance, 'b+', label = 'variance explained by each PC')
plt.ylabel('Explained variance (in %)'), plt.xlabel('Principal component number'), plt.legend()
```

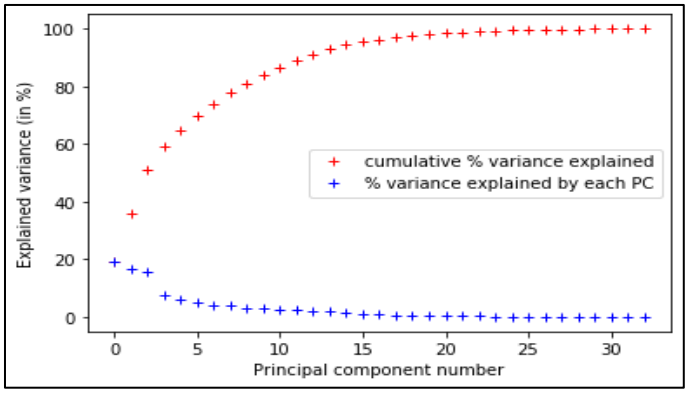


Figure 5.3: Variance explained by principal components

`

A popular approach for determining the number of PCs to retain is to select the number of PCs that cumulatively capture atleast 90% (or 95%) of the variance. The captured variance threshold should be guided by the expected level of noise or non-systematic variation that you do not expect to be captured. Alternative methods include cross-validation, scree tests, AIC criterion, etc. However, none of these methods are universally best in all the situations.

```
# decide # of PCs to retain and compute reduced data in PC space
n_comp = np.argmax(cum_explained_variance >= 90) + 1
score_train_reduced = score_train[:,0:n_comp]

>>> print('Number of PCs cumulatively explaining atleast 90% variance: ', n_comp)

Number of PCs cumulatively explaining atleast 90% variance: 13
```

Thus, we have achieved ~60% reduction in dimensionality (from 33 to 13) by sacrificing just 10% of the information. To confirm that only about 10% of the original information has been lost, we will reconstruct the original normalized data from the scores. Figure 5.4 provides a visual confirmation as well where it is apparent that the systematic trends in variables have been reconstructed while noisy fluctuations have been removed.

```
# confirm that only about 10% of original information is lost
from sklearn.metrics import r2_score

V_matrix = pca.components_.T
P_matrix = V_matrix[:,0:n_comp]

data_train_normal_reconstruct = np.dot(score_train_reduced, P_matrix.T)
R2_score = r2_score(data_train_normal, data_train_normal_reconstruct)

>>> print('% information lost = ', 100*(1-R2_score))

% information lost = 9.0469
```

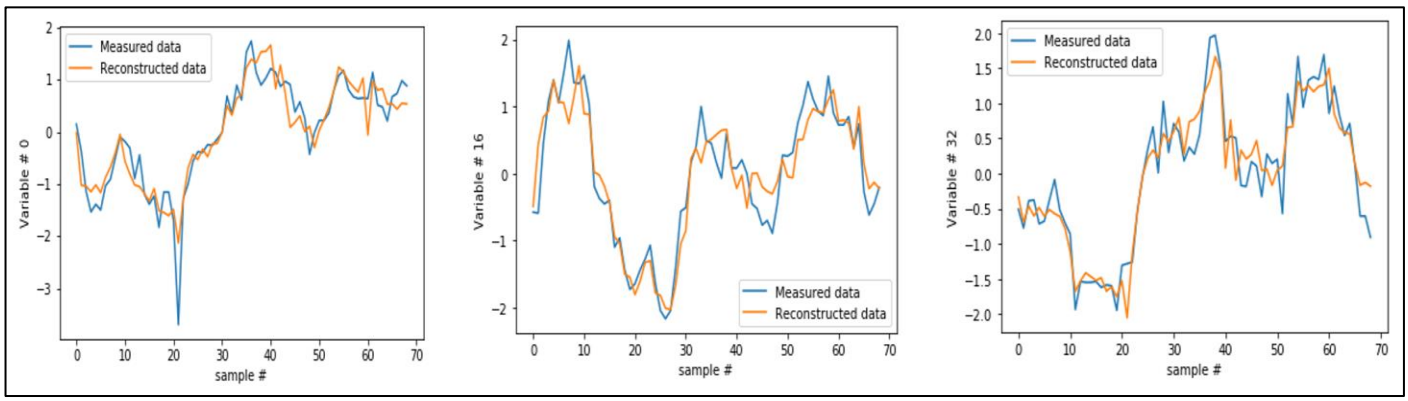


Figure 5.4: Comparison of measured and reconstructed values for a few variables

`

The 90% threshold could also have been specified during model training itself through the n_components parameter: pca = PCA(n_components = 0.9). In this case the insignificant PCs are not computed and the score_train_reduced matrix can be computed from the model using the transform method.

```
# alternative approach
pca = PCA(n_components = 0.9)
score_train_reduced = pca.fit_transform(data_train_normal)

data_train_normal_reconstruct = pca.inverse_transform(score_train_reduced)
R2_score = r2_score(data_train_normal, data_train_normal_reconstruct)

>>> print('% information lost = ', 100*(1-R2_score))

% information lost = 9.0469
```

5.2 Process Monitoring via PCA for Polymer Manufacturing Process

In Figure 5.2, we saw that it was not easy to infer process abnormality after 69th sample by simply looking at the combined time-series plot of all the available variables. Individual variable plot may provide better clues, but continuously monitoring all the 33 plots of individual variables is not a convenient task.

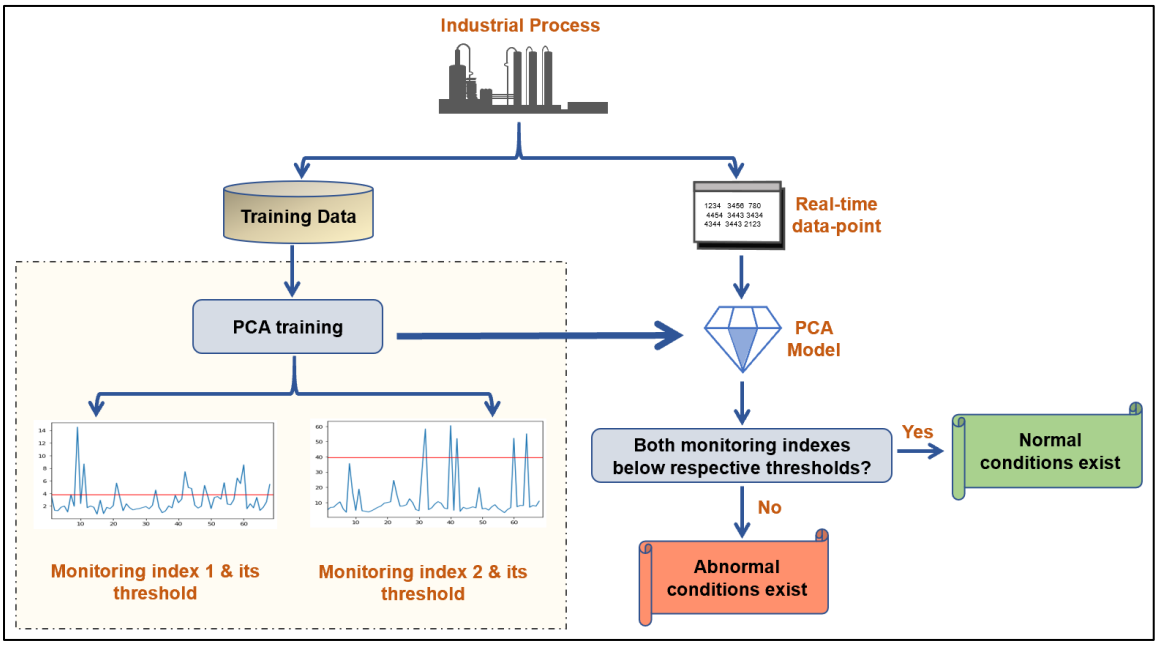


Figure 5.5: PCA-based process monitoring workflow

`

PCA makes the monitoring task easy by summarizing the state of any complex multivariate process into two simple indicators or monitoring indices as shown in Figure 5.5. During model training, statistical thresholds are determined for the indices and for a new data-point, the new indices' values are compared against the thresholds. If any of the two thresholds are violated, then presence of abnormal process condition is confirmed.

Process monitoring/fault detection indices

Hotelling's T^2 and SPE (also called Q) statistics are the two monitoring indices utilized for process monitoring. Beware the potential confusion between the score matrix (***T***) and the scalar T^2 value. This is the standard notation used by the PCA community. Both the statistics are computed for each data-point. Let $\boldsymbol{t}_i$ denote the i^{th} row of ***T*** which represents the transformed i^{th} data-point in the PC space. The T^2 index for this data-point is calculated as

$$T^2 = \sum_{j=1}^{k} \frac{t_{i,j}^2}{\lambda_j} = \boldsymbol{t}_i \boldsymbol{\Lambda}_k^{-1} \boldsymbol{t}_i^T \qquad \text{eq. 5}$$

Note that $\boldsymbol{\Lambda_k} = diag\{\lambda_1, \lambda_2, \ldots, \lambda_k\}$. It is apparent that T^2 is squared Mahalanobis distance or the weighted distance of a data-point from the origin in the PC space. If the data follow a multivariate normal distribution, the threshold or control limit for a specified false alarm rate of α (often set at 0.05 or 0.01) is given as follows

$$T_{CL}^2 = \frac{k(N^2-1)}{N(N-k)} F_{k,N-k}(\alpha) \qquad \text{eq. 6}$$

$F_{k,N-k}(\alpha)$ is the *(1-α)* percentile of a *F*-distribution with *k* and *n-k* degrees of freedom. In essence, $T^2 \leq T_{CL}^2$ represents an ellipsoidal boundary around the training data-points in the PC space. The second index, Q, represents the distance between the original and reconstructed data-point. Let e_i denote the i^{th} row of *E*. Then

$$Q = \sum_{j=1}^{m} e_{i,j}^2 \qquad \text{eq. 7}$$

Again, under normality assumption, the control limit for Q is given by the following expression

$$Q_{CL} = \theta_1 \left(\frac{z_\alpha \sqrt{2\theta_2 h_0^2}}{\theta_1} + 1 + \frac{\theta_2 h_0 (1-h_0)}{\theta_1^2} \right)^2 \qquad \text{eq. 8}$$

$$h_0 = 1 - \frac{2\theta_1 \theta_3}{3\theta_2^2} \quad \text{and} \quad \theta_r = \sum_{j=k+1}^{m} \lambda_j^r \quad ; r=1,2,3$$

z_α is the *(1-α)* percentile of a standard Gaussian distribution. We now have all the information required to generate plots of the monitoring indices, also called the control charts, for training data.

`

```
# calculate T^2 for training data
lambda_k = np.diag(pca.explained_variance_[0:n_comp]) # eigenvalue = explained variance
lambda_k_inv = np.linalg.inv(lambda_k)

T2_train = np.zeros((data_train_normal.shape[0],))
for i in range(data_train_normal.shape[0]):
    T2_train[i] = np.dot(np.dot(score_train_reduced[i,:],lambda_k_inv),score_train_reduced[i,:].T)

# calculate Q for training data
error_train = data_train_normal - data_train_normal_reconstruct
Q_train = np.sum(error_train*error_train, axis = 1)

# T2 control limit
import scipy.stats

N = data_train_normal.shape[0]
k = n_comp

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

# Q control limit
eig_vals = pca.explained_variance_
m = data_train_normal.shape[1]

theta1 = np.sum(eig_vals[k:])
theta2 = np.sum([eig_vals[j]**2 for j in range(k,m)])
theta3 = np.sum([eig_vals[j]**3 for j in range(k,m)])
h0 = 1-2*theta1*theta3/(3*theta2**2)

z_alpha = scipy.stats.norm.ppf(1-alpha)
Q_CL = theta1*(z_alpha*np.sqrt(2*theta2*h0**2)/theta1+ 1 + theta2*h0*(1-h0)/theta1**2)**2

# Q_train plot with CL
plt.figure()
plt.plot(Q_train)
plt.plot([1,len(Q_train)],[Q_CL,Q_CL], color='red')
plt.xlabel('Sample #'), plt.ylabel('Q for training data')

# T2_train plot with CL
plt.figure()
plt.plot(T2_train)
plt.plot([1,len(T2_train)],[T2_CL,T2_CL], color='red')
plt.xlabel('Sample #'), plt.ylabel('T$^2$ for training data')
```

`

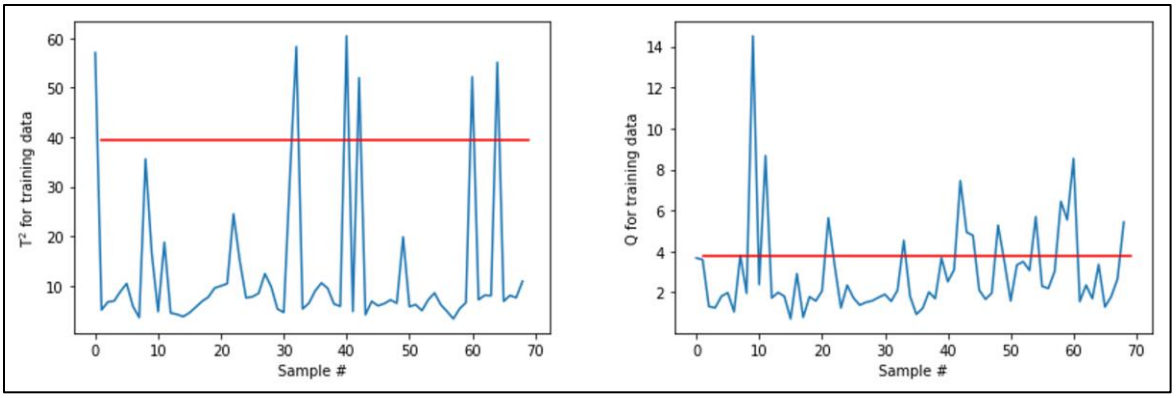


Figure 5.6: Monitoring charts for training data

Figure 5.6 shows that quite a few data-points in training data violate the thresholds, which was not expected with 99% control limits. This indicates that the multivariate normality assumption does not hold for this dataset. Other specialized ML methods like KDE, SVDD can be employed for control boundary determination for non-Gaussian data. We will study these methods in later chapters. Alternatively, if *N* is large, another popular approach is to directly find the control limits as 99th percentiles of the T^2 and *Q* values for training dataset.

Importance of both T^2 and Q statistic

The T^2 and *Q* indices quantify different kinds of variations in data. While T^2 is a measure of distance from origin in PC space, *Q* is a measure of the portion of data that is not explained by the PCA model. It is possible that an abnormal situation violates control limit of one index, but not the other. Therefore, it is crucial to monitor both the indices.

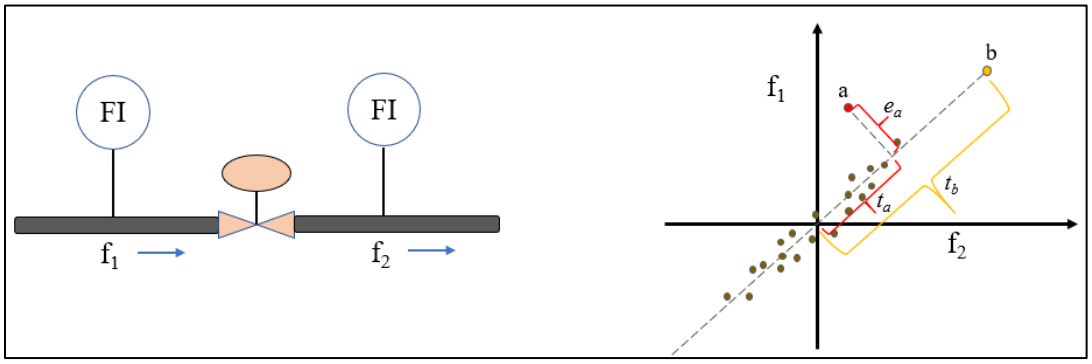


Figure 5.7: (Left) Flow measurements across a valve (Right) Mean-centered flow readings with two abnormal instances (samples *a* and *b*)

For example, consider the scenario in Figure 5.7. The two flow measurements are expectedly correlated. Normal data-points lie along the 45° line (PC1 direction), except, instances '*a*' and '*b*' which exhibit different type of abnormalities. For sample '*a*', the correlation between the two flow variables is broken which may be the result of a leak in valve. This results in abnormally high Q_a value; T_a^2 however is not abnormally high because the projected score, t_a,

`

is similar to those of normal data-points. For sample '*b*', the correlation remains intact resulting in low (zero) Q_b value. The score, t_b, however, is abnormally far away from the origin resulting in abnormally high T_b^2 value.

Fault detection

It's time now to check whether our T^2 and Q charts can help us detect the presence of process abnormalities in test data (samples 70 onwards). For this, we will compute the monitoring statistics for the test data.

```
# get test data, normalize it
data_test = data.iloc[69:,]
data_test_normal = scaler.transform(data_test) # using scaling parameters from training data

# compute scores and reconstruct
score_test = pca.transform(data_test_normal)
score_test_reduced = score_test[:,0:n_comp]
data_test_normal_reconstruct = np.dot(score_test_reduced, P_matrix.T)

# calculate T2_test
T2_test = np.zeros((data_test_normal.shape[0],))
for i in range(data_test_normal.shape[0]): # eigenvalues from training data are used
    T2_test[i] = np.dot(np.dot(score_test_reduced[i,:],lambda_k_inv),score_test_reduced[i,:].T)

# calculate Q_test
error_test = data_test_normal_reconstruct - data_test_normal
Q_test = np.sum(error_test*error_test, axis = 1)
```

Figure 5.8 juxtaposes the monitoring statistics for training and test data. By looking at these plots, it is immediately evident that the test data exhibit severe process abnormality. Both T^2 and Q values are significantly above the respective control limits.

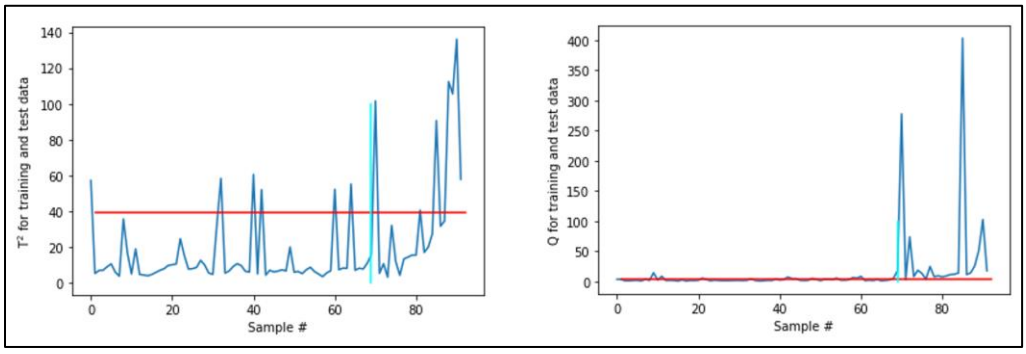


Figure 5.8: Monitoring charts for training and test data. Vertical cyan-colored line separates training and test data

`

Fault diagnosis

After detection of process faults, the next crucial task is to diagnose the issue and identify which specific process variables are showing abnormal behavior. The popular mechanism to accomplish this is based on contribution plots. As the name suggests, a contribution plot is a plot of the contribution of original process variables to the abnormality indexes. The variables with highest contributions are flagged as potentially faulty variables.

For SPE (squared prediction error), let's reconsider Eq. 7 as shown below where SPE_j denotes the SPE contribution of the j^{th} variable.

$$SPE = \sum_{j=1}^{m} e_j^2 = \sum_{j=1}^{m} SPE_j \qquad \text{eq. 9}$$

Therefore, SPE contribution of a variable is simply squared error for that variable. If SPE index has violated its control limit, then the variables with relatively large SPE_j values are considered the potentially faulty variables. For T^2 contributions, calculations are not as straight-forward. Several expressions have been postulated in literature[18]. The commonly used expression below was proposed by wise et al.[19].

$$T^2 \text{ contribution of variable } j = j^{th} \text{ element of } \left(\boldsymbol{D}^{1/2}\boldsymbol{x}\right)^2 \qquad \text{eq. 10}$$
$$\boldsymbol{D} = \boldsymbol{P}\boldsymbol{\Lambda}_k^{-1}\boldsymbol{P}^T$$

Note that these contributions are computed for each data-point. Let's find which variables need to be further investigated at 85th sample.

```
# T2 contribution
sample = 85 - 69
data_point = np.transpose(data_test_normal[sample-1,])

D = np.dot(np.dot(P_matrix,lambda_k_inv),P_matrix.T)
T2_contri = np.dot(scipy.linalg.sqrtm(D),data_point)**2 # vector of contributions

plt.figure()
plt.plot(T2_contri), plt.ylabel('T$^2$ contribution plot')

# SPE contribution
error_test_sample = error_test[sample-1,]
SPE_contri = error_test_sample*error_test_sample # vector of contributions
```

[18] S. Joe Qin, Statistical process monitoring: basics and beyond, Journal of Chemometrics, 2003
[19] Wise et. al., PLS toolbox user manual, 2006

`

```
plt.figure()
plt.plot(SPE_contri), plt.ylabel('SPE contribution plot')
```

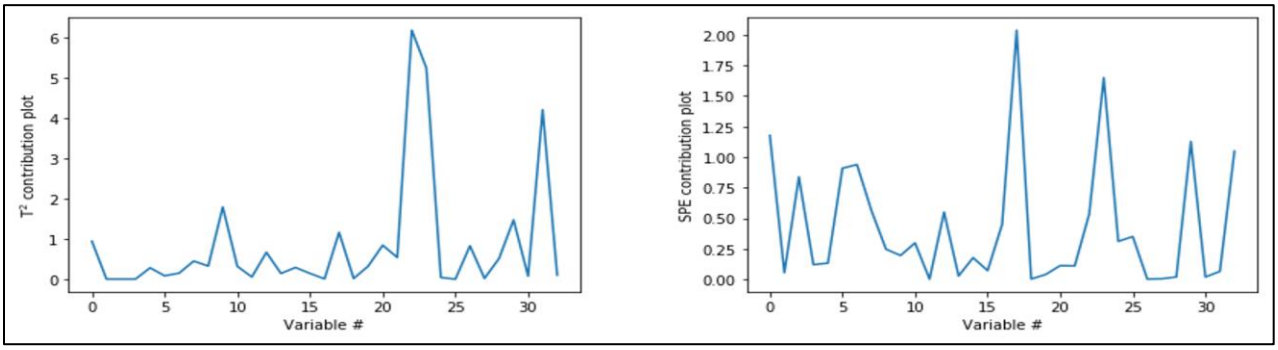


Figure 5.9: T^2 and SPE contribution plots for sample 85

Variable # 23 makes large contributions to both the indices and in Figure 5.10 we can see that there was a sharp decline in its value towards the end of the sampling period. A plant operator can use his/her judgement to further troubleshoot the abnormality to isolate the root-cause.

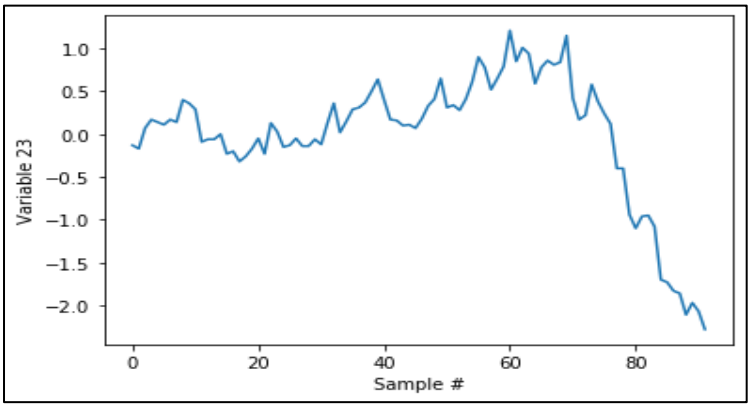


Figure 5.10: Temporal evolution of variable # 23

5.3 Variants of Classical PCA

Several implicit assumptions were made in the previously discussed PCA methodology. Latent variables were assumed to be a linear combination of measured variables, each sample was assumed to be statistically independent of past samples, and data were assumed to be Gaussian distributed. It is common to find these assumptions violated in process data and therefore, in this section, we will relax these assumptions and learn how to implement PCA for nonlinear and dynamic data. The methodologies primarily involve transforming raw data appropriately such that standard PCA can be applied on them to get satisfactory results.

`

Dynamic PCA (DPCA)

In previous section, we studied static PCA where static correlations among process variables were extracted. However, variables could be auto-correlated as well, i.e., the samples may not be temporally independent of each other. Such observations arise when samples are collected sequentially in time with small sampling period, for example, taking process data every few minutes over the last one year. Static PCA does not extract out the dynamic relationships resulting in auto-correlated errors which lead to unsatisfactory model performance such as high missed detection rate in process monitoring applications.

To tackle process dynamics, Ku et al.[20] proposed a simple extension of static PCA where past data-points are treated as additional process variables as shown in the illustration below.

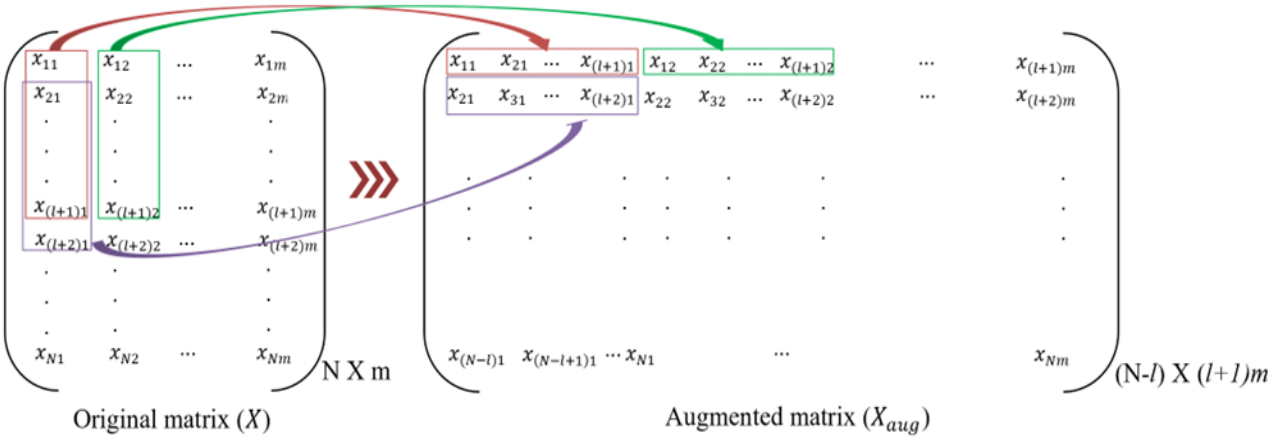


Illustration: Augmentation of lagged/past values to generate augmented data matrix

Each sample (except the first l samples) in the original data matrix has been augmented with data-points from the past l samples. Static PCA can now be performed on this augmented matrix after mean-centering and scaling just like we did before. Ku et al. have shown that this approach is efficient at extracting out linear static and dynamic correlations. A simple example of dynamic correlation could be

$$x_{t,1} = 0.8x_{t-1,1} + x_{t-1,2} \quad ; x \in \mathbb{R}^{2\times 1} \qquad \text{eq. 11}$$

Augmenting just one previous sample would suffice for such a system. Augmented sample row is similarly generated for test data when using the fitted PCA model to derive the scores for a test data-point. DPCA approach has proven to be useful in several industrial applications. The code below shows how to generate the augmented matrix with lag period of 5. Note that the lag period is a hyperparameter and its value needs to be set judiciously. Cross-validation

[20] Ku et al., Disturbance detection and isolation by dynamic principal component analysis, Chemometrics and intelligent laboratory systems, 1995

`

can be used. Another approach is to apply PCA on original data and observe the ACFs (auto-correlation function values) of the scores. Lagged values are augmented if autocorrelation is observed. This process is repeated until no autocorrelation remains.

```
#  augment training data
lag = 5
N = data_train.shape[0], m = data_train.shape[1]

data_train_augmented = np.zeros((N-lag,(lag+1)*m))
for sample in range(lag, N):
   dataBlock = data_train.iloc[sample-lag:sample+1,:].values # pandas dataframe to NumPy array
   data_train_augmented[sample-lag,:] = np.reshape(dataBlock, (1,-1), order = 'F')
```

Multiway PCA

PCA has been found useful for monitoring batch processes as well where raw data have an additional layer of variability, the inter-batch variability. To capture the variability in batch data, the three-dimensional data is unfolded to form a two-dimensional data matrix as shown in Figure 5.11. Static PCA is applied on this unfolded data matrix after normalizing each column[21]. Once a test batch data becomes available, test scores are obtained for further analysis.

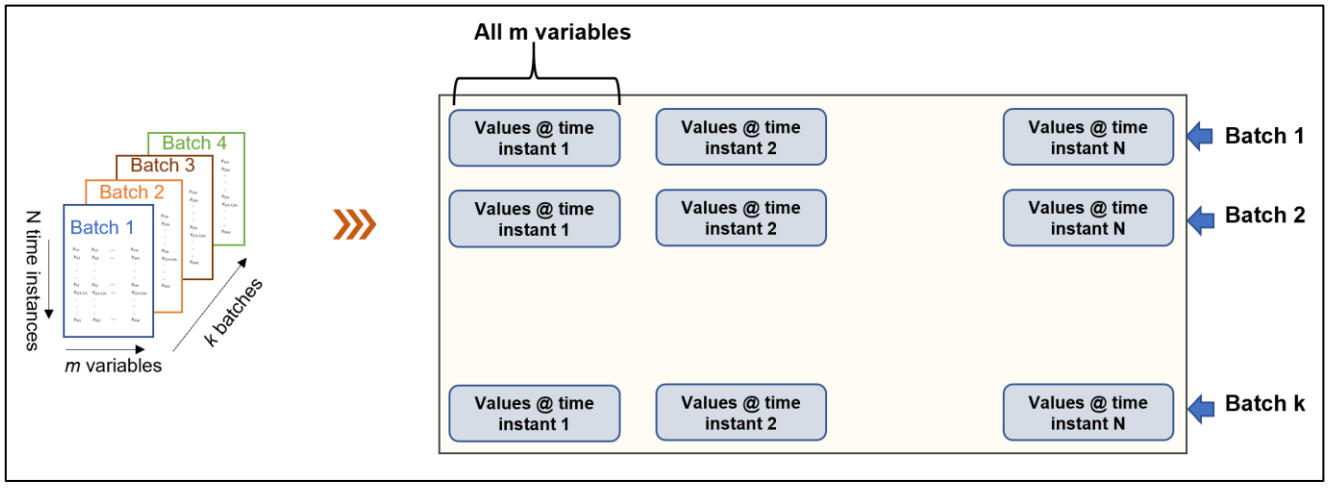


Figure 5.11: Unfolding batch data for multiway PCA

Note that in Figure 5.11, in each 'batch-row', batch data has been re-arranged by time-instances order. However, if desired, you can group them by variables as well. You only need to ensure that the same rearrangement is adopted for the test data as well. We will employ multiway PCA to a semiconductor manufacturing batch process data in Chapter 8.

[21] Nomikos & MacGregor, Monitoring batch processes using multiway principal component analysis, AIChE journal, 1994

`

Kernel PCA

Kernalized methods have become very attractive for dealing with nonlinear data while retaining the simplicity of their linear counterparts. For illustration, consider Figure 5.12 where, unlike in Figure 5.1, the variables are not related linearly. Nevertheless, it is apparent that the 3-dimensional data lie along a lower-dimensional manifold. However, the linear PCA-based abnormality detection will fail to detect the shown abnormal data-point.

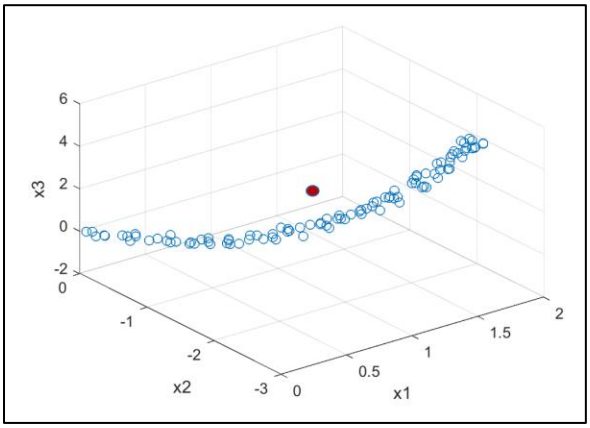


Figure 5.12: Nonlinearly related data with an abnormal (red) data-point

In such scenarios, the frequently employed solution is to transform the data into a higher-dimensional space where different classes (normal and abnormal classes) of data become linearly separable. To understand this trick, consider Figure 5.13, where the two classes of data are not linearly separable in the original measurement space. However, when an artificial variable ($x1^2 + x2^2$) is added to the system, the classes become easy to separate. This high-dimensional space is called the feature space.

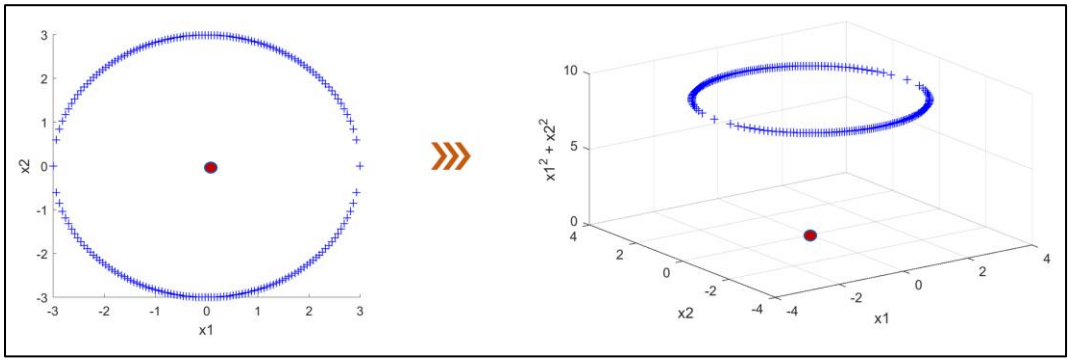


Figure 5.13: Mapping to feature space to enable linear separation

Kernel PCA (KPCA) simply entails implementing PCA in the feature space. However, the task of finding the nonlinear mapping that maps raw data to feature space is not trivial. To overcome this, another trick, called the kernel trick, is employed. While we will defer the detailed study of kernel trick until Chapter 7, the reader is encouraged to see the work by Lee

`

et al.[22] for details on computation of monitoring indexes for KPCA-based process monitoring. The code below shows how to analyze the data in Figure 5.12 using the KernelPCA library.

```
# fetch data and scale
data = pd.read_excel('KPCA_example.xlsx')
scaler = StandardScaler()
data_train_normal = scaler.fit_transform(data)

# fit KPCA model
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(kernel='rbf', gamma = 1)
score = kpca.fit_transform(data)
```

5.4 PLS: An Introduction

Partial least squares (PLS) is a supervised multivariate regression technique that estimates linear relationship between a set of input variables and a set of output variables. Like PCA, PLS transforms raw data into latent components - input (***X***) and output (***Y***) data matrices are transformed into score matrices ***T*** and ***U***, respectively. Figure 5.14 provides a conceptual comparison of PLS methodology with those of other popular linear regression techniques, principal component regression (PCR) and multivariate linear regression (MLR).

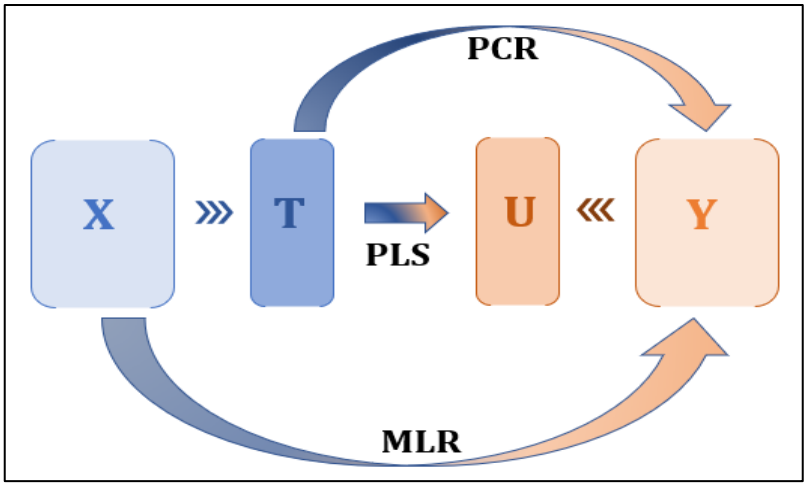


Figure 5.14: PLS, PCR, MLR methodology overview. Note that the score matrix, ***T***, for PLS and PCR can be different.

[22] Lee et al., Nonlinear process monitoring using kernel principal component analysis, Chemical Engineering Science, 2004

`

While MLR computes the least-squares fit between ***X*** and ***Y*** directly, PCR first performs PCA on input data and then computes least-squares fit between the score matrix and ***Y***. By doing so, PCR is able to overcome the issues of collinearity, high correlation, noisy measurements, and limited training dataset. However, the latent variables are computed independent of the output data and therefore, the score matrix may capture those variations in ***X*** which are not relevant for predicting ***Y***. PLS overcomes this issue by estimating the score matrices, ***T*** and ***U***, simultaneously such that the variation in ***X*** that is relevant for predicting ***Y*** is maximally captured in the latent variable space.

Note that if the number of latent components retained in PLS or PCR model is equal to the original number of input variables (m), then PLS and PCR models are equivalent to MLR model.

The unique favorable properties of PLS along with low computational requirements has led to its widespread usage in process monitoring for real-time process monitoring, soft-sensing, fault classification, and so on.

Mathematical background

PLS performs 3 simultaneous jobs:

- Capture maximum variability in ***X***
- Capture maximum variability in ***Y***
- Maximize correlation between ***X*** and ***Y***

To see how PLS achieves its objectives, consider again the data matrix $\boldsymbol{X} \in \mathbb{R}^{N \times m}$ consisting of *N* observations of *m* input variables where each row represents a data-point in the original measurement space. In addition, we also have an output data matrix with *p* (≥ 1) output variables, $\boldsymbol{Y} \in \mathbb{R}^{N \times p}$. It is assumed that each column is normalized to zero mean and unit variance in both the matrices. The first latent component scores are given by:

$$\boldsymbol{t}_1 = \mathbf{X}\boldsymbol{w}_1 \text{ and } \boldsymbol{u}_1 = \mathbf{Y}\boldsymbol{c}_1 \qquad \text{eq. 12}$$

The vectors $\boldsymbol{w}_1$ and $\boldsymbol{c}_1$, termed weight vectors, are computed such that the covariance between $\boldsymbol{t}_1$ and $\boldsymbol{u}_1$ are maximized. Referring to the definition of covariance, we can see that by maximizing the covariance, PLS tries to meet all the three objectives simultaneously.

$$Cov(\boldsymbol{t}_1, \boldsymbol{u}_1) = Correlation(\boldsymbol{t}_1, \boldsymbol{u}_1) * \sqrt{Var(\boldsymbol{t}_1)} * \sqrt{Var(\boldsymbol{u}_1)}$$

`

In the next step, loading vectors, $\boldsymbol{p}_1$ and $\boldsymbol{q}_1$, are found

$$\boldsymbol{X} = \boldsymbol{t}_1\boldsymbol{p}_1^T + \boldsymbol{E}_1 \text{ and } \boldsymbol{Y} = \boldsymbol{u}_1\boldsymbol{q}_1^T + \boldsymbol{F}_1 \qquad \text{eq. 13}$$

In Eq. 13, $\boldsymbol{E}$ and $\boldsymbol{F}$ are called residual matrices and represent the part of $\boldsymbol{X}$ and $\boldsymbol{Y}$ that have not yet been captured. To find the next component scores, the above three steps are repeated with matrices $\boldsymbol{E}_1$ and $\boldsymbol{F}_1$ replacing $\boldsymbol{X}$ and $\boldsymbol{Y}$. Note that the maximum number of possible components equals *m*. For each component, the weight vectors are found via iterative procedures like NIPALS or SIMPLS. The final PLS decomposition looks like the following

$$\boldsymbol{X} = \mathbf{T}\mathbf{P}^{\boldsymbol{T}} + \mathbf{E} = \sum_{i=1}^{k} \boldsymbol{t}_i\boldsymbol{p}_i^T + \boldsymbol{E}$$

$$\boldsymbol{Y} = \boldsymbol{U}\boldsymbol{Q}^{\boldsymbol{T}} + \boldsymbol{F} = \sum_{i=1}^{k} \boldsymbol{u}_i\boldsymbol{q}_i^T + \boldsymbol{F} \qquad \text{eq. 14}$$

where *k* is the number of latent components computed. The expressions in Eq. (12) are referred to as the outer relations for the X and Y blocks, respectively. An inner relation is also estimated for each pair $\{\boldsymbol{t}_i, \boldsymbol{u}_i\}$ via linear regression

$$\boldsymbol{u}_i \approx b_i\boldsymbol{t}_i$$

$$or \qquad \boldsymbol{U} = [\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_k] \approx [b_1\boldsymbol{t}_1, b_2\boldsymbol{t}_2, \cdots, b_k\boldsymbol{t}_k] = \boldsymbol{T}\boldsymbol{B}$$

where $\boldsymbol{B} = diag([b_1, b_2, \cdots, b_k])$ is a diagonal matrix of regression coefficients and connects the two blocks together; it is used to estimate an unknown $\boldsymbol{Y}$ for a given $\boldsymbol{X}$ through the latent variables as follows

$$\widehat{\boldsymbol{Y}} = \widehat{\boldsymbol{U}}\boldsymbol{Q}^{\boldsymbol{T}} = \boldsymbol{T}\boldsymbol{B}\boldsymbol{Q}^{\boldsymbol{T}} \qquad \text{eq. 15}$$

If these algorithmic details appear intimidating, do not worry. Sklearn provides the class PLSRegression which is very convenient to use as we will see in the next section where we will develop a PLS-based soft sensor.

5.5 Soft Sensing via PLS for Pulp & Paper Manufacturing Process

Soft sensors are used in process industry to provide estimates or predictions of key process outputs or product qualities using all other available process variable measurements. Soft sensing proves especially useful when cost of physical sensors is high or real-time product quality measurements are not available. For illustration, we will study 'Kamyr digester' dataset

`

from a pulp and paper manufacturing process. In this process, wood chips are processed into pulp whose quality is quantified by Kappa number. In the dataset, 301 hourly samples of the Kappa number and 21 other process variables are provided. Figure 5.15 shows that there is considerable variability in the product quality and our goal now is to develop a soft sensor application to predict Kappa number using other process data.

Before we build our PLS model, some pre-processing is needed. A quick glance at data indicates that there are a lot of missing values. To keep the analysis simple, we will remove variables with large number of missing values and then remove the samples with any missing value.

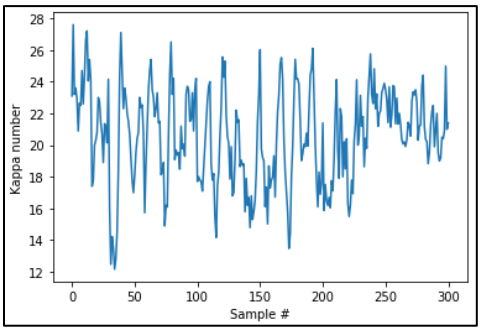


Figure 5.15: Kappa number data in Kamyr dataset

```
# import required packages
import numpy as np, pandas as pd
from sklearn.cross_decomposition import PLSRegression

# fetch data
data = pd.read_csv('kamyr-digester.csv', usecols = range(1,23))
# find the # of nan entries in each column
na_counts = data.isna().sum(axis = 0)

# remove columns that have a lot of nan entries
data_cleaned = data.drop(columns = ['AAWhiteSt-4 ','SulphidityL-4 '])

# remove any row that have any nan entry
data_cleaned = data_cleaned.dropna(axis = 0)

# separate X, y
y = data_cleaned.iloc[:,0].values[:, np.newaxis] # StandardScaler requires 2D array
X = data_cleaned.iloc[:,1:].values

print('Number of samples left: ', X.shape[0])

 >>> Number of samples left: 263
```

`

We now split the remaining samples into training data (80%) and test data (20%), and normalize them.

```
# separate training and test data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 100)

# scale data
from sklearn.preprocessing import StandardScaler
X_scaler = StandardScaler()
X_train_normal, X_test_normal = X_scaler.fit_transform(X_train), X_scaler.transform(X_test)

y_scaler = StandardScaler()
y_train_normal, y_test_normal = y_scaler.fit_transform(y_train), y_scaler.transform(y_test)
```

We are now ready to build the PLS model.

```
# PLS model
pls = PLSRegression(n_components = 9)
pls.fit(X_train_normal, y_train_normal)

# Training vs Test accuracy
y_train_normal_predict = pls.predict(X_train_normal)
y_test_normal_predict = pls.predict(X_test_normal)

print('Accuracy over training data: ', pls.score(X_train_normal, y_train_normal))
print('Accuracy over test data: ', pls.score(X_test_normal, y_test_normal))

>>> Accuracy over training data: 0.6615
>>> Accuracy over test data: 0.6812
```

Although not perfect, we have obtained a reasonably good inferential sensor. You are encouraged to change the number of latent components to maximum possible (19 in this case) to get MLR-equivalent accuracies. Negligible improvement in accuracies is obtained.

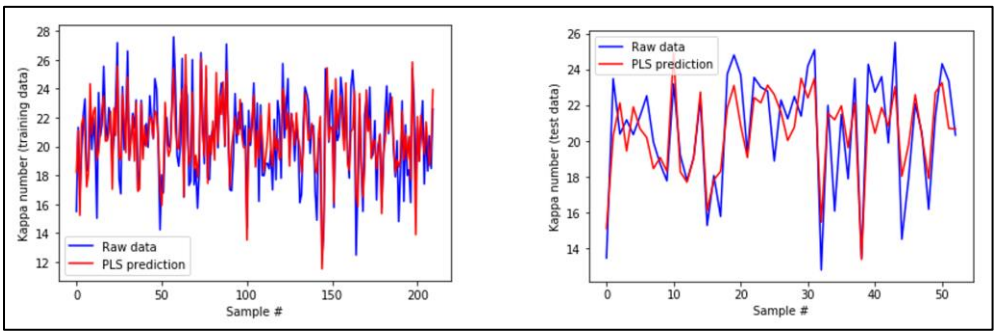


Figure 5.16: Measured vs predicted Kappa number for training and test data

`

A good rule of thumb when developing soft sensors is to always choose a model with lowest complexity if multiple models with similar predictive accuracies are available. For PLS and PCR, higher number of latent components imply higher complexity.

We used 9 latent components in our PLS model. This was determined via K-fold cross-validation procedure. As shown in code below, training data is split into 10 folds. For each possible n_comp, average of the MSE computed for each of the 10 folds is stored. For n_comp = 9, a local minimum can be observed in validation MSE plot.

```
# import required packages
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

scaler = StandardScaler()
fit_MSE = []
validate_MSE = []
for n_comp in range(1,20):

    local_fit_MSE = [] # store MSE for each fold
    local_validate_MSE = []

    kfold = KFold(n_splits = 10, shuffle = True, random_state = 100)
    for fit_index, validate_index in kfold.split(y_train):

        X_fit_normal = scaler.fit_transform(X_train[fit_index])
        X_validate_normal = scaler.transform(X_train[validate_index])

        y_fit_normal = scaler.fit_transform(y_train[fit_index])
        y_validate_normal = scaler.transform(y_train[validate_index])

        pls = PLSRegression(n_components = n_comp)
        pls.fit(X_fit_normal, y_fit_normal)

        local_fit_MSE.append(mean_squared_error(y_fit_normal, pls.predict(X_fit_normal)))
        local_validate_MSE.append(mean_squared_error(y_validate_normal,
                                                     pls.predict(X_validate_normal)))

    fit_MSE.append(np.mean(local_fit_MSE))
    validate_MSE.append(np.mean(local_validate_MSE))
```

`

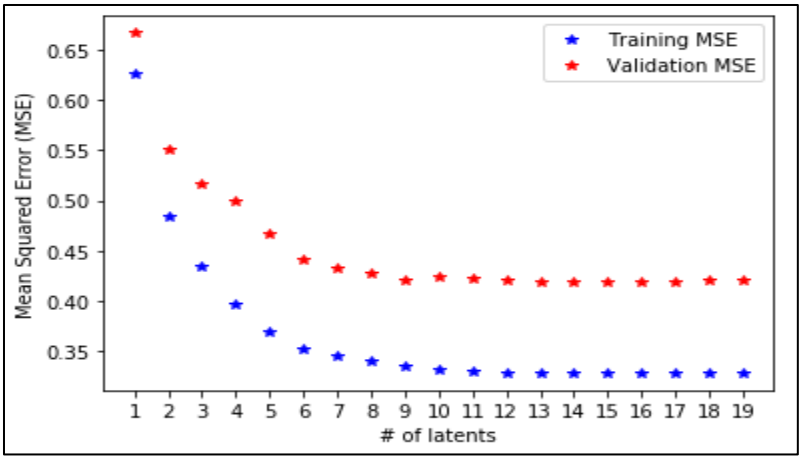


Figure 5.17: MSE plot for n_components determination

Determining number of retained components can be automated by simply looking at the ratio of validation MSE for consecutive n_comp. If the ratio is greater than some threshold (say 0.95), then the search is concluded. The underlying logic remains that the number of retained latents is not increased unless significantly better validation prediction is obtained.

5.6 Process Monitoring via PLS for Polyethylene Manufacturing Process

PLS framework renders itself useful for process monitoring as well. The overall methodology is similar to PCA-based monitoring: after PLS modeling, monitoring indices are computed, control limits are determined, and violation of the control limits are checked for fault detection. PLS-based monitoring is preferred when process data can be divided into input and output blocks. For illustration, we will use data collected from an LDPE (low-density polyethylene) manufacturing. The dataset consists of 54 samples of 14 process variables and 5 product quality variables. It is known that a process fault occurs sample 51 onwards (Figure 5.18).

Our objective here is to build a fault detection tool that clearly indicates the onset of process fault. To appreciate the need for such a tool, let's look at the alternative conventional monitoring approach. If a plant operator was manually monitoring the 5 quality variables continuously, he/she could notice a slight drop in values for the last 4 samples. However, given that the quality variables exhibit large variability during normal operations, it is difficult to make any decision without first examining other process variables because the quality variables may simply be responding to 'normal' changes elsewhere in the process. Unfortunately, it would be very inconvenient to manually interpret all the process plots simultaneously.

`

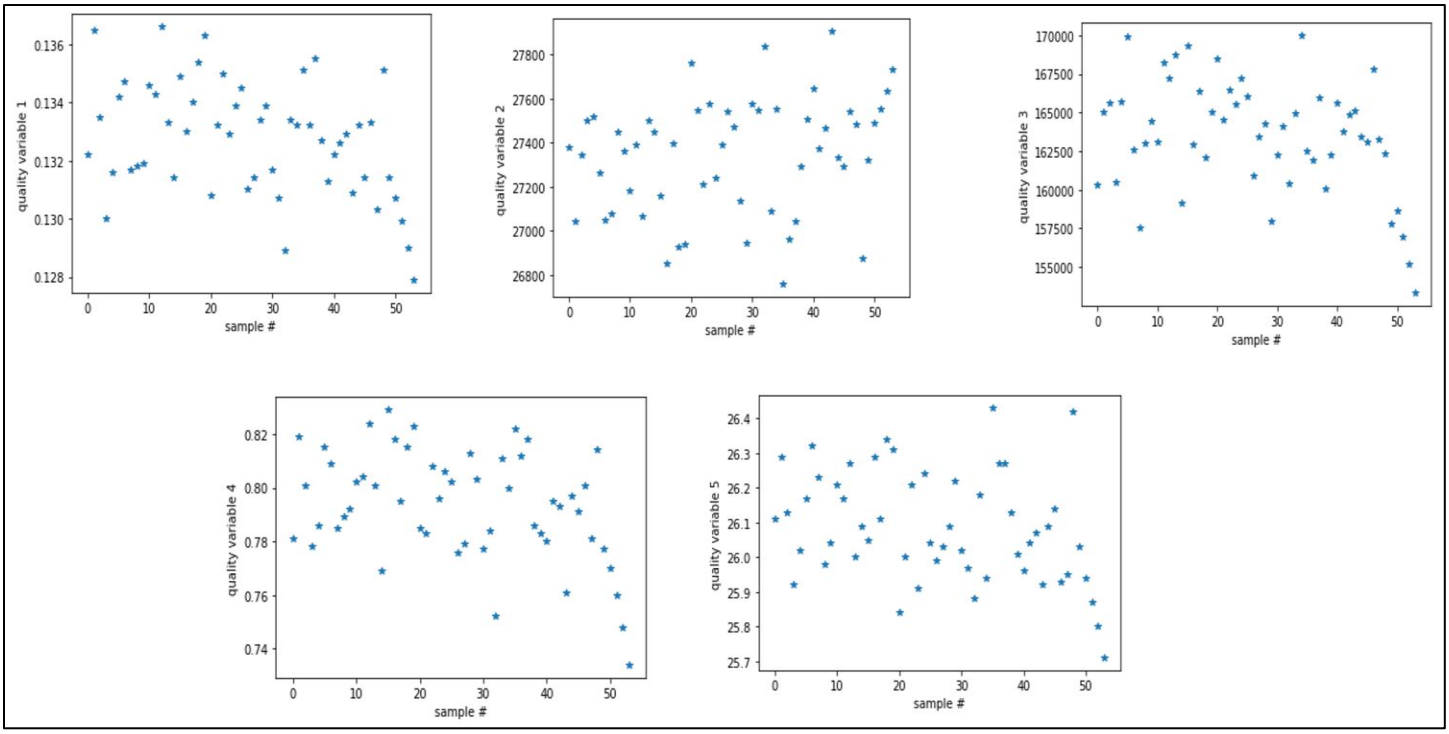


Figure 5.18: Plot of quality variables in LDPE dataset

We begin by building a PLS model using 3 latent components[23].

```
# fetch data
data = pd.read_csv('LDPE.csv', usecols = range(1,20)).values
data_train = data[:-4,:] # exclude last 4 faulty samples

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

# build PLS model
from sklearn.cross_decomposition import PLSRegression
X_train_normal = data_train_normal[:,:-5]
Y_train_normal = data_train_normal[:,-5:]

pls = PLSRegression(n_components = 3)
pls.fit(X_train_normal, Y_train_normal)
```

[23] Kourti & MacGregor, Process analysis, monitoring and diagnosis, using multivariate projection methods, Chemometrics and Intelligent Laboratory Systems, 1995

`

Computation of captured variances reveal that just 56% of the information in ***X*** can explain almost 90% of the variation in ***Y***; this implies that there are variations in ***X*** which have only minor impact on quality variables.

```
# X and Y variance captured
from sklearn.metrics import r2_score

print('Y variance captured: ', 100*pls.score(X_train_normal, Y_train_normal), '%')

Tscores = pls.x_scores_
X_train_normal_reconstruct = np.dot(Tscores, pls.x_loadings_.T)
# can also use pls.inverse_transform(Tscores)

print('X variance captured: ', 100*r2_score(X_train_normal, X_train_normal_reconstruct), '%')

>>> Y variance captured: 89.91 %
>>> X variance captured: 56.03 %
```

A look at ***t*** vs ***u*** score plots further confirms that linear correlation was a good assumption for this dataset. We can also see how the correlation becomes poor for higher components.

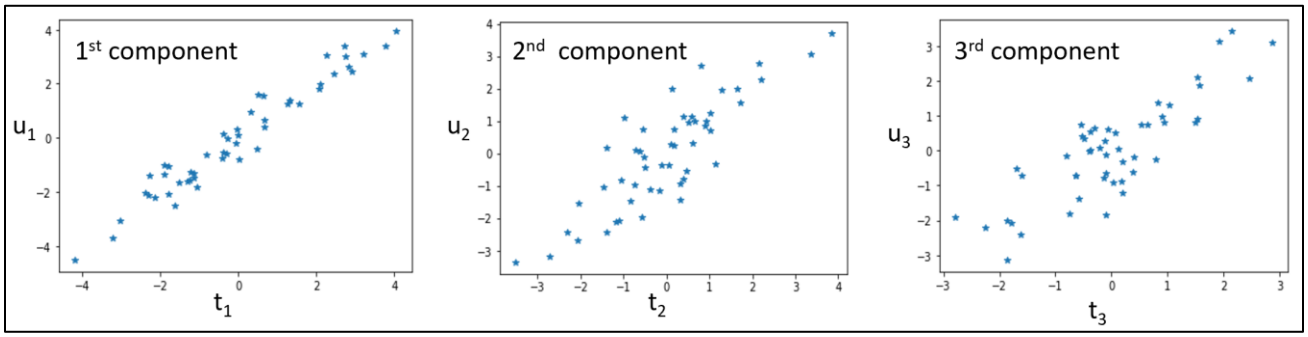


Figure 5.19: X-scores vs Y-scores. Here $\boldsymbol{t}_j$ and $\boldsymbol{u}_j$ refer to the j[th] columns of ***T*** and ***U*** matrices, respectively.

Fault detection indices

For PLS, 3 monitoring indices are computed: one T^2 statistic from t-scores, and two SPE statistics from ***X*** and ***Y*** residuals. Like in PCA, T^2 statistic quantifies the systematic variations in predictor variables (***X***) that are related to the systematic variations in response (***Y***) variables. Large deviation in T^2 statistic implies significant changes in process operating conditions. Let $\boldsymbol{t}_i$ denote the i^{th} row of ***T***. The T^2 index for this data-point is given by

$$T^2 = \sum_{j=1}^{k} \frac{t_{i,j}^2}{\sigma_j} = \boldsymbol{t}_i \boldsymbol{\Lambda}_k^{-1} \boldsymbol{t}_i^T$$

`

$\boldsymbol{\Lambda}_k$, a diagonal matrix, is the covariance matrix of $\boldsymbol{T}$ with σ_j (variance of j^{th} component scores) as its diagonal elements. T_{CL}^2 is again obtained by the following expression

$$T_{CL}^2 = \frac{k(N^2-1)}{N(N-k)} F_{k,N-k}(\alpha)$$

The second and third indices, SPE_x and SPE_y, represents the residuals or the unmodelled part of $\boldsymbol{X}$ and $\boldsymbol{Y}$, *respectively*. Let $\boldsymbol{e}_i$ and $\boldsymbol{f}_i$ denote the i^{th} row of $\boldsymbol{E}$ and $\boldsymbol{F}$, respectively. Then

$$SPE_x = \sum_{j=1}^{m} e_{i,j}^2$$

$$SPE_y = \sum_{j=1}^{p} f_{i,j}^2$$

Note that if output measurements are not available in real-time then SPE_y is not calculated. With normality assumption for the residuals, the control limit for SPE statistic is given by the following expression

$$SPE_{CL} = g\chi_\alpha^2(h)$$

$$h = \frac{2\mu^2}{\sigma}, \qquad g = \frac{\sigma}{2\mu}$$

χ_α^2 is the *(1-α)* percentile of a chi-squared distribution[24] with h degrees of freedom; μ denotes the mean value and σ denotes the variance of the SPE statistic. Note that this expression could be used for PCA SPE statistic as well.

```
# monitoring indices for training data
# T2
T_cov = np.cov(Tscores.T)
T_cov_inv = np.linalg.inv(T_cov)

T2_train = np.zeros((data_train_normal.shape[0],))
for i in range(data_train_normal.shape[0]):
   T2_train[i] = np.dot(np.dot(Tscores[i,:],T_cov_inv),Tscores[i,:].T)

# SPEx
x_error_train = X_train_normal - X_train_normal_reconstruct
SPEx_train = np.sum(x_error_train*x_error_train, axis = 1)

# SPEy
```

[24] Yin et al., A review of basic data-driven approaches for industrial process monitoring, IEEE Transactions on Industrial Electronics, 2014

`

```
y_error_train = Y_train_normal - pls.predict(X_train_normal)
SPEy_train = np.sum(y_error_train*y_error_train, axis = 1)

#  control limits
#T2
import scipy.stats
N = data_train_normal.shape[0]
k = 3

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

# SPEx
mean_SPEx_train = np.mean(SPEx_train)
var_SPEx_train = np.var(SPEx_train)

g = var_SPEx_train/(2*mean_SPEx_train)
h = 2*mean_SPEx_train**2/var_SPEx_train
SPEx_CL = g*scipy.stats.chi2.ppf(1-alpha, h)

# SPEy
mean_SPEy_train = np.mean(SPEy_train)
var_SPEy_train = np.var(SPEy_train)

g = var_SPEy_train/(2*mean_SPEy_train)
h = 2*mean_SPEy_train**2/var_SPEy_train
SPEy_CL = g*scipy.stats.chi2.ppf(1-alpha, h)
```

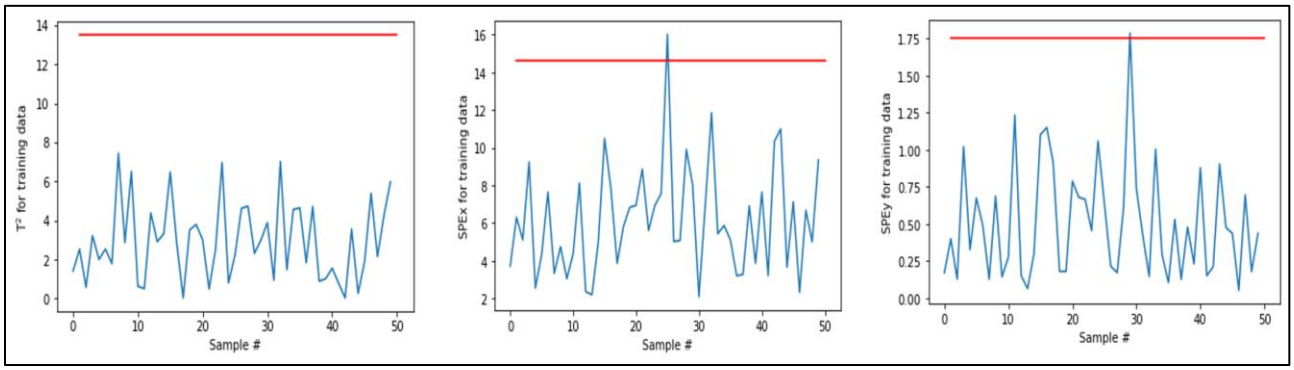


Figure 5.20: Monitoring charts for LDPE training data

Fault detection on test data

Let's see now if our monitoring statistics can detect the presence of process abnormality towards the end of the data samples. We will consider the whole dataset as test data for ease

`

of comparison. Monitoring charts clearly indicates that the process has encountered severe abnormality at the end of the sampling period. Significantly high SPE_x indicates that the abnormality has significantly affected input variable correlations.

```
# get test data, normalize it
data_normal = scaler.transform(data)
X_normal = data_normal[:,:-5]
Y_normal = data_normal[:,-5:]

# get model predictions
Tscores_test = pls.transform(X_normal)
X_normal_reconstruct = np.dot(Tscores_test, pls.x_loadings_.T)
Y_normal_pred = pls.predict(X_normal)

# compute monitoring statistics
T2_test = np.zeros((data_normal.shape[0],))
for i in range(data_normal.shape[0]):
    T2_test[i] = np.dot(np.dot(Tscores_test[i,:],T_cov_inv),Tscores_test[i,:].T)

x_error_test = X_normal - X_normal_reconstruct
SPEx_test = np.sum(x_error_test*x_error_test, axis = 1)

y_error_test = Y_normal - pls.predict(X_normal)
SPEy_test = np.sum(y_error_test*y_error_test, axis = 1)
```

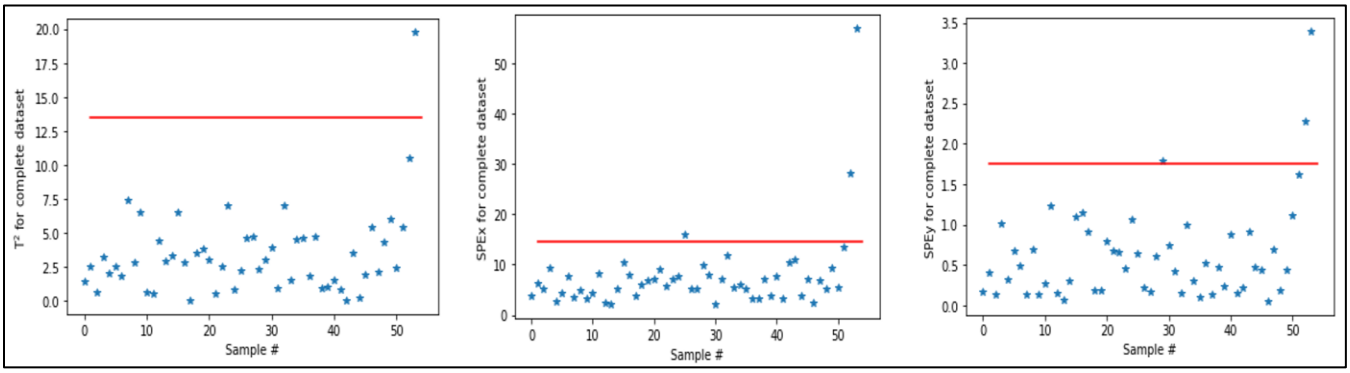


Figure 5.21: Monitoring charts for the complete LDPE dataset

5.7 Variants of Classical PLS

Variants of classical PLS have been devised to deal with dynamic and non-linear systems. For dynamic systems, past values of process inputs affect current values of outputs and

`

therefore including past measurements improve predictive accuracy[25]. For dynamic PLS, an augmented input data matrix is built (using the same procedure as that for dynamic PCA) using lagged values of only input variables or both input and output variables. Using only lagged inputs leads to a FIR (finite impulse response) model while using both lagged inputs and outputs leads to an ARX (autoregressive with exogenous variables) model. The value of lag period can be determined using cross-validation or time-constant of the process.

For nonlinear systems, kernel PLS is an efficient method for soft sensing quality variables. Using kernel function, the input variables are implicitly mapped onto a high-dimensional feature space where data behaves more linearly. Linear PLS model is then built between the feature variables and output variables. The reader is encouraged to see the work of Zhang et al.[26] for details on kernel PLS-based soft sensing.

Model maintenance for time-varying process

Model fidelity degrades over time due to changes in process correlations because of aging equipment or changing process conditions. Therefore, it becomes crucial to regularly update PCA/PLS models. There are two broad techniques of model adaptation: recursive update[++] and moving window update[**]. In moving-window approach, existing model is discarded, and a completely new process model is built by replacing oldest data with new data. In recursive approach, the existing model is not discarded, rather a new model is built by updating existing model with new data.

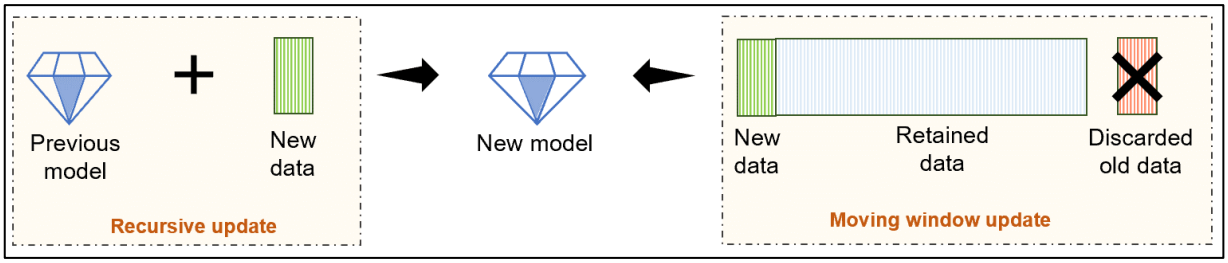


Figure 5.22: Model-maintenance schemes

[25] Kano et al., Inferential control system of distillation compositions using dynamic partial least squares regression, Journal of Process Control, 2000

[26] Zhang et al., Nonlinear multivariate quality estimation and prediction based on kernel partial least squares, Industrial Engineering & Chemistry Research, 2008

++ Li et al., Recursive PCA for adaptive process monitoring, Journal of Process Control, 2000

++ S. Joe Qin, Recursive PLS algorithms for adaptive data modeling, Computers & Chemical Engineering, 1998

** Wang et al., Process monitoring approach using fast moving window PCA

`

Summary

With this chapter we have reached a significant milestone in our ML journey. You have seen how hidden process knowledge can be conveniently extracted from process data and converted into process insights. With PCA and PLS tools in your arsenal you are now well-equipped to tackle most of the process modeling and monitoring related problems. However, our journey does not end here. In the next chapter, we will study a few more latent-variable-based techniques that are equally powerful.