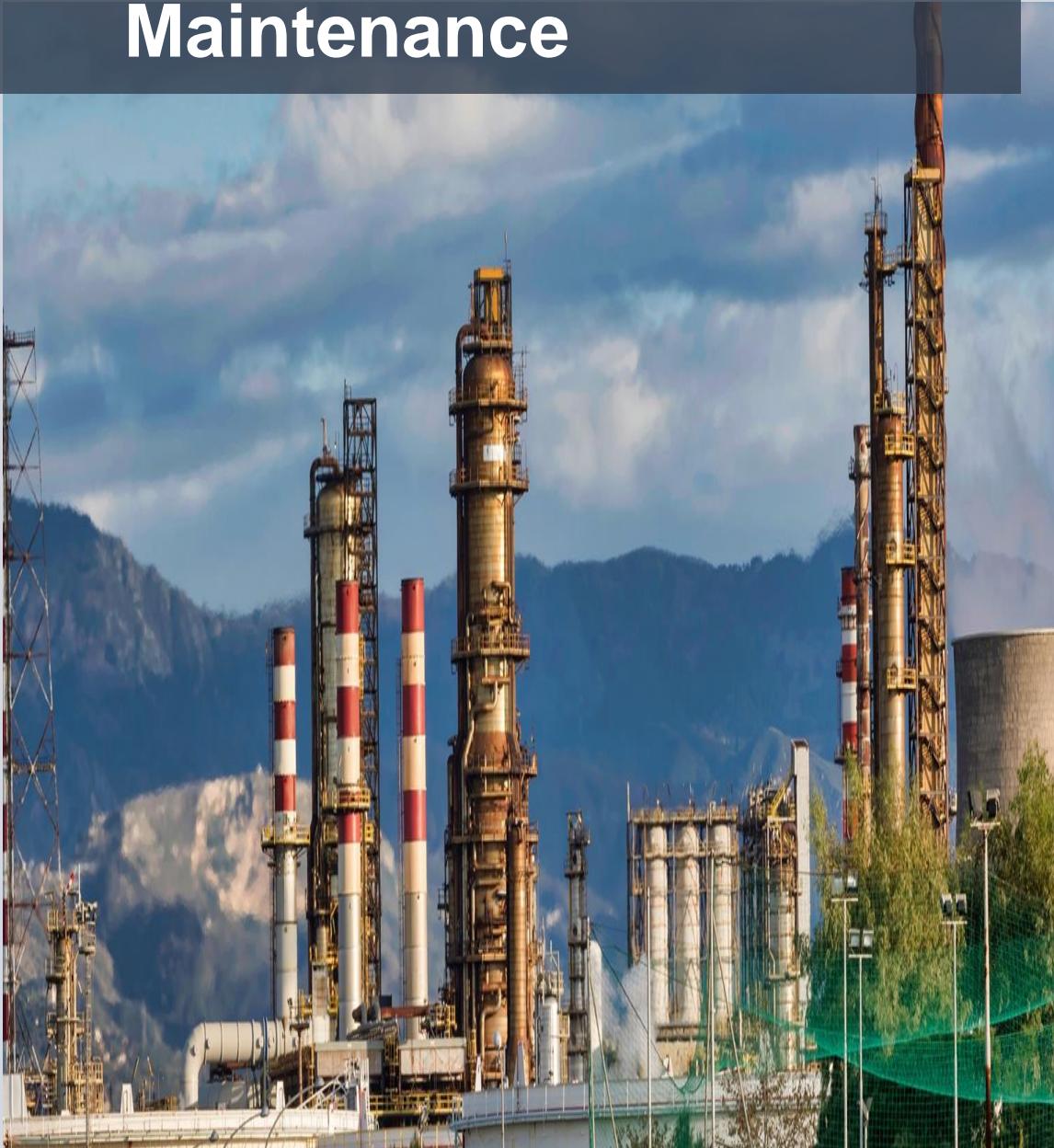


Machine Learning in Python for Process and Equipment Condition Monitoring, and Predictive Maintenance

From Data to Process Insights



2024



First Edition

Ankur Kumar, Jesus Flores-Cerrillo

Machine Learning in Python for Process and Equipment Condition Monitoring, and Predictive Maintenance

From Data to Process Insights

**Ankur Kumar
Jesus Flores-Cerrillo**

Dedicated to our spouses, families, friends, motherlands, and all the data-science enthusiasts



आचार्यत्यादमादते पादं शिष्यः स्वमेधया ।

पादं सब्रह्मचारिभ्यः पादं कालक्रमेण च ॥

*A student receives a quarter (of his/her learning) from the teacher,
a quarter by way of his/her intelligence,
a quarter from fellow students,
and a quarter through the course of time.*

- A popular Sanskrit shloka

Machine Learning in Python for Process and Equipment Condition Monitoring, and Predictive Maintenance

www.MLforPSE.com



Copyright © 2024 Ankur Kumar

All rights reserved. No part of this book may be reproduced or transmitted in any form or in any manner without the prior written permission of the authors.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented and obtain permissions for usage of copyrighted materials. However, the authors make no warranties, expressed or implied, regarding errors or omissions, and assume no legal liability or responsibility for loss or damage resulting from the use of information contained in this book.

Plant image on cover page obtained from <https://pixabay.com/>.

To request permissions, contact the authors at MLforPSE@gmail.com

First published: January 2024

About the Authors



Ankur Kumar holds a PhD degree (2016) in Process Systems Engineering from the University of Texas at Austin and a bachelor's degree (2012) in Chemical Engineering from the Indian Institute of Technology Bombay. He currently works at Linde in the Advanced Digital Technologies & Systems Group in Linde's Center of Excellence, where he has developed several in-house machine learning-based monitoring and process control solutions for Linde's hydrogen and air-separation plants. Ankur's tools have won several awards both within and outside Linde. One of his tools, PlantWatch (a plantwide fault detection and diagnosis tool), received the 2021 Industry 4.0 Award by the Confederation of Industry of the Czech Republic. Ankur has authored or co-authored several peer-reviewed journal papers (in the areas of data-driven process modeling and monitoring), is a frequent reviewer for many top-ranked Journals, and has served as Session Chair at several international conferences. Ankur served as an Associate Editor of the Journal of Process Control from 2019 to 2021, and currently serves on the Editorial Advisory Board of Industrial & Engineering Chemistry Research Journal. Most recently, he was included in the 'Engineering Leaders Under 40, Class of 2023' by *Plant Engineering Magazine*.



Jesus Flores-Cerrillo is currently an Associate Director - R&D at Linde and manages the Advanced Digital Technologies & Systems Group in Linde's Center of Excellence. He has over 20 years of experience in the development and implementation of monitoring technologies and advanced process control & optimization solutions. Jesus holds a PhD degree in Chemical Engineering from McMaster University and has authored or co-authored more than 40 peer-reviewed journal papers in the areas of multivariate statistics and advanced process control among others. His team develops and implements novel plant monitoring, machine learning, IIOT solutions to improve the efficiency and reliability of Linde's processes. Jesus's team received the Artificial Intelligence and Advanced Analytics Leadership 2020 award from the National Association of Manufacturers' Manufacturing Leadership Council.

Note to the readers

Jupyter notebooks and Spyder scripts with complete code implementations are available for download at https://github.com/ML-PSE/Machine_Learning_for_PM_and_PdM. Code updates, when necessary, will be made and updated on the GitHub repository. Updates to the book's text material will be available on Leanpub (www.leanpub.com) and Google Play (<https://play.google.com/store/books>). We would greatly appreciate any information about any corrections and/or typos in the book.

Series Introduction

In the 21st century, data science has become an integral part of the work culture at every manufacturing industry and process industry is no exception to this modern phenomenon. From predictive maintenance to process monitoring, fault diagnosis to advanced process control, machine learning-based solutions are being used to achieve higher process reliability and efficiency. However, few books are available that adequately cater to the needs of budding process data scientists. The scant available resources include: 1) generic data science books that fail to account for the specific characteristics and needs of process plants 2) process domain-specific books with rigorous and verbose treatment of underlying mathematical details that become too theoretical for industrial practitioners. Understandably, this leaves a lot to be desired. Books are sought that have process systems in the backdrop, stress application aspects, and provide a guided tour of ML techniques that have proven useful in process industry. This series '**Machine Learning for Process Industry**' addresses this gap to reduce the barrier-to-entry for those new to process data science.

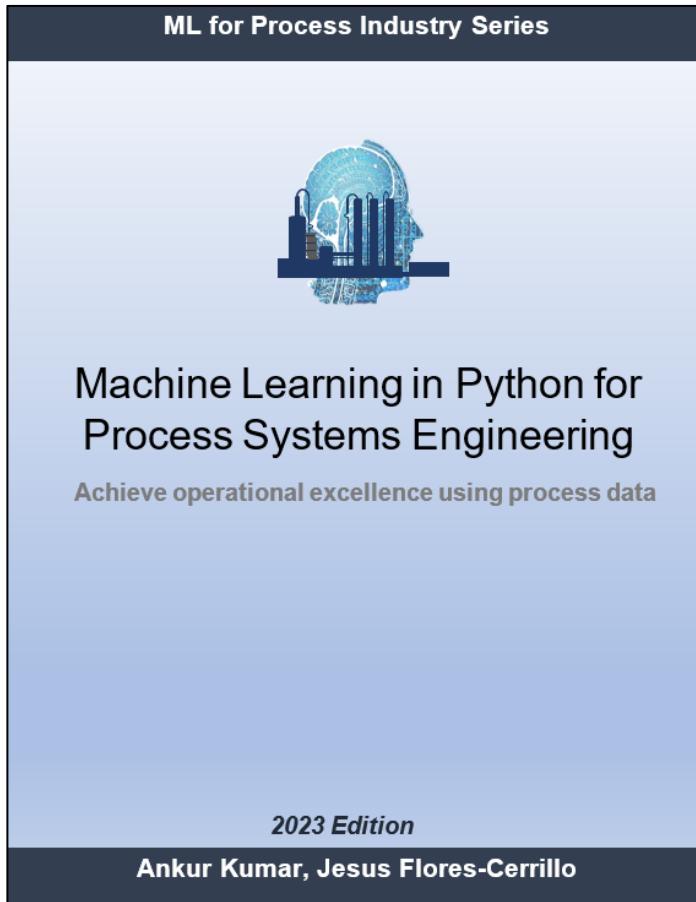
The first book of the series '**Machine Learning in Python for Process Systems Engineering**' covers the basic foundations of machine learning and provides an overview of broad spectrum of ML methods primarily suited for static systems. Step-by-step guidance on building ML solutions for process monitoring, soft sensing, predictive maintenance, etc. are provided using real process datasets. Aspects relevant to process systems such as modeling correlated variables via PCA/PLS, handling outliers in noisy multidimensional dataset, controlling processes using reinforcement learning, etc. are covered. The second book of the series '**Machine Learning in Python for Dynamic Process Systems**' focuses on dynamic systems and provides a guided tour along the wide range of available dynamic modeling choices. Emphasis is paid to both the classical methods (ARX, CVA, ARMAX, OE, etc.) and modern neural network methods. Applications on time series analysis, noise modeling, system identification, and process fault detection are illustrated with examples. This third book of the series takes a deep dive into an important application area of ML, viz, prognostics and health management. ML methods that are widely employed for the different aspects of plant health management, namely, fault detection, fault isolation, fault diagnosis, and fault prognosis, are covered in detail. Emphasis is placed on conceptual understanding and practical implementations. Future books of the series will continue to focus on other aspects and needs of process industry. It is hoped that these books can help process data scientists find innovative ML solutions to the real-world problems faced by the process industry.

With the growing trend in usage of machine learning in the process industry, there is growing demand for process domain experts/process engineers with data science/ML skills. These

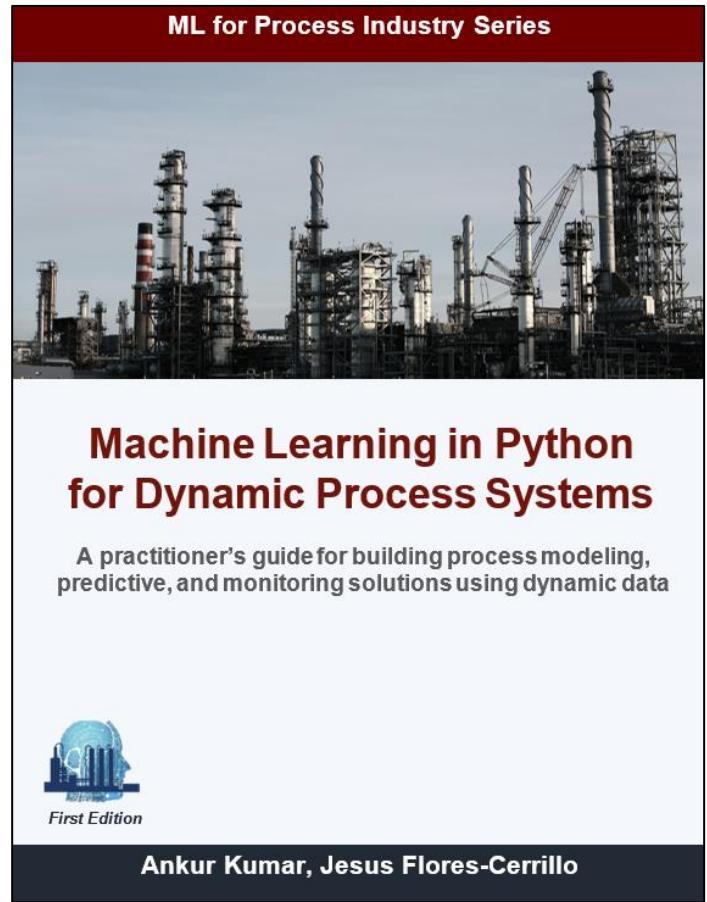
books have been written to cover the existing gap in ML resources for such process data scientists. Specifically, books of this series will be useful to budding process data scientists, practicing process engineers looking to ‘pick up’ machine learning, and data scientists looking to understand the needs and characteristics of process systems. With the focus on practical guidelines and industrial-scale case studies, we hope that these books lead to wider spread of data science in the process industry.

Other book(s) from the series
[\(https://MLforPSE.com/books/\)](https://MLforPSE.com/books/)

Book 1



Book 2



Preface

Imagine yourself in the shoes of a process engineer/analyst who has been assigned his/her first machine learning-based project with the objective of building a plantwide monitoring tool. Although an exciting task, it may easily turn into a frustrating effort due to the difficulty in finding the right methodology that works for the process system at hand. Building a successful process monitoring tool is challenging due to the different characteristics a process dataset may possess which precludes the possibility of a single methodology that works for all scenarios. Consequently, a number of powerful techniques have been devised over the past several decades. While it is good to be spoilt with choices, it is easy for a newcomer to get ‘drowned’ in the huge (and still burgeoning) literature on process monitoring (PM) and predictive maintenance (PdM). There are a lot of scattered resources on PM and PdM. However, unfortunately, no textbook exists that focusses on practical implementation aspects and provides comprehensive coverage of commonly used PM/PdM techniques that have proven useful in process industry. There is a gap in available machine learning resources for PM/PdM catering to industrial practitioners and this book attempts to cover this gap. Specifically, we wished to create a reader-friendly and easy-to-understand book that can help its readers become ‘experts’ on ML-based PM/PdM ‘quickly’ (disclaimer: there is no magic potion; hard work is still required!) with the right guidance.

In this book, we cover all three main aspects of process monitoring and predictive maintenance, namely, fault/anomaly detection, fault diagnosis/identification, and fault prognostics/remaining useful life estimation (RUL). Our intent is not to give a full treatise on all the PM/PdM techniques that exist out there; albeit our focus is to help budding process data scientists (PDSs) gain a bird’s-eye view of the PM/PdM landscape, obtain working-level knowledge of the mainstream techniques, and have the practical know-how to make the right choice of models. In terms of the spectrum of methodologies covered, we place equal emphasis on modern deep-learning methods and classical statistical methods. While deep-learning has provided remarkable results in recent times, the classical statistical (and ML/data mining) methods are not yet obsolete. Infact MSPM (multivariate statistical process monitoring) techniques are still widely used for process monitoring. Accordingly, this book covers the complete spectrum of methodologies with univariate Shewhart-/CUSUM-/EWMA-based control charts on one end and deep-learning-based RUL estimations on the other.

Guided by our own experiences from building monitoring models for varied industrial applications over the past several years, this book covers a curated set of ML techniques that have proven useful for PM/PdM. The broad objectives of the book can be summarized as follows:

- reduce barrier-to-entry for those new to the field of PM/PdM
- provide working-level knowledge of PM/PdM techniques to the readers
- enable readers to make judicious selection of PM/PdM techniques appropriate for their problems through intuitive understanding of the advantages and drawbacks of different techniques
- provide step-by-step guidance for developing industrial level solutions for PM/PdM
- provide practical guidance on how to choose model hyperparameters judiciously

This book adopts a tutorial-style approach. The focus is on guidelines and practical illustrations with a delicate balance between theory and conceptual insights. Hands-on-learning is emphasized and therefore detailed code examples with industrial-scale datasets are provided to concretize the implementation details. A deliberate attempt is made to not weigh readers down with mathematical details, but rather use it as a vehicle for better conceptual understanding. Complete code implementations have been provided in the GitHub repository.

We are quite confident that this text will enable its readers to build process monitoring and prognostics models for challenging problems with confidence. We wish them the best of luck in their career.

Who should read this book

The application-oriented approach in this book is meant to give a quick and comprehensive coverage of mainstream PM/PdM methodologies in a coherent, reader-friendly, and easy-to-understand manner. The following categories of readers will find the book useful:

- 1) Data scientists new to the field of process monitoring, equipment condition monitoring, and predictive maintenance
- 2) Regular users of commercial anomaly detection software (such as Aspen Mett) looking to obtain a deeper understanding of the underlying concepts
- 3) Practicing process data scientists looking for guidance for developing process monitoring and predictive maintenance solutions
- 4) Process engineers or process engineering students making their entry into the world of data science
- 5) Industrial practitioners looking to build fault detection and diagnosis solutions for rotating machinery using vibration data

Pre-requisites

No prior experience with machine learning or Python is needed. Undergraduate-level knowledge of basic linear algebra and calculus is assumed.

Book organization

The book follows a holistic and hands-on approach to learning ML where readers first gain conceptual insight and develop intuitive understanding of a methodology, and then consolidate their learning by experimenting with code examples. Under the broad theme of ML for process systems engineering, this book is an extension of the first two book of the series (which dealt with fundamentals of ML, varied applications of ML in process industry, and ML methods for dynamic system modeling); however, it can also be used as a standalone text. Industrial process data could show varied characteristics such as multidimensionality, non-Gaussianity, multimodality, nonlinearity, dynamics, etc. Therefore, to give due treatment to the different modeling methodologies designed for dealing with systems with different data characteristics, this book has been divided into seven parts.

Part 1 lays down the basic foundations of ML-assisted process and equipment condition monitoring, and predictive maintenance. **Part 2** provides in-detail presentation of classical ML techniques for univariate signal monitoring. Different types of control charts and time-series pattern matching methodologies are discussed. **Part 3** is focused on the widely popular multivariate statistical process monitoring (MSPM) techniques. Emphasis is paid to both the fault detection and fault isolation/diagnosis aspects. **Part 4** covers the process monitoring applications of classical machine learning techniques such as k-NN, isolation forests, support vector machines, etc. These techniques come in handy for processes that cannot be satisfactorily handled via MSPM techniques. **Part 5** navigates the world of artificial neural networks (ANN) and studies the different ANN structures that are commonly employed for fault detection and diagnosis in process industry. **Part 6** focusses on vibration-based monitoring of rotating machinery and **Part 7** deals with prognostic techniques for predictive maintenance applications.

This book attempts to cover a lot of concepts. Therefore, to avoid the book from getting bulky, we have not included contents that are not directly relevant to PM/PdM and have already been covered in detail in the first two books of the series. For example, ML fundamentals related to cross-validation, regularization, noise removal, etc., are illustrated in great detail in Book 1 of the series and not in this book.

Symbol notation

The following notation has been adopted in the book for representing different types of variables:

- lower-case letters refer to vectors ($x \in \mathbb{R}^{m \times 1}$) and upper-case letters denote matrices ($X \in \mathbb{R}^{n \times m}$)
- individual element of a vector and a matrix are denoted as x_j and x_{ij} , respectively.
- any i^{th} vector in a dataset gets represented as subscripted lower-case letter ($x_i \in \mathbb{R}^{m \times 1}$)

Table of Contents

Part 1: Introduction and Fundamentals

Chapter 1: Machine Learning, Process and Equipment Condition Monitoring, and Predictive Maintenance	1
1.1 Process Industry and ML-based Plant Health Management <ul style="list-style-type: none">-- What are process faults and abnormalities	
1.2 Plant Health Management (PHM) Workflow	
1.3 ML Modeling Landscape for Plant Health Management	
1.4 ML Model Development Workflow	
1.5 ML-based Plant Health Management Solution Deployment	
 Chapter 2: The Scripting Environment	 16
2.1 Introduction to Python	
2.2 Introduction to Spyder and Jupyter	
2.3 Python Language: Basics	
2.4 Scientific Computing Packages: Basics <ul style="list-style-type: none">-- Numpy, Pandas, Sklearn	
 Chapter 3: Exploratory Data Analysis: Getting to Know Your Data Well	 33
3.1 Why Exploratory Data Analysis Matters	
3.2 Nonlinearity Assessment Techniques	
3.3 Gaussianity Assessment Techniques	
3.4 Dynamics Assessment Techniques	
3.5 Multimode Distribution Assessment Techniques	
3.6 Data Characteristics Investigation of Tennessee Eastman Process Dataset	
 Chapter 4: Machine Learning for Plant Health Management: Workflow and Best Practices	 53
4.1 ML Model Development Workflow	
4.2 Data Selection	
4.3 Data Pre-processing <ul style="list-style-type: none">-- Handling data imbalance	
4.4 Model Evaluation	
4.5 Model Tuning	

Part 2: Univariate Signal Monitoring

Chapter 5: Control Charts for Statistical Process Control

67

- 5.1** Control Charts: Simple and Time-tested Process Monitoring Tools
- 5.2** Shewhart Charts: An Introduction
- 5.3** CUSUM Charts: An Introduction
- 5.4** EWMA Charts: An Introduction
- 5.5** Case Study: Monitoring Air Flow in an Aeration Tank
- 5.6** Pitfalls of Univariate Control Charts and Alternative Solutions

Chapter 6: Process Fault Detection via Time Series Pattern Matching

81

- 6.1** Time Series Anomalies and Pattern Matching
- 6.2** Fault Detection via Historical Pattern Search
- 6.3** Fault Detection via Discord Discovery

Part 3: Multivariate Statistical Process Monitoring

Chapter 7: Multivariate Statistical Process Monitoring for Linear and Steady-State Processes: Part 1

90

- 7.1** PCA: An Introduction
- 7.2** Fault Detection via PCA: Polymer Manufacturing Case Study
 - Fault detection indices
- 7.3** Fault Isolation via Contribution Analysis for PCA
- 7.4** PLS: An Introduction
- 7.5** Fault Detection via PLS: Polyethylene Manufacturing Case Study
 - Fault detection indices
- 7.6** Fault Isolation via Contribution Analysis for PLS

Chapter 8: Multivariate Statistical Process Monitoring for Linear and Steady-State Processes: Part 2

116

- 8.1** ICA: An Introduction
 - Deciding the number of independent components
- 8.2** Fault Detection via ICA: Tennessee Eastman Process Case Study
 - Fault detection indices
- 8.3** FDA: An Introduction
- 8.4** Fault Classification via ICA: Tennessee Eastman Process Case Study

Chapter 9: Multivariate Statistical Process Monitoring for Linear and Dynamic Processes

137

- 9.1** Dynamic PCA: An Introduction
- 9.2** DPCA-based Fault Detection
- 9.3** Dynamic PLS: An Introduction
- 9.4** Canonical Variate Analysis (CVA): An Introduction
- 9.5** Process Monitoring of Tennessee Eastman Process via CVA

Chapter 10: Multivariate Statistical Process Monitoring for Nonlinear Processes

156

- 10.1** Kernel PCA: An Introduction
- 10.2** Fault Detection using Kernel PCA
- 10.3** Kernel PLS: An Introduction
- 10.4** Fault Detection using Kernel PLS

Chapter 11: Process Monitoring of Multimode Processes

174

- 11.1** Need and Methods for Specialized Handling of Multimode Processes
- 11.2** Multimode Semiconductor Manufacturing dataset
- 11.3** K-means Clustering: An Introduction
- 11.4** Gaussian Mixture Modeling: An Introduction
 - Deciding the number of clusters
- 11.5** Fault Detection via GMM: Semiconductor Manufacturing Case Study

Part 4: Classical Machine Learning Methods for Process Monitoring

Chapter 12: Support Vector Machines for Fault Detection

196

- 12.1** SVMs: An Introduction
 - Hard margin vs soft margin classification
- 12.2** The Kernel Trick for Nonlinear Data
 - Sklearn implementation of support vector classifier
- 12.3** SVDD: An Introduction
- 12.4** Fault Detection via SVDD: Semiconductor Manufacturing Case Study

Chapter 13: Decision Trees and Ensemble Learning for Fault Detection

214

- 13.1 Decision Trees: An Introduction
- 13.2 Random Forests: An Introduction
- 13.3 Fault Classification using Random Forests: Gas Boiler Case Study
- 13.4 Introduction to Ensemble Learning
 - Bagging
 - Boosting
- 13.5 Fault Classification using XGBoost: Gas Boiler Case Study

Chapter 14: Proximity-based Techniques for Fault Detection

234

- 14.1 KNN: An Introduction
 - Application for fault detection for metal-etch process
- 14.2 LOF: An Introduction
 - Application for fault detection for metal-etch process
- 14.3 Isolation Forest: An Introduction
 - Application for fault detection for metal-etch process

Part 5: Artificial Neural Networks for Process Monitoring**Chapter 15: Fault Detection & Diagnosis via Supervised Artificial Neural Networks Modeling**

249

- 15.1 ANN: An Introduction
- 15.2 Process Modeling via FFNN: Combined Cycle Power Plant Case Study
- 15.3 RNN: An Introduction
- 15.4 ANN-based External Analysis for Fault Detection in a Debutanizer Column
- 15.5 Fault Classification using ANNs

Chapter 16: Fault Detection & Diagnosis via Unsupervised Artificial Neural Networks Modeling

265

- 16.1 Autoencoders: An Introduction
 - Dimensionality reduction via autoencoders
- 16.2 Process Monitoring using Autoencoders: FCCU Case Study
 - Fault diagnosis via contribution plot
- 16.3 Self-Organizing Maps: An Introduction
 - Evaluating SOM fit
- 16.4 Visualization of Semiconductor Dataset via SOM
- 16.5 Process Monitoring using SOM: Semiconductor Case Study
 - fault diagnosis via contribution plot

Part 6: Vibration-based Condition Monitoring

Chapter 17: Vibration-based Condition Monitoring: Signal Processing and Feature Extraction

287

- 17.1** Vibration: A Gentle Introduction
- 17.2** Vibration-based Condition Monitoring: Workflow
- 17.3** Vibration Signal Processing
 - Frequency domain analysis
 - Time-frequency domain analysis
- 17.4** Feature Extraction from Vibration Signals
 - Time domain features
 - Frequency domain features
 - Time-frequency domain features

Chapter 18: Vibration-based Condition Monitoring: Fault Detection & Diagnosis

305

- 18.1** VCM Workflow: Revisited
- 18.2** Classical VCM Approaches: A Quick Primer
- 18.3** Machine Learning-based VCM: Motor Fault Classification via SVM

Part 7: Predictive Maintenance

Chapter 19: Fault Prognosis: Concepts & Methodologies

314

- 19.1** Fault Prognosis: Introduction & Workflow
- 19.2** Machinery health Indicators: Introduction & Approaches
- 19.3** Health Indicator Construction Using Vibration Signals for a Wind Turbine

Chapter 20: Fault Prognosis: RUL Estimation

327

- 20.1** RUL: Revisited
- 20.2** Health Indicator-based RUL Estimation Strategies
 - Health degradation modeling
 - Trajectory similarity-based modeling
- 20.3** RUL Estimation via Degradation Modeling for a Wind Turbine
- 20.4** RUL Estimation via ANN-based Regression Modeling for a Gas Turbine

Part 1

Introduction & Fundamentals

Chapter 1

Machine Learning, Process and Equipment Condition Monitoring, and Predictive Maintenance: An Introduction

Ask a plant manager about what gives him/her sleepless nights and you will invariably get plant equipment failures and process abnormalities causing downtimes among the top answers. Such concerns about plant reliability are not unfounded. Incipient abnormalities, if left undetected, can cause cascading damages leading to economic losses, plant downtimes, and even fatalities. Several major disasters in the process industry (Philadelphia refinery explosion in 2019, Bhopal gas tragedy in 1984, etc.) were the results of failures in timely detection and correction of process faults. While such disasters are fortunately infrequent, ‘innocuous’ process abnormalities that lead to non-optimal plant efficiencies and degradations in product quality occur routinely. Without exaggeration, it can be said that 24X7 monitoring of process performance and plant equipment health status, and forecast of impending failures are no longer a ‘nice to have’ but an absolute necessity!

Process industry has responded to the above challenges by putting more sensors and collecting more real-time data. Unfortunately, this has led to data deluge and operators being overwhelmed with ‘too much information but little insights’. Thankfully, machine learning comes to the rescue with its ability to parse huge amount of data and find hidden patterns in real-time. ML allows smart process monitoring (PM) wherein objective is not just to detect process abnormalities but to catch the issues at early stages. Furthermore, ML facilitates predictive maintenance (PdM) through advance prediction of equipment failure times.

In this chapter we will take a bird’s-eye view of the ML landscape for PM/PdM and understand what it takes to achieve the above objectives. Specifically, the following topics are covered

- Introduction to process/equipment abnormalities and faults
- Typical workflow for ML-based process monitoring and predictive maintenance
- ML landscape for process monitoring and predictive maintenance
- Common PM/PdM solution deployment infrastructure employed in industry

Machine learning is a great tool, but it’s not magic; it still takes a lot ‘ML art’ to get the things right. Let’s now take the first step towards mastering this art.

1.1 Process Industry and ML-based Plant Health Management

Process industry is a parent term used to refer to industries like petrochemical, oil & gas, chemical, power, paper, cement, pharmaceutical, etc. These industries use processing plants to manufacture intermediate or final consumer products. As emphasized in Figure 1.1, the prime concerns of the management of these plants include, amongst others, optimal and safe operations, quality control, and high reliability through proactive process monitoring and predictive maintenance. All these tasks fall under the ambit of process systems engineering (PSE). While ML is being slowly incorporated in the PSE tasks (for example, deep learning-based process controller¹), ML has had the biggest influence on the tasks related to plant health management, viz, fault detection, fault diagnosis, and predictive maintenance.

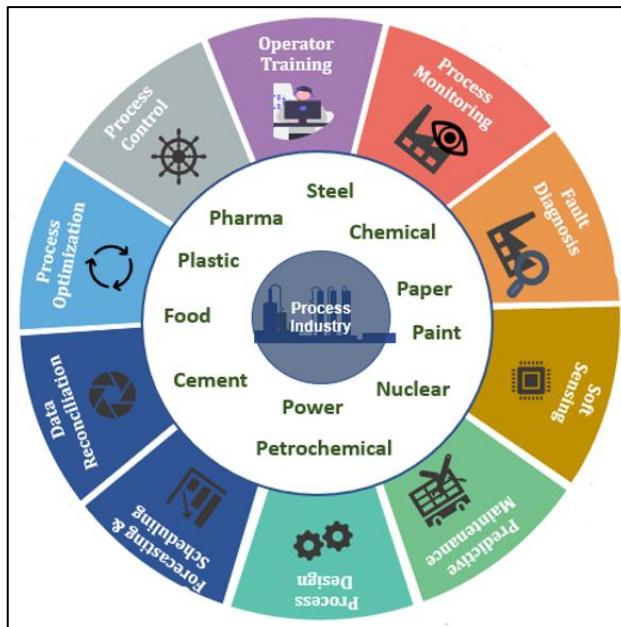


Figure 1.1: Overview of industries constituting process industry and the common PSE tasks

Figure 1.2 shows a sample process flowsheet with traditional measurements of flow, temperature, pressure, level, composition, power, and vibration. Such complex and highly integrated operations, tight product specifications, and the economic compulsion to push processes to their limits are making industrial operations more prone to failures. Nonetheless, there is an increasing trend to have unmanned or lean-staffed plants with less human eyes to monitor the process. This is where automated plant health management comes into play to resolve this dichotomy. Process models combined with sensor data are used for continuous monitoring of processes to detect, isolate, and diagnose faults, and for predicting fault

¹ <https://www.aspentech.com/en/products/msc/aspen-dmc3>

progression. The obvious gains are prevention of costly downtimes through better planned maintenance. For developing process models, data-driven/ML models have become more popular due to the relative ease of implementation and model maintenance compared to first principle-based models.

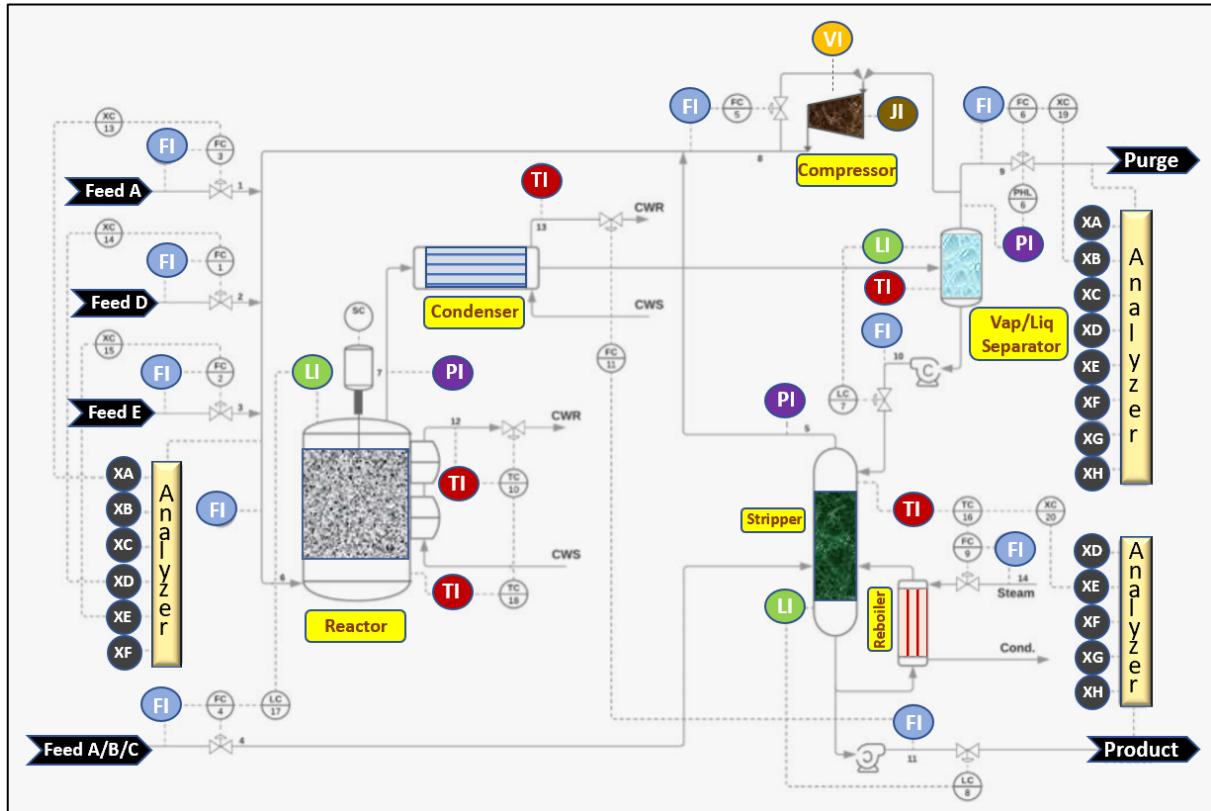


Figure 1.2: A typical process flowsheet² with flow (FI), temperature (TI), pressure (PI), composition (Analyzers), level (LI), power (JI), vibration (VI) measurements.

Let's continue learning about the ML-based plant health management by first taking a closer look at the meaning of process faults and abnormalities.

What are process faults and abnormalities?

Colloquially speaking, process faults or abnormalities are unexpected and unfavorable deviations/patterns in process variables that defy the normal/acceptable process behavior. The deviations could be undesirable decreases in product yield and product purity, fluctuations in critical liquid levels, rise in temperatures, increase in rotating machine vibrations, etc. There are various causes of faults in process system including, amongst others, fouling, pipe blockages, leaks, catalyst poisoning, and valve stiction. The flowsheet below illustrates some common fault sources.

² Adapted from the original flowsheet by Gilberto Xavier (<https://github.com/gmxavier/TEP-meets-LSTM>) provided under Creative-Commons Attribution 4.0 International License (<https://creativecommons.org/licenses/by/4.0/>).

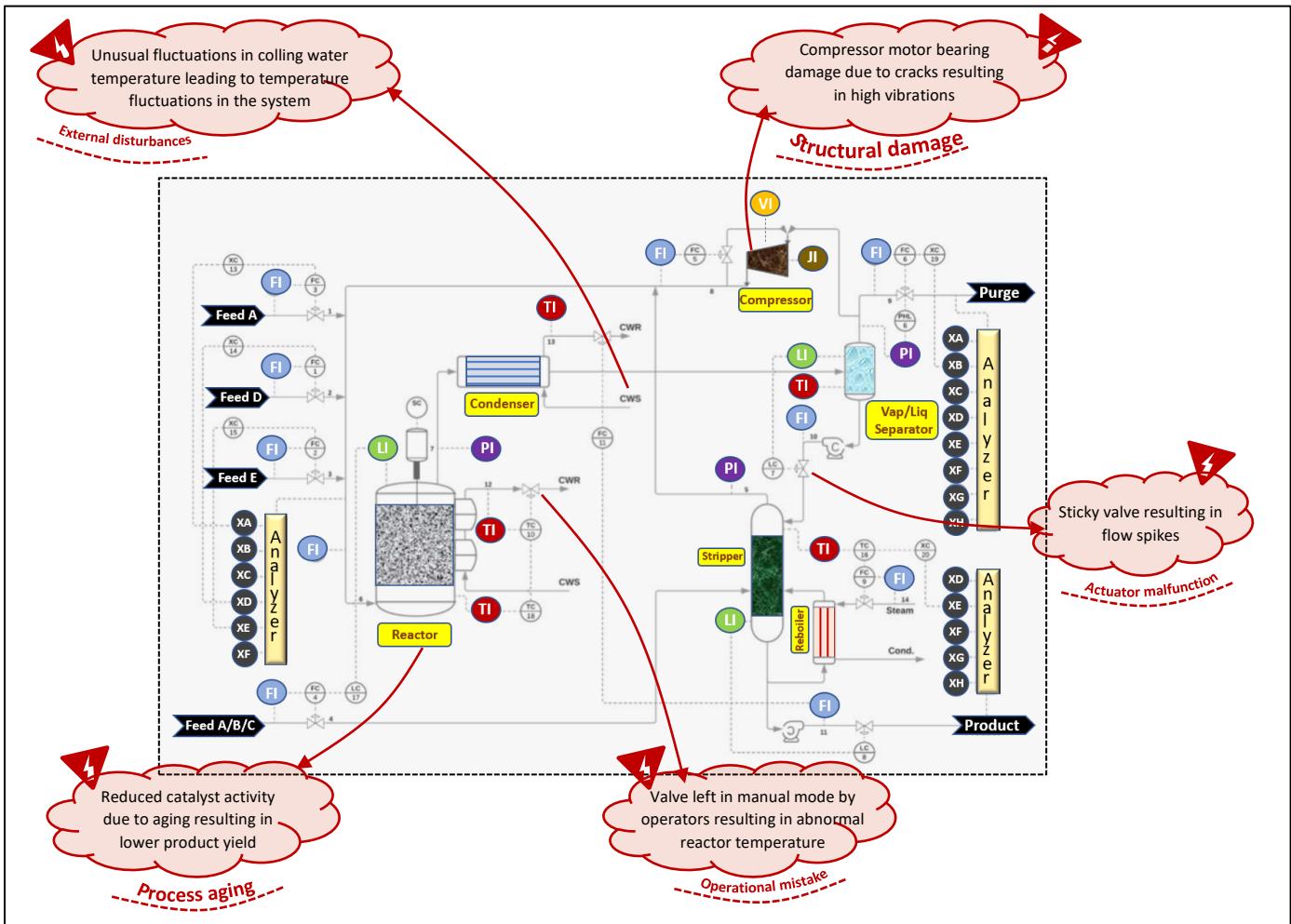


Figure 1.3: Some common sources of process faults in a process plant

Equipment monitoring vs plantwide monitoring

A common approach for process monitoring in process industry is to monitor the different critical equipment of a plant separately. The ML models are built separately for each equipment. While this approach makes ML model development easier as each ML model handles only a subset of the plant variables, one is left with having to maintain and analyze results from multiple ML models. An alternative approach is plantwide monitoring wherein the whole plant comprising of multiple equipment is monitored using a single ML model. The downside of this approach is high dimensionality of the variable-set and reduced fault detection performance. Same ML models can be employed for either of the approaches. Nonetheless, some specialized techniques have been devised for plantwide monitoring. We will remark upon these techniques as suitable in the upcoming chapters.

As remarked before, faults entail unwanted deviations in process variables. Figure below shows samples of data patterns that may be observed under the influence of process faults.

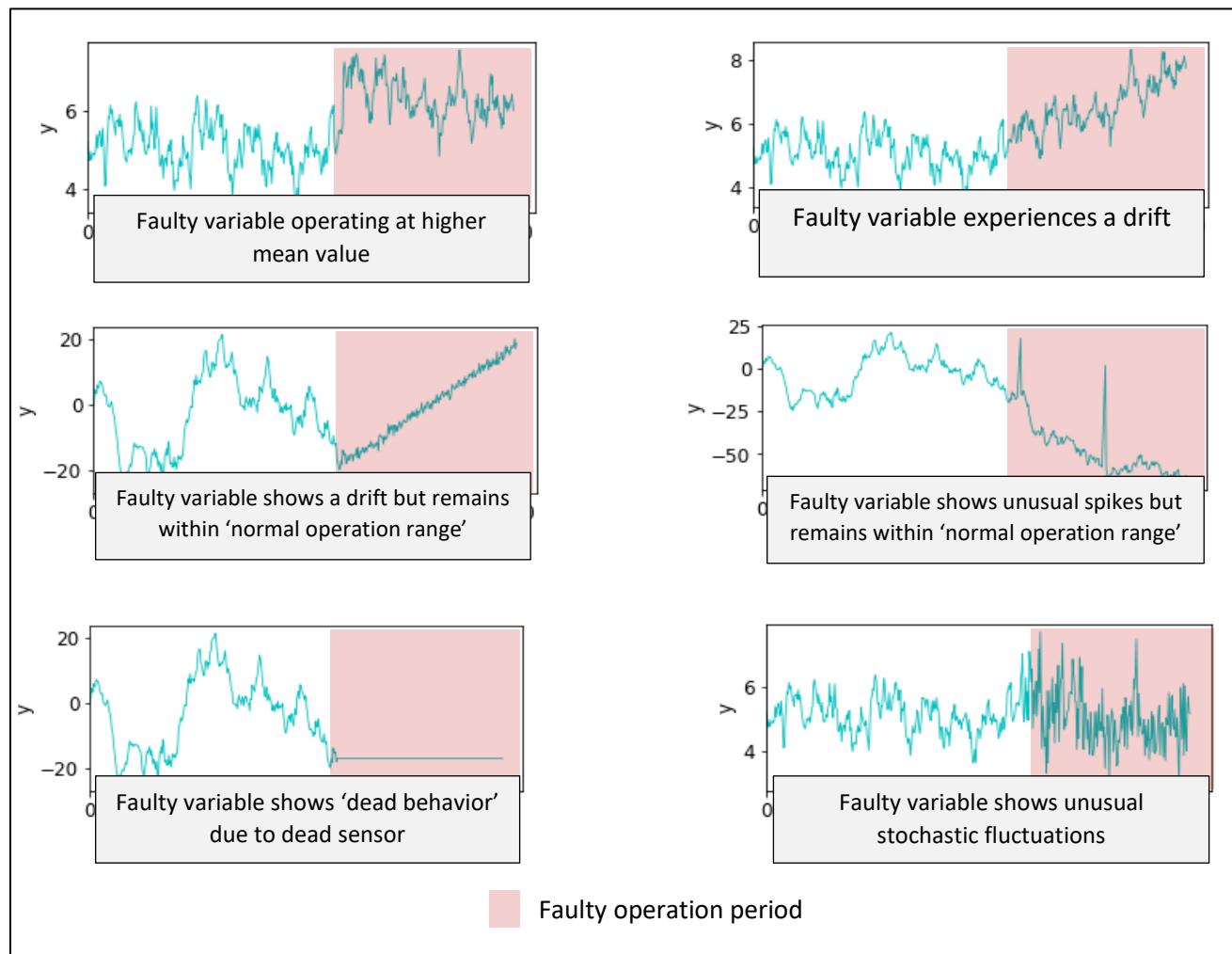


Figure 1.4: Sample of data patterns under faulty conditions

Figure 1.4 shows why automated fault detection is not a very straightforward activity. Under normal plant operation, process variables do not remain at fixed values but show stochastic fluctuations and normal variations due to changing plant load, product grade, ambient conditions, etc. Therefore, one can't just compare each process variable against some fixed thresholds to ascertain the healthy state of the process. Modeling the multivariable relationships among the plant variables become indispensable in most of the scenarios.

The takeaway message is that modern process plants are prone to multiple failures, and it takes an 'army' to ensure reliable operations. In the industry 4.0 era, ML is being employed as that 'army'. Before we look at the different ML models available at our disposal, let's try and understand what exactly an ML model is expected to do.

1.2 Plant Health Management (PHM) Workflow

In the previous section we discussed in some detail the fault detection aspect of plant health management. However, it is only a part of the journey towards reliable plant operation. Figure 1.5 shows the different milestones of the journey. As shown, fault detection is followed by fault isolation or fault diagnosis wherein the objective is to identify the process variables that have been affected by the fault or determine the underlying cause of the fault, respectively. For example, for the valve malfunction problem illustrated in Figure 1.3, fault isolation pinpoints the flow from the separator to the stripper as the faulty variable and fault diagnosis pinpoints the valve stiction as the root cause of faulty behavior.

FDI vs FDD



In the process monitoring literature, you will find the acronyms FDI and FDD very often. FDI stands for fault detection and isolation, and FDD stands for fault detection and diagnosis. As alluded to before, although fault diagnosis is different from fault isolation, it is often used (incorrectly) to refer to the task of finding variables showing abnormal behavior.

Other terms that you may encounter are fault identification and fault classification. While fault identification is same as fault isolation, fault classification refers to categorizing/classifying a fault into one of several pre-defined fault types.

Following FDI/FDD, lies the task of fault prognosis which entails forecasting the progression of the identified fault. Fault prognosis helps to determine the amount of time left before the equipment affected by the fault needs to be taken out of service for maintenance or the whole plant needs to be shutdown for fault repair. For equipment-level monitoring, fault prognosis provides what is popularly known as remaining useful life (RUL). For example, for the compressor bearing damage problem illustrated in Figure 1.3, the vibrations will only be slightly higher than normal during the initial stages of crack development. However, with time the crack grows leading to greater and greater vibrations, and ultimately the compressor fails or becomes too dangerous to operate. A good fault prognostic model can accurately estimate the time left until the failure point of compressor is reached.

The advancement in fault prognosis algorithms have popularized the concept of predictive maintenance, wherein the plant management can plan well-in-advance the maintenance

schedule based on actual equipment/process health condition. As you can imagine, this approach has obvious economic benefits (compared to time-based/preventive maintenance) and, to nobody's surprise, has caught fascination of process industry executives!

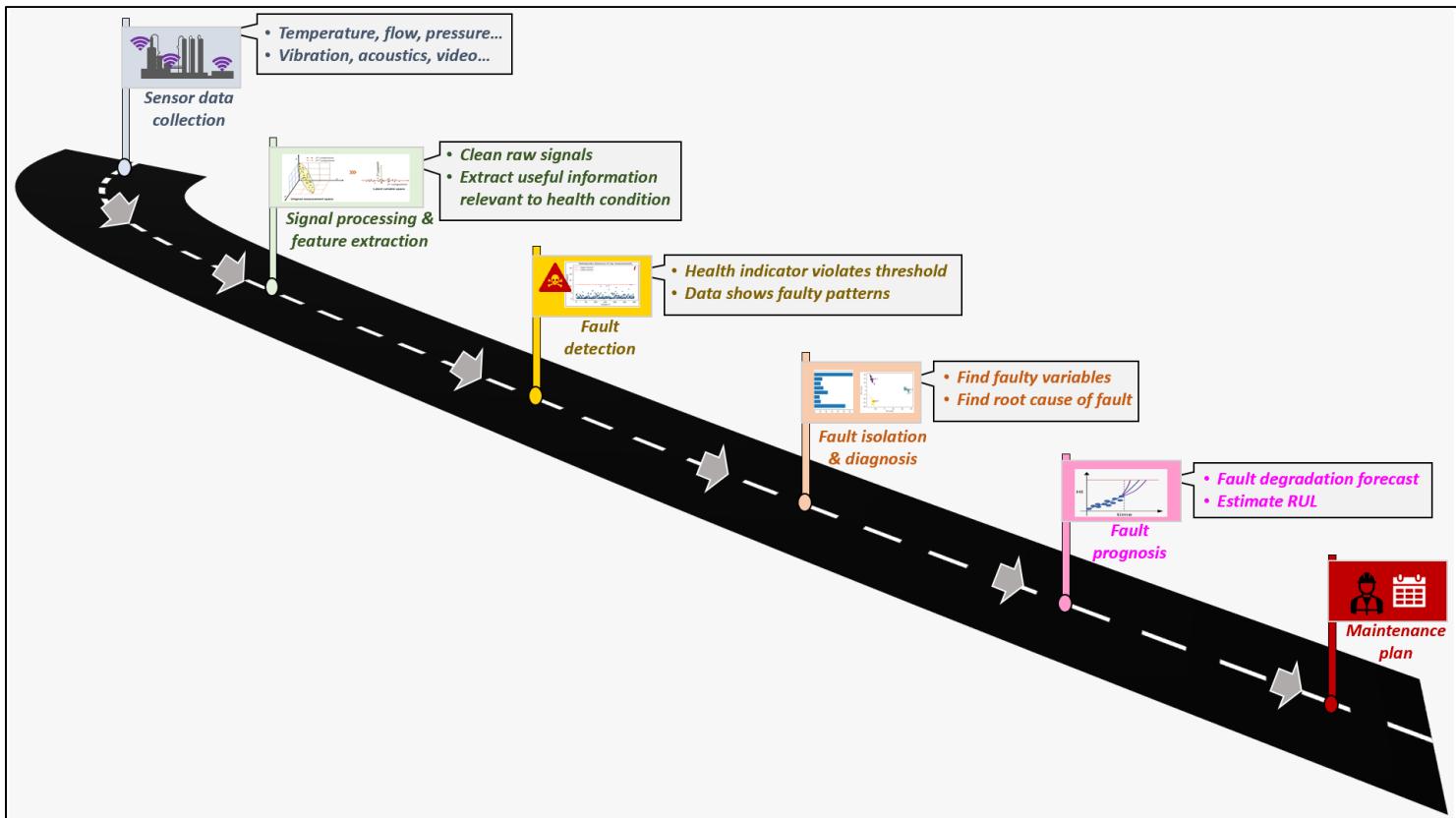


Figure 1.5: Plant health management workflow

Prognostic and Health Management

In industrial community, the PM/PdM workflow shown in Figure 1.5 is commonly called as prognostic and health management³. This includes both condition monitoring (fault detection, isolation, and diagnosis) and predictive maintenance (fault prognosis) aspects.

In the upcoming chapters, we will study in detail all the shown major aspects of PHM and learn how to implement the end-to-end solutions.

³ Although the acronym 'PHM' is commonly used by the prognostic research community to refer to prognostic and health management, we will use it to denote 'plant health management' in this book.

1.3 ML Modeling Landscape for Plant Health Management

As process data scientists, we have to live with the harsh truth that there is no single universally good model for all occasions. One reason for this is that process data can show different characteristics (such as nonlinearity, non-Gaussianity, dynamics, multi-modality, etc.) which necessitates selection of different modeling methodologies. Additionally, the availability of historical faulty data, the user's end goal, and the type of installed sensors can also influence the model selection as shown in Figure 1.6. This makes the task of correct selection of ML model daunting (and potentially overwhelming for beginner PDSs). Fortunately, the recourse is open-secret and is as simple as having a good understanding of your data and system, and conceptually sound knowledge of pros and cons of the available methods.

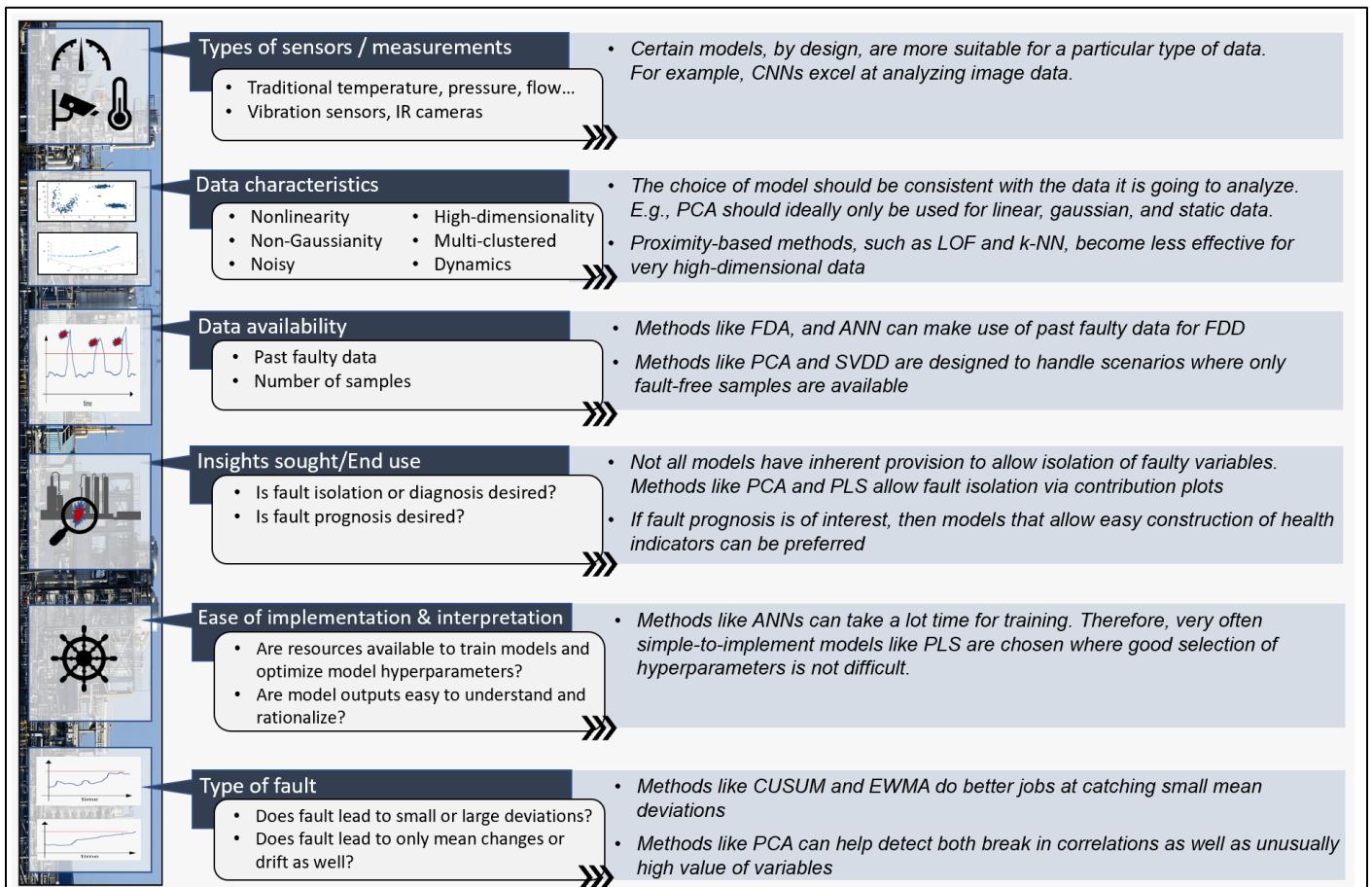


Figure 1.6: Sample factors that influence ML model selection for PM/PdM

Before you embark upon modeling your process system, you would already have knowledge of the various factors listed in the above figure, except possibly for the data characteristics. We will study the techniques used to ascertain data characteristics in Chapter 3. Now that we understand the factors that influence model selection, we are ready to see what models are available at our disposal.

 *Traditional process measurements such as flow, temperature, pressure, composition, etc., and vibration measurements dominate the signals recorded in process industry. Therefore, case studies presented in this book use only these signals. Computer vision-based ML solutions are not covered.*

Figure 1.7 below shows the modeling methodologies for process monitoring that we will cover in this book. Fault detection and diagnosis are precursors to fault prognosis and therefore the same methodologies are employed for building predictive maintenance solutions as well. As the category topics show, these methods cater to process data with different characteristics. The methods range from ‘simple’ traditional control charts to modern deep learning.

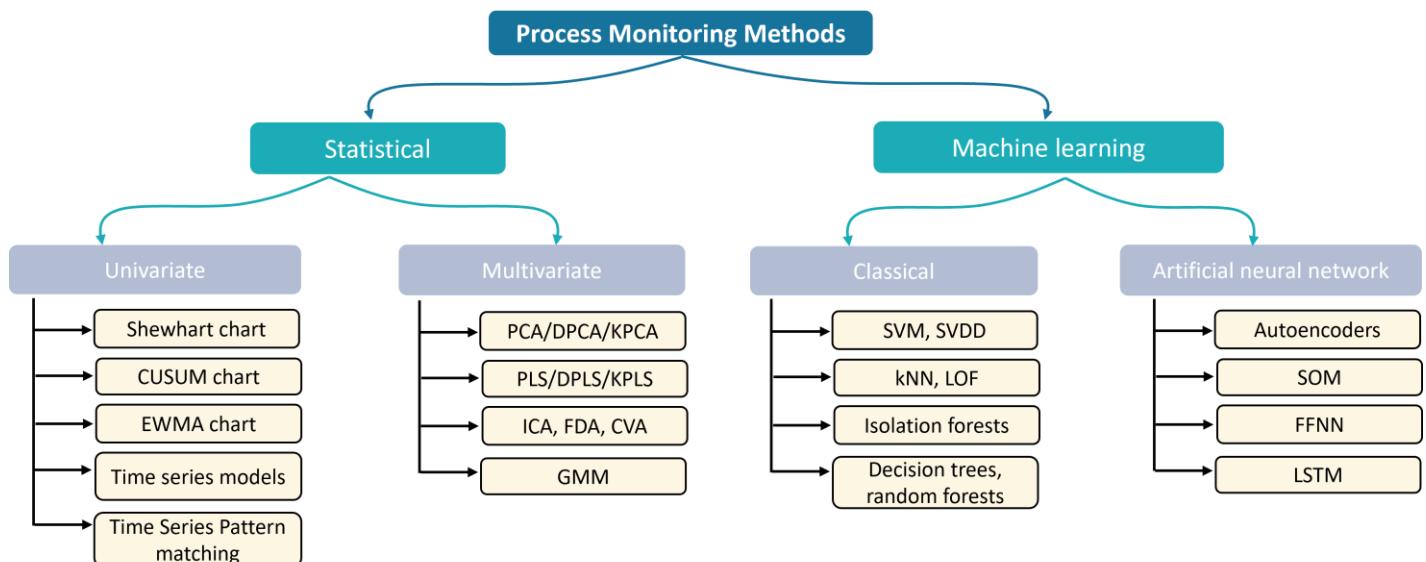


Figure 1.7: Model tree for process monitoring

The category of MSPM (multivariate statistical process monitoring) methods (PCA, PLS, GMM, etc.) deserves special attention as it has been the bedrock of health monitoring of complex process plants. A large section of the book will therefore cater to these methods. However, irrespective of their popularity, MSPM methods have shortcomings. Therefore, machine learning and deep learning models like Autoencoders, LSTMs, LOFs are covered as well.

In Figure 1.7, the modeling methodologies have been broadly divided into four categories, viz, univariate statistical models, multivariate statistical models, classical machine learning models, and artificial neural networks (ANN) models. Each of these categories are dealt with in separate parts of the book. The statistical⁴ PM models extract a statistical model of the system using past data. Within this category lies simple control-chart models that are used for single variable monitoring. Though useful, these univariate models are understandably too restrictive to handle plantwide monitoring of complex industrial plants. On the other end of the model spectrum lies complex deep learning models that can theoretically handle any type of process systems; the downside is cumbersome model training procedure and hyperparameter optimization. In between these two extremes, lie the MSPM methods whose ease of implementation and interpretable results have led to wide popularity. However, MSPM methods tend to falter for highly nonlinear processes with complex data distributions. Therefore, classical ML and deep learning methods have been receiving considerable attention for process monitoring solution development for complex industrial processes.

The models in Figure 1.7 cater to the different scenarios that you may encounter in practice. If you have abundant past faulty samples then classification models such as FDA, SVM, ANN, etc. can be employed. However, in process industry, most of the time you will not have the luxury of having past faulty data and therefore, many of the fault detection techniques covered in this book cater to this scenario. The figure below illustrates the different principles employed to detect the presence of process faults using only NOC data during model training.

⁴ In legacy process monitoring terminology, statistical process monitoring is also called statistical process control (SPC). Although SPC methods do not involve any feedback to the process controllers, the word ‘control’ signifies the objective of keeping the process ‘in-control’ through continuous monitoring.

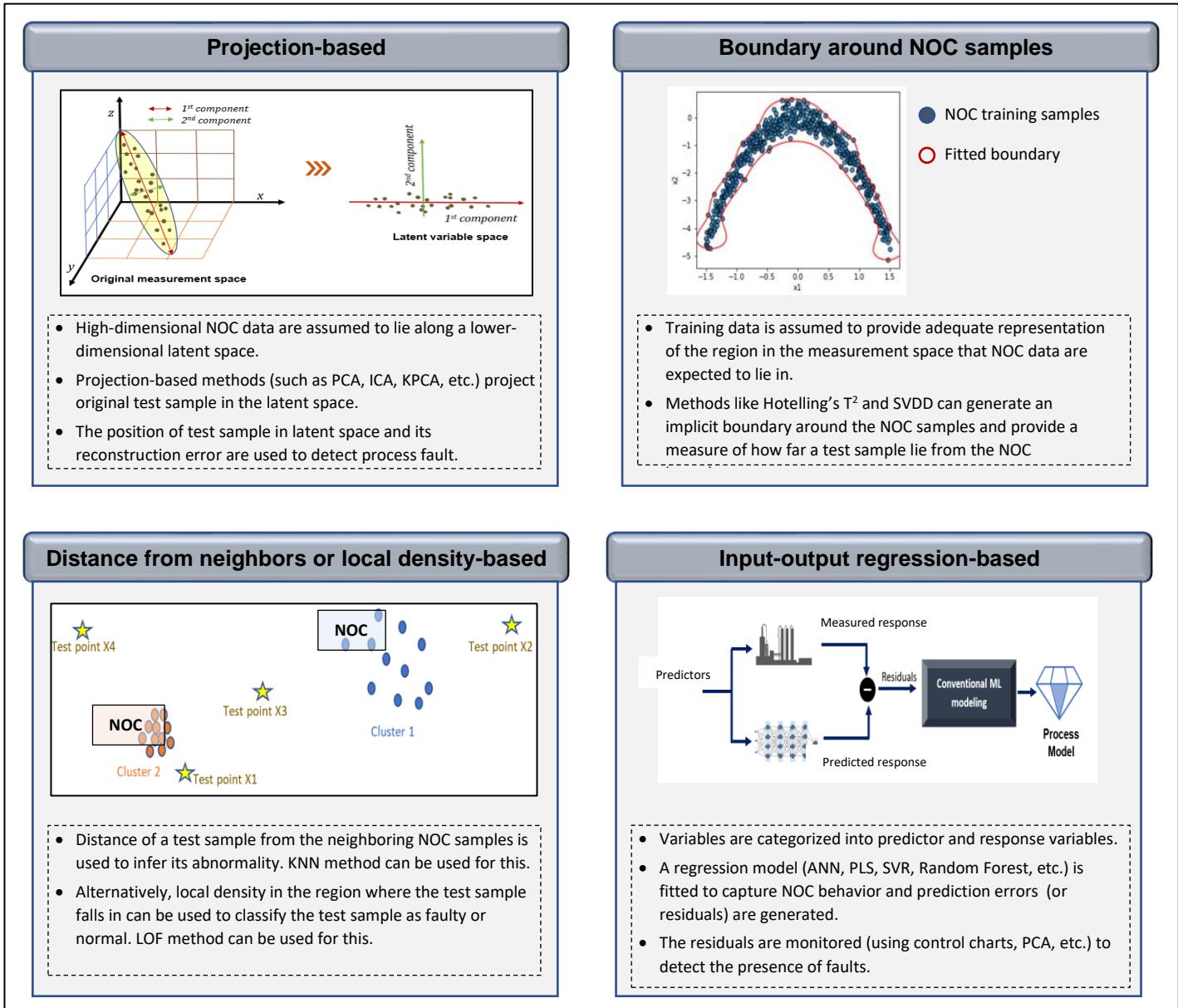


Figure 1.8: Popular fault detection methodologies using only NOC data

Note that the models in Figure 1.7 are applicable to both equipment level monitoring and plantwide monitoring. Let us now move to an overview of how these models are actually developed.

1.4 ML Model Development Workflow

In Figure 1.7, we saw different types of ML models for PM applications. Fortunately, the workflow for model development and deployment remains similar, and is shown in Figure 1.9. As is typical for a ML project, the tasks can be categorized into offline computations and online/real-time computations. In online computations, process data are parsed through the model to provide real-time insights and results. The models are built offline using historical process data. This offline exercise is performed once or repeated at regular intervals for model update. Brief description of the essential steps performed are provided below:

- *Exploratory data analysis:* Exploratory data analysis (EDA) involves preliminary investigation of data to get a ‘feel’ of the process dataset characteristics. The activities may include assessment of the presence of nonlinear relationships among process variables, non-Gaussian distribution, etc. Inferences made during EDA help make the right model selection. EDA is covered in detail in Chapter 3.

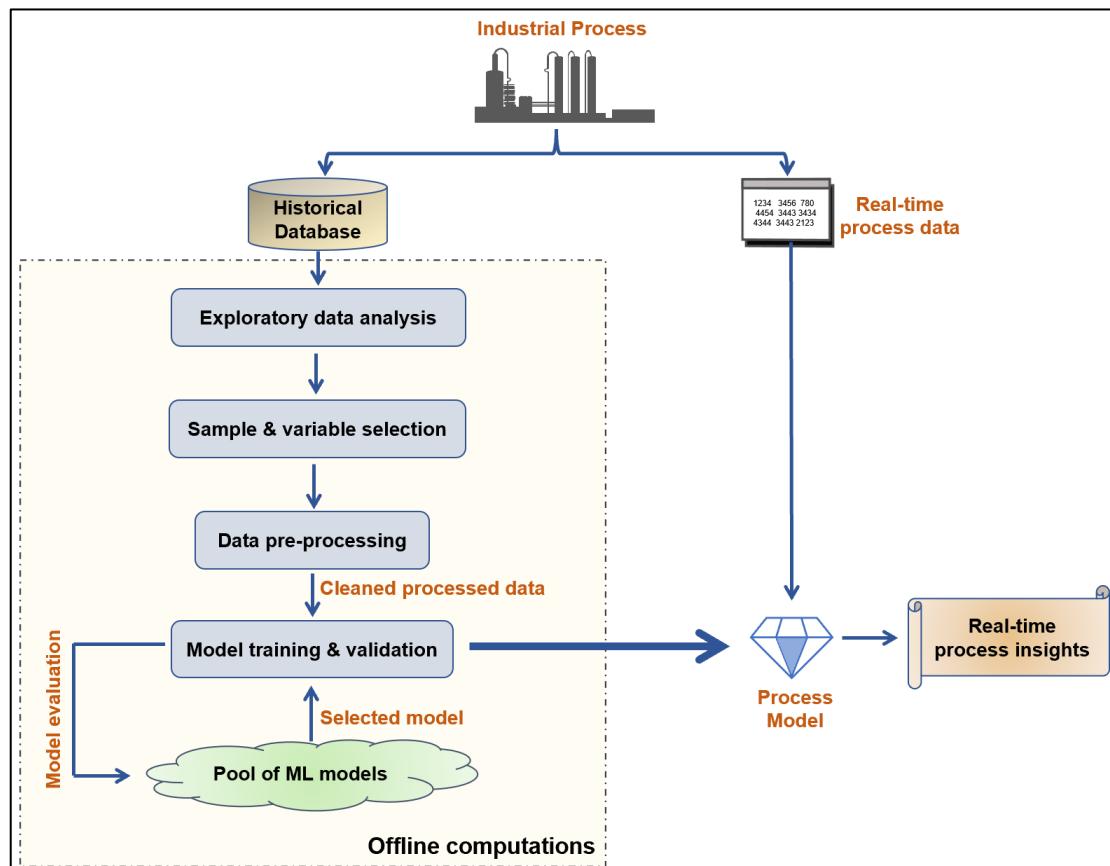


Figure 1.9: Steps involved in a typical ML model development for process monitoring

- **Sample and variable selection:** One does not simply dump all the available historical data and sensor measurements into a model training module. If a model is being built to identify the normal process behavior, then care must be taken to include only samples from fault-free operations in the model training dataset. Furthermore, if your model does not handle dynamics then data from periods of process transitions should be excluded.

Variable selection warrants judicious consideration as well. Inclusion of unnecessary variables makes data noisier and reduces effectiveness of fault detection model. A generic guidance is to include only those variables that can assist in early fault detection; a variable that does not show any change in behavior under the influence of process faults of interest should be excluded.

- **Data pre-processing:** “Garbage in, garbage out” is an age-old principle in computer simulations. The same holds for ML model training for PHM. Your model will be practically useless if training data is not ‘clean’. Your process monitoring model won’t be able to detect process abnormalities accurately if it has been trained with outlier-infested training data. Data pre-processing includes, amongst others, identification and removal of outliers, noise reduction, transformation of variables, and extraction of features. The overall objective of this step is to increase the ‘information content’ of your training dataset so that the PM model’s ability to distinguish between normal and faulty operations is bolstered. Several aspects of data pre-processing are dealt with in Chapter 4.
- **Model training and validation:** Model training imply estimating the parameters of the chosen ML model, for example, the neuron weights in an ANN model. Model validation is employed for finding optimal values of model hyperparameters, for example, the number of neurons in the ANN model. At the end of this step, the coveted process model is obtained.

Additional activities related to computation of health indicator and subsequent RUL estimation involved in fault prognosis are covered in Part 7 of the book which deals specifically with prognostic techniques for predictive maintenance applications.

1.5 ML-based Plant Health Management Solution Deployment

After you have developed a satisfactory PHM model, the real test of your solution lies in how well it is received by the end-users. The end-users could be reliability personnel/engineers at the local plant sites or the central team of experts remotely supervising the plants. Figure 1.10 below shows a (simplified) common architecture for bringing your tool's results to these end-users. As shown, the ML model could be setup to run on local PCs at every site or a central server machine/cloud resource⁵. Plant operators may access the tool's results on the local control-room screens or via web browsers in case of centralized deployment. The web user interface could be either built using third-party visualization software (Tableau, Sisense, Power BI, etc.) or completely custom-built using front-end web frameworks like bootstrap.

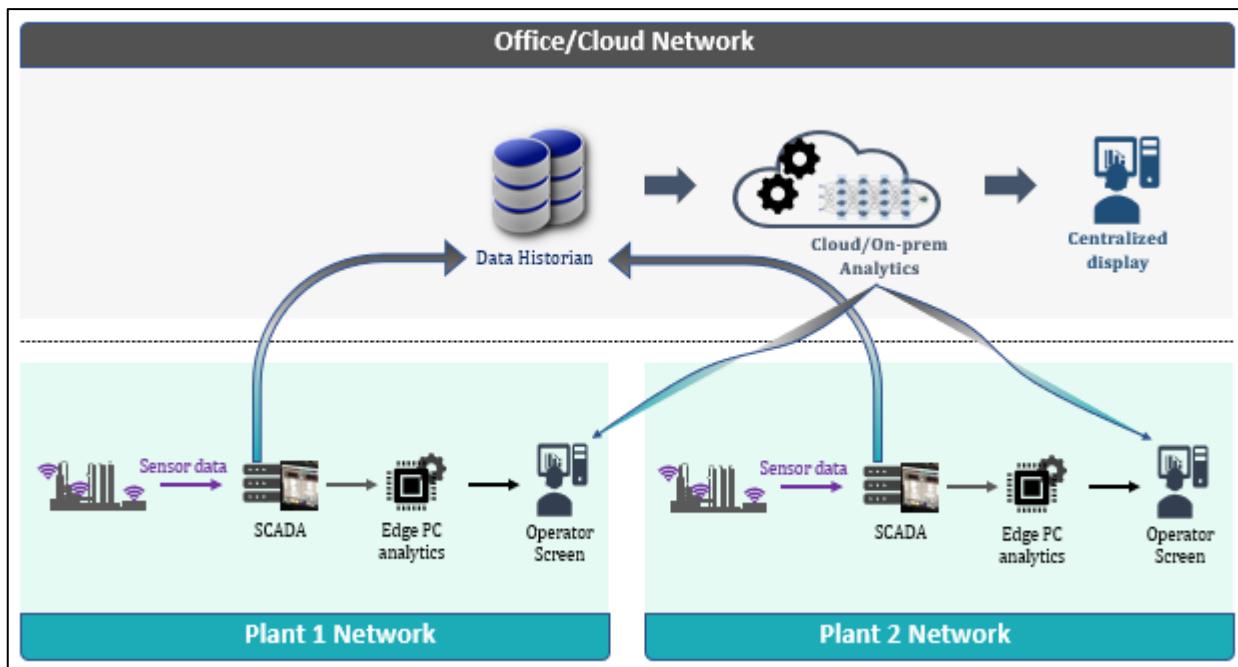


Figure 1.10: ML solution deployment

That is all it takes to deploy a ML-based PHM solution in a production environment. This concludes our quick attempt to impress upon you the importance of process monitoring and predictive maintenance in process industry. Hopefully, you also now have a good idea of what resources you have to achieve your PM/PdM goals and what is takes to build a PM/PdM solution.

⁵ There exists a specialized branch of machine learning engineering, called MLOps (short for machine learning operations) that deals with reliable and scalable deployment of ML models in production.

Summary

In this chapter, we looked at the importance of plant health management for increasing process safety, reducing downtime costs, and increasing equipment life. We understood the meaning of process faults and abnormalities. We looked at the different stages of plant health management, familiarized ourselves with the factors that influence model selection, and looked at the different models available at our disposal to achieve the PHM goals. We also briefly looked at the generic workflow for process monitoring model development and understood how PM/PdM solutions are deployed in modern industrial settings. In the next chapter, we will take the first step and learn about the environment we will use to execute our Python scripts containing ML code for PHM.

Chapter 2

The Scripting Environment

In the previous chapter, we studied the various aspects of machine learning-based process monitoring and predictive maintenance. In this chapter, we will quickly familiarize ourselves with the Python language and the scripting environment that we will use to write ML codes, execute them, and see results. This chapter won't make you an expert in Python but will give you enough understanding of the language to get you started and help understand the several in-chapter code implementations in the upcoming chapters. If you already know the basics of Python, have a preferred code editor, and know the general structure of a typical ML script, then you can skip to Chapter 3.

The ease of using and learning Python, along with the availability of a plethora of open-access useful packages developed by the user-community over the years, has led to immense popularity of Python. In recent years, development of specialized libraries for machine and deep learning has made Python the default language of choice among ML community. These advancements have greatly lowered the entry barrier into the world of machine learning for new users.

With this chapter, you are putting your first foot into the ML world. Specifically, the following topics are covered

- Introduction to Python language
- Introduction to Spyder and Jupyter, two popular code editors
- Overview of Python data structures and scientific computing libraries

2.1 Introduction to Python

In simple terms, Python is a high-level general-purpose computer programming language that can be used, amongst others, for application development and scientific computing. If you have used other computer languages like Visual Basic, C#, C++, Java, Javascript, then you would understand the fact that Python is an interpreted and dynamic language. If not, then think of Python as just another name in the list of computer programming languages. What is more important is that Python offers several features that sets it apart from the rest of the pack making it the most preferred language for machine learning. Figure 2.1 lists some of these features. Python provides all the tools to conveniently carry out all steps of an ML-based PM/PdM project, namely, data collection, data pre-processing, data exploration, ML modeling, visualization, and solution deployment to end-users. In addition, several freely available tools make writing Python code very easy⁶.



Figure 2.1: Features contributing to Python language's popularity

Installing Python

One can download official and the latest version of Python from the python.com website. However, the most convenient way to install and use Python is to install Anaconda (www.anaconda.com) which is an open-source distribution of Python. Along with the core Python, Anaconda installs a lot of other useful packages. Anaconda comes with a GUI called Anaconda Navigator (Figure 2.2) from where you can launch several other tools.

⁶ Most of the content of this chapter is like that of Chapter 2 of the book ‘Machine Learning in Python for Process Systems Engineering’ and have been re-produced with appropriate changes to maintain standalone nature of this book.

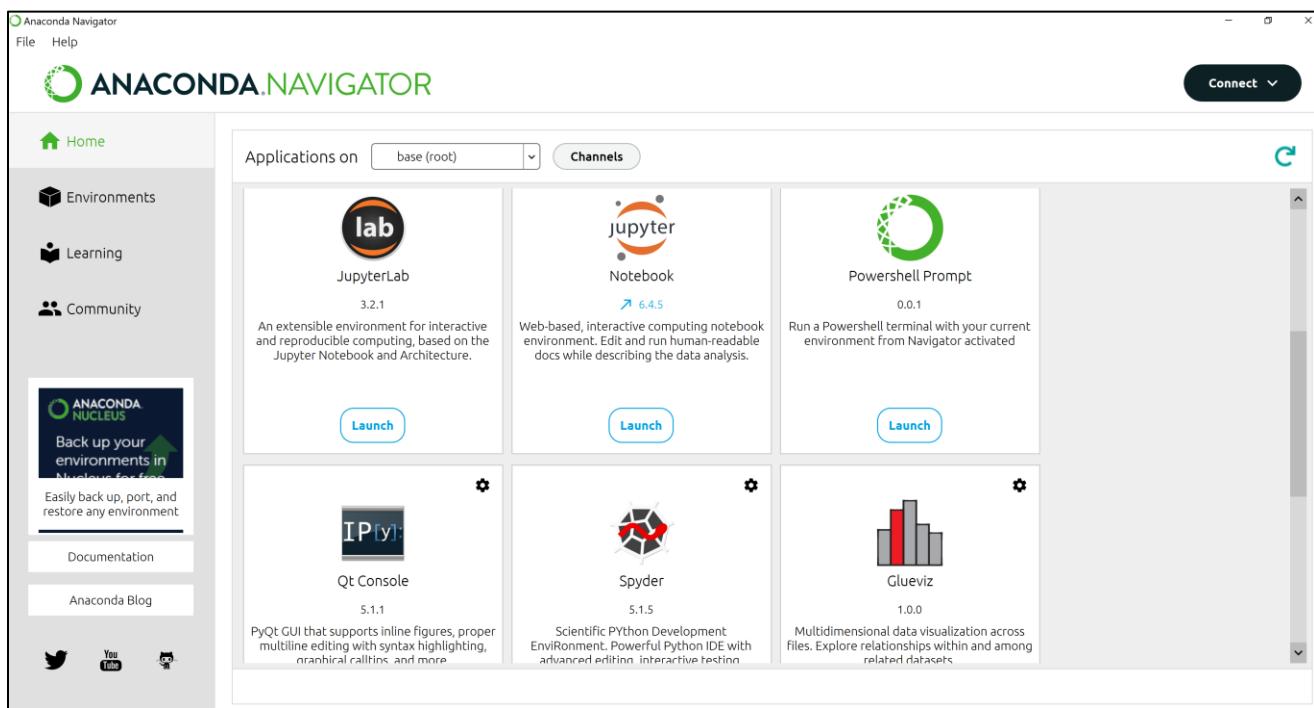


Figure 2.2: Anaconda Navigator GUI

Running/Executing Python code

Once Anaconda is installed, you can write your code in any text editor (like Notepad), save it in .py format, and then run via Anaconda command terminal. However, a convenient alternative is to use IDEs (integrated development environments). IDEs not just allow creating .py scripts, but also execute them on the fly, i.e., we can make code edits and see the results immediately, all within an integrated environment. Among the several available Python IDEs (PyCharm, VS Code, Spyder, etc.), we found Spyder to be the most convenient and functionality-rich, and therefore is discussed in the next section.

Jupyter Notebooks are another very popular way of writing and executing Python code. These notebooks allow combining code, execution results, explanatory text, multimedia resources in a single document. As you can imagine, this makes saving and sharing complete data analysis very easy.

In the next section, we will provide you with enough familiarity on Spyder and Jupyter so that you can start using them.

2.2 Introduction to Spyder and Jupyter

Figure 2.3 shows the interface⁷ (and its different components) that comes up when you launch Spyder. These are the 3 main components:

- *Editor*: You can type and save your code here. Clicking ➤ button executes the code in the active editor tab.
- *Console*: Script execution results are shown here. It can also be used for executing Python commands and interact with variables in the workspace.
- *Variable explorer*: All the variables generated by running editor scripts or console are shown here and can be interactively browsed.

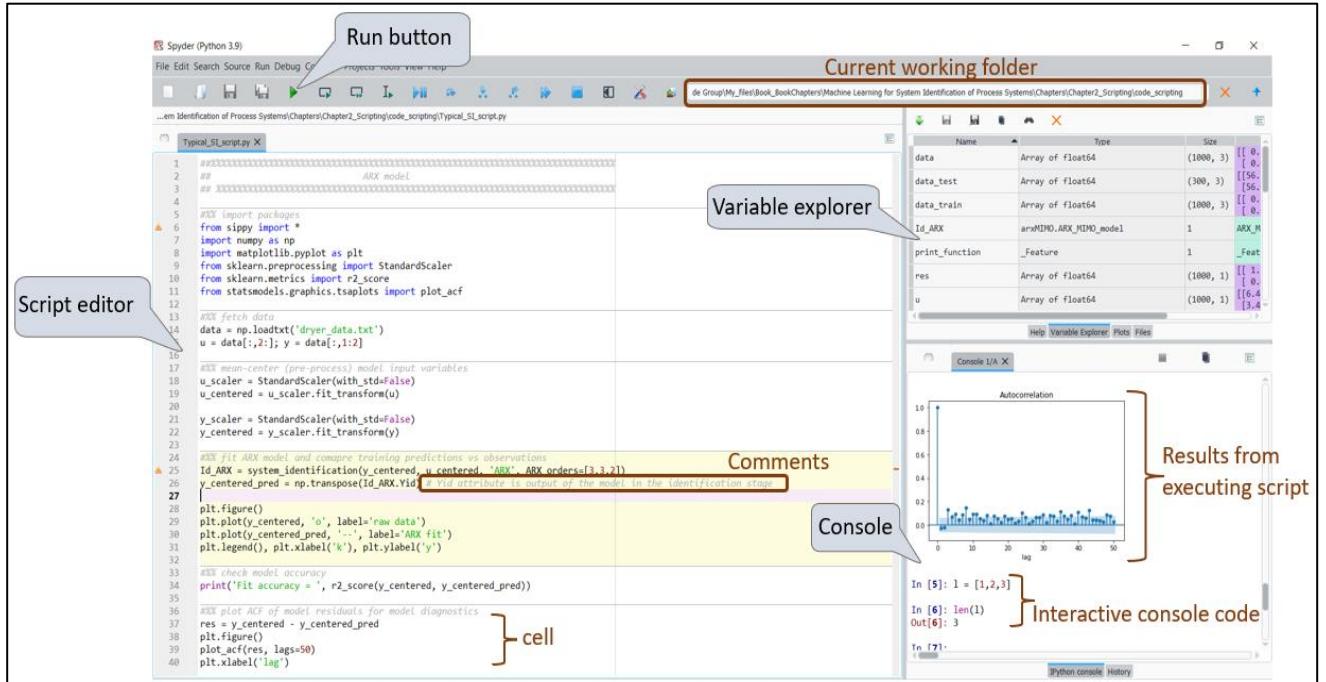


Figure 2.3: Spyder interface

Like any IDE, Spyder offers several features. You can divide your script into cells and execute only any selected cell if you choose to (by pressing Ctrl + Enter buttons). Intellisense allows you to autocomplete your code by pressing Tab key. Extensive debugging functionalities make troubleshooting easier. These are only some of the features available in Spyder. You are encouraged to explore the different options (such as pausing and canceling script execution, clearing out variable workspace, etc.) on the Spyder GUI.

⁷ If you have used MATLAB, you will find the interface very familiar.

With Spyder, you have to run your script again to see execution results if you close and reopen your script. In contrast to this, consider the Jupyter interface in Figure 2.4. Note that the Jupyter interface opens in a browser. We can save the shown code, the execution outputs, and explanatory text/figures as a (.ipnb) file and have them remain intact when we reopen the file in Jupyter Notebook.

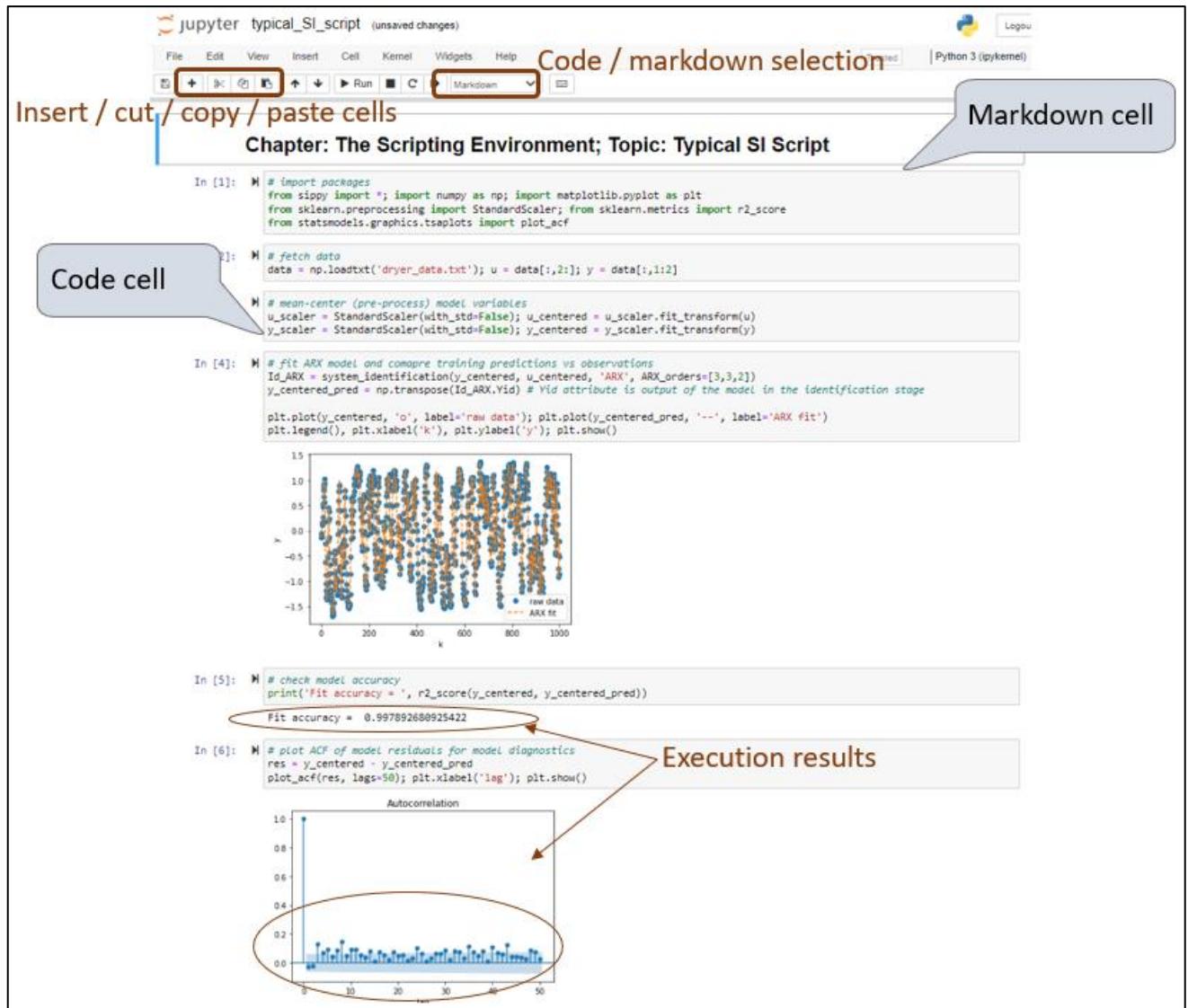


Figure 2.4: Jupyter interface

You can designate any input cell as a code or markdown (formatted explanatory text). You can press Shift + Enter keys to execute any active cell. All the input cells can be executed via the *Cell* menu.

This completes our quick overview of Spyder and Jupyter interface. You can choose either of them for working through the codes in the rest of the book.

2.3 Python Language: Basics

In the current and next sections, we will see several simple examples of manipulating data using Python and scientific packages. While these simple operations may seem unremarkable (and boring) in the absence of any larger context, they form the building blocks of more complex scripts presented later in the book. Therefore, it will be worthwhile to give these at least a quick glance.

Note that you will find '#' used a lot in these examples; these hash marks are used to insert explanatory comments in code. Python ignores (does not execute) anything written after # on a line.

Basic data types

In Python, you work with 4 types of data types

```
# 4 data types: int, float, str, bool
i = 2 # integer; type(i) = int
f = 1.2 # floating-point number; type(f) = float
s = 'two' # string; type(s) = str
b = True # boolean; type(b) = bool
```

With the variables defined, you can perform basic operations

```
# print function prints/displays the specified message
print(i+2) # displays 4
print(f*2) # displays 2.4
```

Lists, tuples as ordered sequences

Related data (not necessarily of the same data type) can be arranged together as a sequence in a list as shown below

```
# different ways of creating lists
list1 = [2,4,6]
list2 = ['air',3,1,5]
list3 = list(range(4)) # equals [0,1,2,3]; range function returns a sequence of numbers starting
# from 0 (default) with increments of 1 (default)
list3.append(8) # returns [0,1,2,3,8]; append function adds new items to existing list
list4 = list1 + list2 # equals [2,4,6,'air',3,1,5]
list5 = [list2, list3] # nested list [[['air', 3, 1, 5], [0, 1, 2, 3, 8]]]
```

Tuples are another sequence construct like lists, with a difference that their items and sizes cannot be changed. Since tuples are immutable/unchangeable, they are more memory efficient.

```
# creating tuples
tuple1 = (0,1,'two')
tuple2 = (list1, list2) # equals ([2, 4, 6, 8], ['air', 3, 1, 5])
```

A couple of examples below illustrate list comprehension which is a very useful way of creating new lists from other sequences

```
# generate powers of individual items in list3
newList1 = [item**2 for item in list3] # equals [0,1,4,9,64]

# nested list comprehension
newList2 = [item2**2 for item2 in [item**2 for item in list3]] # equals [0,1,16,81,4096]
```

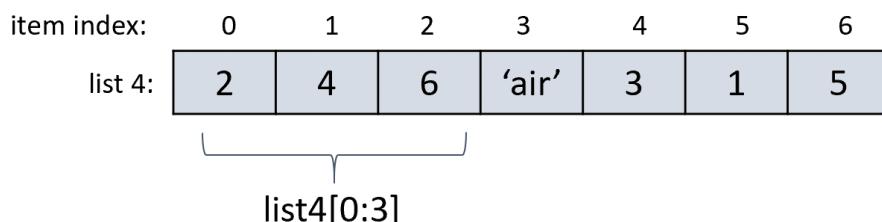
Indexing and slicing sequences

Individual elements in a list can be accessed and modified as follows

```
# working with single item using positive or negative indexes
print(list1[0]) # displays 2, the 1st item in list1
list2[1] = 1 # list2 becomes ['air',1,1,5]
print(list2[-2]) # displays 1, the 2nd last element in list2
```

Note that Python indexing starts from zero. Very often, we need to work with multiple items of the list. This can be accomplished easily as shown below.

```
# accessing multiple items through slicing
# Syntax: givenList[start,stop,step]; if unspecified, start=0, stop=list length, step=1
print(list4[0:3]) # displays [2,4,6], the 1st, 2nd, 3rd items; note that index 3 item is excluded
```



```
print(list4[:3]) # same as above
print(list4[4:len(list4)]) # displays [3,1,5]; len() function returns the number of items in list
```

```

print(list4[4:]) # same as above
print(list4[::-3]) # displays [2, 'air', 5]
print(list4[::-1]) # displays list 4 backwards [5, 1, 3, 'air', 6, 4, 2]
list4[2:4] = [0,0,0] # list 4 becomes [2, 4, 0, 0, 0, 3, 1, 5]

```

Execution control statements

These statements allow you to control the execution sequence of code. You can choose to execute any specific parts of the script selectively or multiple times. Let's see how you can accomplish these

Conditional execution

```

# selectively execute code based on condition
if list1[0] > 0:
    list1[0] = 'positive'
else:
    list1[0] = 'negative'

# list1 becomes ['positive', 4, 6]

```

Loop execution

```

# compute sum of squares of numbers in list3
sum_of_squares = 0
for i in range(len(list3)):
    sum_of_squares += list3[i]**2

print(sum_of_squares) # displays 78

```

Custom functions

Previously we used Python's built-in functions (`len()`, `append()`) to carry out operations pre-defined for these functions. Python allows defining our own custom functions as well. The advantage of custom functions is that we can define a set of instructions once and then re-use them multiple times in our script and project.

For illustration, let's define a function to compute the sum of squares of items in a list.

```

# define function instructions
def sumSquares(givenList):
    sum_of_squares = 0
    for i in range(len(givenList)):
        sum_of_squares += givenList[i]**2

    return sum_of_squares

# call/re-use the custom function multiple times
print(sumSquares(list3)) # displays 78

```



You might have noticed in our custom function code above that we used different indentations (number of whitespaces at beginning of code lines) to separate the ‘for loop’ code from the rest of the function code. This practice is actually enforced by Python and will result in errors or bugs if not followed. While other popular languages like C++, C# use braces {} to demarcate a code block (body of a function, loop, if statement, etc.), Python uses indentation. You can choose the amount of indentation, but it must be consistent within a code block.

This concludes our extremely selective coverage of Python basics. However, this should be sufficient to enable you to understand the codes in the subsequent chapters. Let’s continue now to learn about some specialized scientific packages.

2.4 Scientific Computing Packages: Basics

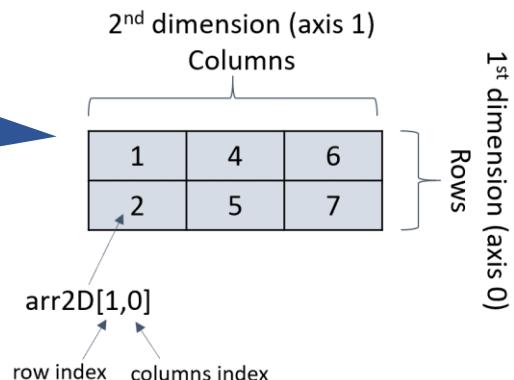
While the core Python data-structures are quite handy, they are not very convenient for the advanced data manipulations we require for machine learning tasks. Fortunately, specialized packages like NumPy, SciPy, Pandas exist which provide convenient multidimensional tabular data structures suited for scientific computing. Let’s quickly make ourselves familiar with these packages.

NumPy

In NumPy, ndarrays are the basic data structures which put data in a grid of values. Illustrations below shows how 1D and 2D arrays can be created and their items accessed

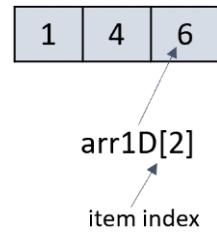
```
# import numpy package & create a 2D array
import numpy as np
arr2D = np.array([[1,4,6],[2,5,7]])

# getting information about arr2D
print(arr2D.size) # returns 6, the no. of items
print(arr2D.ndim) # returns 2, the no. of dimensions
print(arr2D.shape) # returns tuple(2,3) corresponding
                  to 2 rows & 3 columns
```



```
# create a 1D array
arr1D = np.array([1,4,6])
```

getting information about arr1D
print(arr1D.size) # returns 3, the no. of items
print(arr1D.ndim) # returns 1, the no. of dimensions
print(arr1D.shape) # returns tuple (3,) corresponding
to 3 items



Note that the concept of rows and columns do not apply to a 1D array. Also, you would have noticed that we imported the NumPy package before using it in our script ('np' is just a short alias). Importing a package makes available all its functions and sub-packages for use in our script.

Creating NumPy arrays

Previously, we saw how to convert a list to a NumPy array. There are other ways to create NumPy arrays as well. Some examples are shown below

```
# creating sequence of numbers
arr1 = np.arange(3, 6) # same as Python range function; results in array([3,4,5])
arr2 = np.arange(3, 9, 2) # the 3rd argument defines the step size; results in array([3,5,7])
arr3 = np.linspace(1,7,3) # creates evenly spaced 3 values from 1 to 7; results in
array([1,4,7])

# creating special arrays
arr4 = np.ones((2,1)) # array of shape (2,1) with all items as 1
arr5 = np.zeros((2,2)) # all items as zero; often used as placeholder array at beginning of script
arr6 = np.eye(2) # diagonal items as 1

# adding axis to existing arrays (e.g., converting 1D array to 2D array)
print(arr1[:, np.newaxis])
>>> [[3]
     [4]
     [5]]

arr7 = arr1[:, None] # same as above
```

```
# combining / stacking arrays
print(np.hstack((arr1, arr2))) # horizontally stacks passed arrays
>>> [3 4 5 3 5 7]

print(np.vstack((arr1, arr2))) # vertically stacks passed arrays
>>> [[3 4 5]
     [3 5 7]]

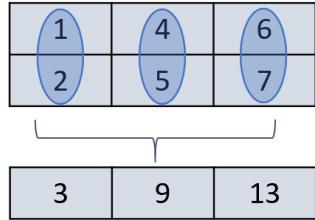
print(np.hstack((arr5,arr4))) # array 4 added as a column into arr5
>>> [[0. 0. 1.]
     [0. 0. 1.]]

print(np.vstack((arr5,arr6))) # rows of array 6 added onto arr5
>>> [[0. 0.]
     [0. 0.]
     [1. 0.]
     [0. 1.]]
```

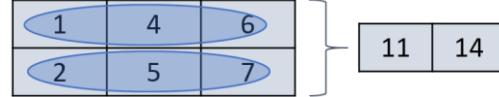
Basic Numpy functions

NumPy provides several useful functions like mean, sum, sort, etc., to manipulate and analyze NumPy arrays. You can specify the dimension (axis) along which data needs to be analyzed. Consider the sum function for example

along the row sum
arr2D.sum(axis=0) # returns 1D array
with 3 items



along the column sum
arr2D.sum(axis=1) # returns 1D array
with 2 items



Executing arr2D.sum() returns the scalar sum over the whole array, i.e., 25.

Indexing and slicing arrays

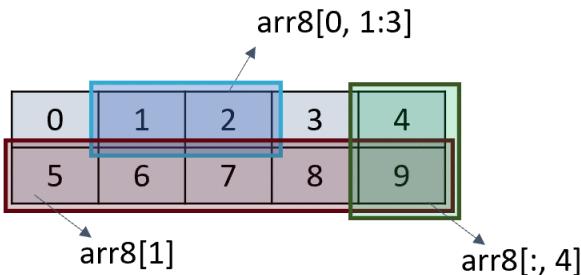
Accessing individual items and slicing NumPy arrays work the same as that for Python lists

```
# accessing individual items
print(arr2D[1,2]) # returns 7

# slicing
arr8 = np.arange(10).reshape((2,5)) # rearrange the 1D array into shape (2,5)
print((arr8[0:1,1:3]))
>>> [[1 2]]

print((arr8[0:1:3])) # note that a 1D array is returned here instead of the 2D array above
>>> [1 2]

# accessing entire row or column
print(arr8[1]) # returns 2nd row as array([5,6,7,8,9]); same as arr8[1,:]
print(arr8[:, 4]) # returns items of 5th column as a 1D array
>>> [4 9]
```



An important thing to note about NumPy array slices is that any change made on sliced view modifies the original array as well! See the following example

```
# extract a subarray from arr8 and modify it
arr8_sub = arr8[:, :2] # columns 0 and 1 from all rows
arr8_sub[1,1] = 1000
print(arr8) # arr8 gets modified as well!!
>>> [[ 0  1  2  3  4]
     [ 5 1000  7  8  9]]
```

This feature becomes quite handy when we need to work on only a small part of a large array/dataset. We can simply work on a leaner view instead of carrying around the large dataset. However, situation may arise where we need to actually work on a separate copy of subarray without worrying about modifying the original array. This can be accomplished via the `copy` method.

```
# use copy method for a separate copy
arr8 = np.arange(10).reshape((2,5))
arr8_sub2 = arr8[:, :2].copy()
arr8_sub2[1, 1] = 100 # arr8 won't be affected here
```

Fancy indexing is another way of obtaining a copy instead of a view of the array being indexed. Fancy indexing simply entails using integer or boolean array/list to access array items. Examples below clarify this concept

```
# combination of simple and fancy indexing
arr8_sub3 = arr8[:, [0, 1]] # note how columns are indexed via a list
arr8_sub3[1, 1] = 100 # arr8_sub3 becomes same as arr8_sub2 but arr8 is not modified here

# use boolean mask to select subarray
arr8_sub4 = arr8[arr8 > 5] # returns array([6,7,8,9]), i.e., all values > 5
arr8_sub4[0] = 0 # again, arr8 is not affected
```

Vectorized operations



Suppose you need to perform element-wise summation of two 1D arrays. One approach is to access items at each index at a time in a loop and sum them. Another approach is to sum up items at multiple indexes at once. The later approach is called vectorized operation and can lead to significant boost in computational time for large datasets and complex operations.

```
# vectorized operations
vec1 = np.array([1,2,3,4])
vec2 = np.array([5,6,7,8])
vec_sum = vec1 + vec2 # returns array([6,8,10,12]); no need to loop through index 0 to 3

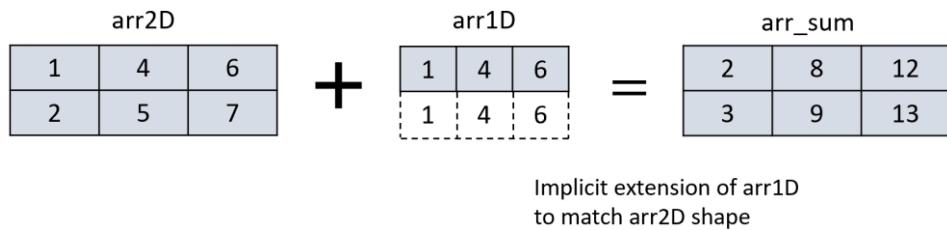
# slightly more complex operation (computing distance between vectors)
vec_distance = np.sqrt(np.sum((vec1 - vec2)**2)) # vec_distance = 8.0
```

Broadcasting

Consider the following summation of arr2D and arr1D arrays

```
# item-wise addition of arr2D and arr1D
arr_sum = arr2D + arr1D
```

NumPy allows item-wise operation between arrays of different dimensions, thanks to ‘broadcasting’ where smaller array is implicitly extended to be compatible with the larger array, as shown in the illustration below. As you can imagine, this is a very convenient feature and for more details on broadcasting rules along with different example scenarios, you are encouraged to see the official documentation⁸.



Pandas

Pandas is another very powerful scientific package. It is built on top of NumPy and offers several data structures and functionalities which make (tabular) data analysis and pre-processing very convenient. Some noteworthy features include label-based slicing/indexing, (SQL-like) data grouping/aggregation, data merging/joining, and time-series functionalities. Series and dataframe are the 1D and 2D array like structures, respectively, provided by Pandas

```
# Series (1D structure)
import pandas as pd

data = [10,8,6]
s = pd.Series(data)
print(s)
>>>
    0    10
    1     8
    2     6

```

0 10
1 8
2 6

default row index

```
# Dataframe (2D structure)
data = [[1,10],[1,8],[1,6]]
df = pd.DataFrame(data, columns=['id', 'value'])
print(df)
>>>
```

	id	value
0	1	10
1	1	8
2	1	6

labeled columns (becomes 0 and 1 if no labels given)

```
# dataframe from series
s2 = pd.Series([1,1,1])
df = pd.DataFrame({'id':s2, 'value':s2}) # same as above
```

Note that `s.values` and `df.values` convert the series and dataframe into corresponding NumPy arrays.

⁸ numpy.org/doc/stable/user/basics.broadcasting.html

Data access

Pandas allows accessing rows and columns of a dataframe using labels as well as integer locations. You will find this feature pretty convenient.

```
# column(s) selection
print(df['id']) # returns column 'id' as a series
print(df.id) # same as above
print(df[['id']]) # returns specified columns in the list as a dataframe
>>> id
0 1
1 1
2 1

# row selection
df.index = [100, 101, 102] # changing row indices from [0,1,2] to [100,101,102] for illustration
print(df)
>>> id value
100 1 10
101 1 8
102 1 6

print(df.loc[101]) # returns 2nd row as a series; can provide a list for multiple rows selection
print(df.iloc[1]) # integer location-based selection; same result as above

# individual item selection
print(df.loc[101, 'value']) # returns 8
print(df.iloc[1, 1]) # same as above
```

Data aggregation

As alluded to earlier, Pandas facilitates quick analysis of data. Check out one quick example below for group-based mean aggregation

```
# create another dataframe using df
df2 = df.copy()
df2.id = 2 # make all items in column 'id' as 2
df2.value *= 4 # multiply all items in column 'value' by 4
print(df2)
>>> id value
100 2 40
101 2 32
```

```
102 2 24
```

```
# combine df and df2
df3 = df.append(df2) # a new object is retuned unlike Python's append function
print(df3)
>>> id value
100 1 10
101 1 8
102 1 6
100 2 40
101 2 32
102 2 24

# id-based mean values computation
print(df3.groupby('id').mean()) # returns a dataframe
>>> value
id
1    8.0
2   32.0
```

File I/O

Conveniently reading data from external sources and files is one of the strong forte of Pandas. Below are a couple of illustrative examples.

```
# reading from excel and csv files
dataset1 = pd.read_excel('filename.xlsx') # several parameter options are available to customize
                                            what data is read
dataset2 = pd.read_csv('filename.xlsx')
```

This completes our very brief look at Python, NumPy, and Pandas. If you are new to Python (or coding), this may have been overwhelming. Don't worry. Now that you are atleast aware of the different data structures and ways of accessing data, you will become more and more comfortable with Python scripting as you work through the in-chapter code examples.

Sklearn

Sklearn is currently the most popular library for machine learning in Python. It provides several modules to conveniently handle different aspects of ML such as data standardization, performance scoring, data splitting, etc. You do not need to implement ML models such as PCA, Random Forests, GMM, etc., from scratch; Sklearn provide ready-to-use implementations of popular ML models. As you will see in the upcoming chapters, ML models can be defined and fitted in a single line of code using Sklearn library.

Summary

In this chapter, we made ourselves familiar with the scripting environment that we will use for writing and executing PHM scripts. We looked at two popular IDEs for Python scripting, the basics of Python language, and learnt how to manipulate data using NumPy and Pandas. In the next chapter, you will learn a few techniques for exploratory data analysis that you can use to get some insights about your data which can help you select your ML model judiciously.

Chapter 3

Exploratory Data Analysis: Getting to Know Your Data Better

Getting to know your enemy is a time-tested strategy for emerging victorious in any battle. For developing a satisfactory process monitoring model, this strategy translates to ‘knowing your process data well’. This task is formally termed as exploratory data analysis (EDA). Most of the machine learning models make some assumptions regarding the distribution (e.g., Gaussian vs uniform distribution) and characteristics (e.g., dynamic vs steady state nature) of the data they operate upon. Therefore, it only serves us well investing some time in EDA so that the consistency between our chosen model’s assumptions and the characteristics of process data at hand can be ascertained. Failure to do so will lead to high rate of false alerts and/or missed/delayed fault detection which will most likely lead to ‘death’ of your monitoring tool due to loss of user confidence!

In this chapter, we will learn how to assess the presence of four important properties in a dataset, viz, nonlinearity, non-Gaussianity, dynamics, and multimodality. We will motivate the study of these properties by understanding their impact on process monitoring performance. We will especially focus on techniques that render themselves convenient for implementation in an automated setting. As is obvious, the concepts learnt in this chapter will help you get better at correct model selection. Specifically, the following topics are covered

- Impact of non-ideal data properties on fault detection performance
- Techniques for assessing nonlinearity, non-Gaussianity, dynamic, and multimodality
- EDA of Tennessee Eastman Process dataset

3.1 Why Exploratory Data Analysis Matters?

Ask any expert process data scientist about some advice to get better at ML model selection and you will very likely get the suggestions to understand your data better. It's true, you cannot over-exaggerate the importance of gathering as many insights about the data as possible before getting your hands dirty. Most of the process monitoring methodologies make assumptions about the data characteristics and therefore, it is imperative to ascertain these characteristics in our process data to ensure selection of appropriate monitoring model. Let's consider the classical PCA model (inarguably the most popular model for monitoring multivariate industrial processes): the ideal dataset is linear, Gaussian distributed, single-clustered, and with no dynamics; Figure 3.1 uses simple datasets to illustrate what the deviations from these ideal characteristics look like.

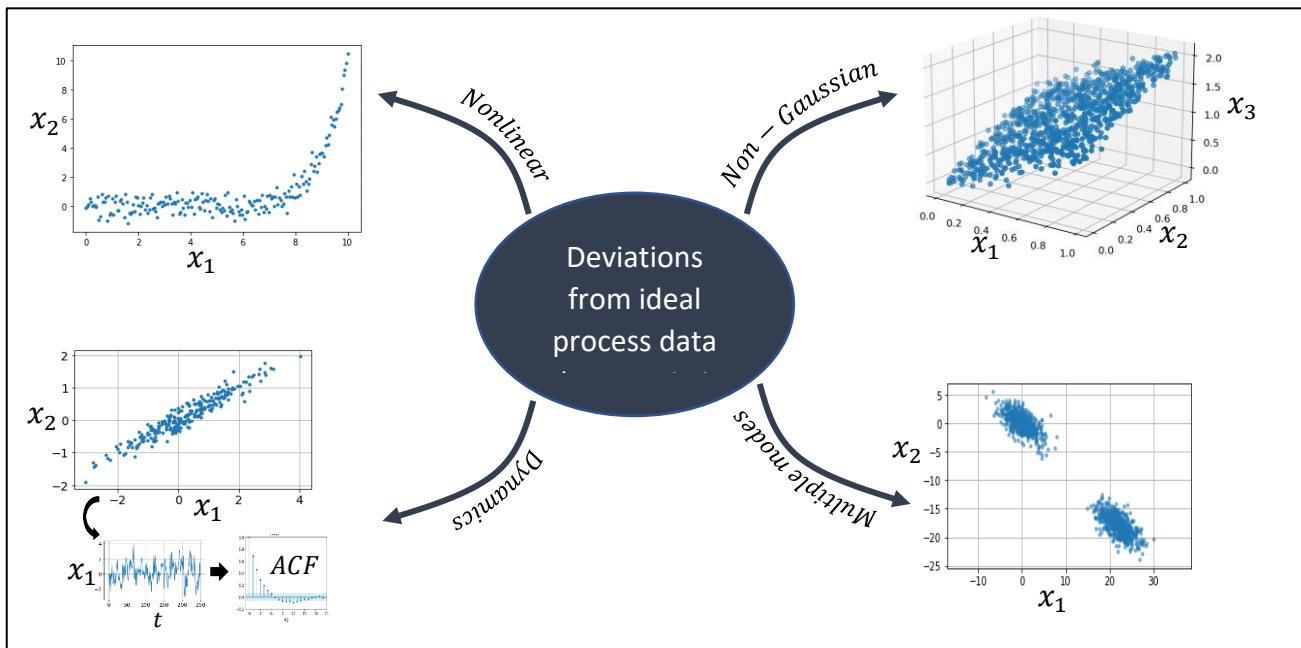
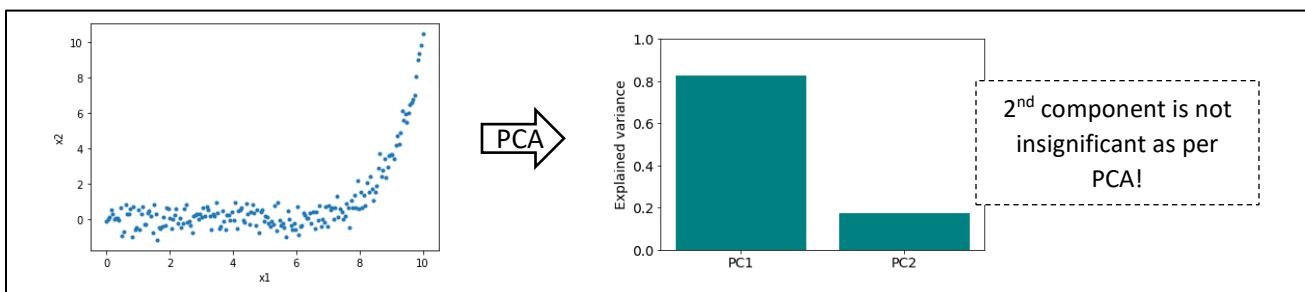


Figure 3.1: Illustration of deviations from ideal process data characteristics

To further motivate the discussions in the rest of the chapter, let's take a quick look at the impacts the non-ideal data characteristics can have on PCA performance.

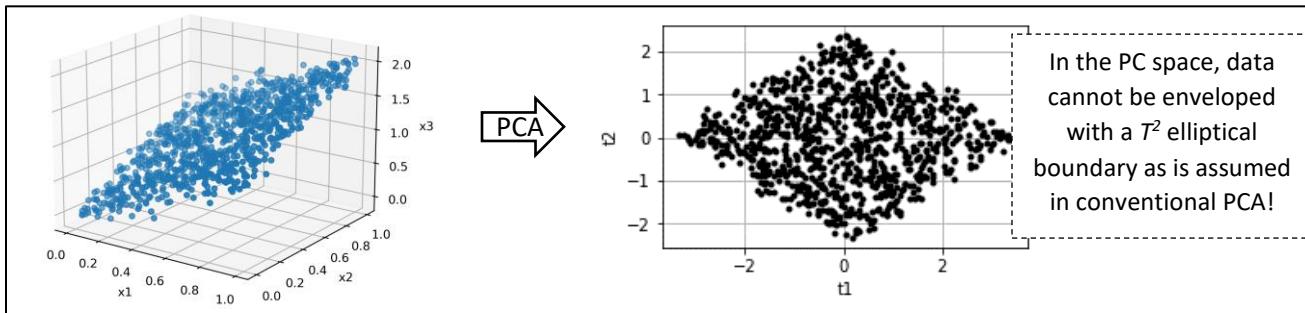
Effect of nonlinearity

In an ideal PCA-compatible dataset, the variables are linearly related which allows the standard PCA method to find the lower-dimensional manifold along which the data is distributed. However, as can be seen below, PCA fails to transform the original 2D dataset to a 1D feature space even though it is apparent that the original data points lie along a curved manifold. This severely limits the ability of standard PCA to detect faulty samples.



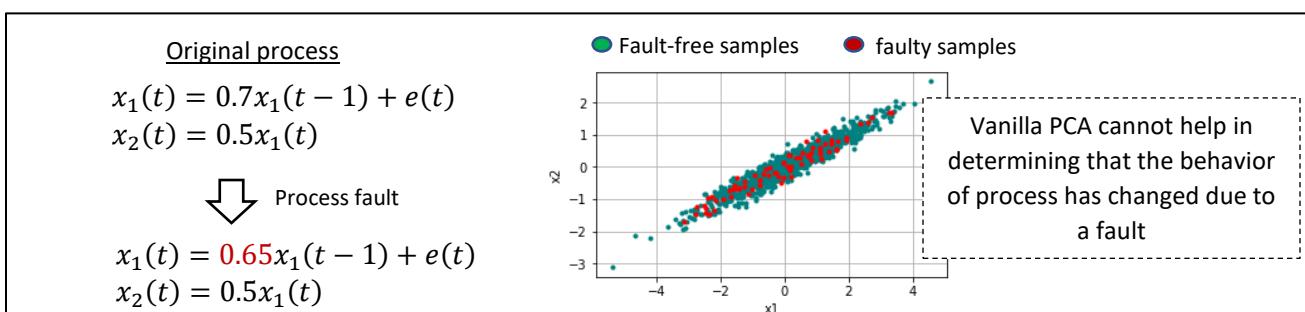
Effect of non-Gaussianity

Not many process modelers pay attention to the Gaussianity of process dataset. The presence of non-Gaussianity can severely impact the performance of many ML models. Illustration below shows why standard PCA-based fault detection⁹ may fail for non-Gaussian data.



Effect of dynamics

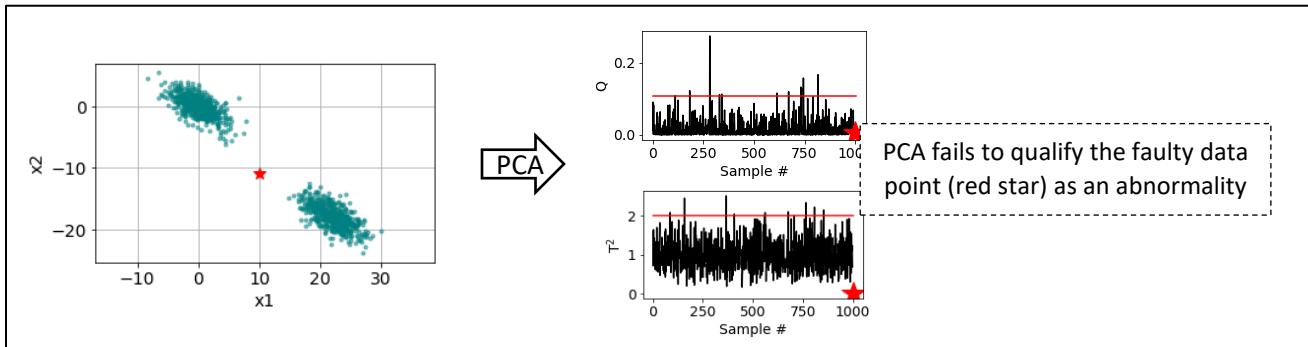
While PCA excels at extracting out static relationships among process variables, it fails to capture any dynamic relationship. For example, consider the following process and the corresponding data distribution in the 2D measurement space.



⁹ Do not worry if you do not understand the PCA jargon. You will become familiar with these terms in Chapter 7.

Effect of multi-clustered distribution

Standard PCA-based fault detection assumes that the data is distributed within a single cluster which enables imposition of a single elliptical boundary around the data in the PC space. However, as the illustration below shows, this assumption can lead to missed alerts if data is multi-clustered.



Don't worry, if the PCA-specific jargon in the above illustrations were not immediately clear to you. You may re-visit this section after we finish the study of PCA technique in Chapter 7. Next, we will take each of the four non-ideal characteristics and learn how to quickly assess its presence in a multivariable dataset.



It is worthwhile to mention two open-source Python libraries that provide modules for EDA of process data. KydLIB¹⁰ ('know your data' library) developed by Melo et al. in 2022¹¹ provides several routines which, in the authors' words, are 'specially designed to work with time series data typically obtained from process system engineering applications'. SPA¹² (smart process analytics) software developed by Sun and Braatz¹³ in 2021 is designed for automated model selection for predictive modeling of manufacturing data.

¹⁰ <https://github.com/afraniomelo/KydLIB>

¹¹ Melo et al., Open benchmarks for assessment of process monitoring and fault diagnosis techniques: A review and critical analysis. *Computers and Chemical Engineering*, 2022.

¹² <https://github.com/vickysun5/SmartProcessAnalytics>

¹³ Sun and Braatz, Smart process analytics for predictive modeling. *Computers and Chemical Engineering*, 2021.

3.2 Nonlinearity Assessment Techniques

If any two variables in a multivariable dataset are nonlinearly related, then the dataset is said to possess nonlinear characteristics. Therefore, one may look at scatter plots between each pair of variables to assess nonlinearity, but this, expectedly, becomes cumbersome for high-dimensional dataset. One common recourse is to use Pearson correlation coefficient (ρ) which quantifies the linear dependency between two variables. As is commonly known, $\rho \in [-1, 1]$ where values of -1 and 1 signify perfect linear correlation. A drawback, however, is that $\rho = 0$ only indicates no linear relationship and the variables could still be nonlinearly related. Figure 3.2 shows some illustrative scenarios.

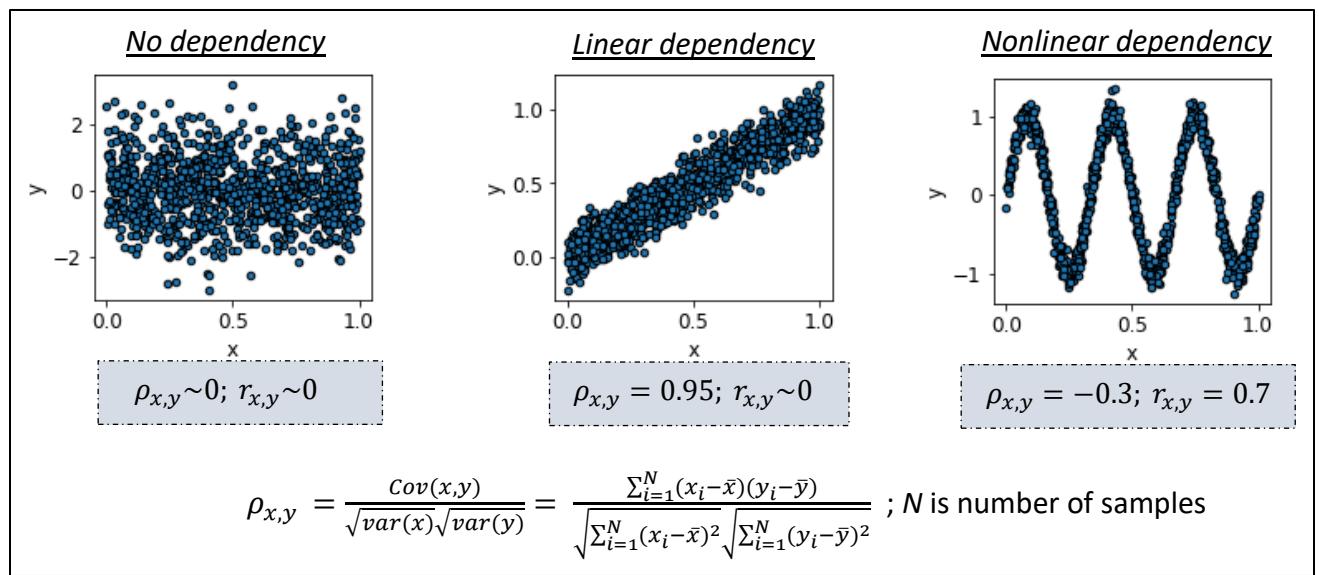


Figure 3.2: Linear ($\rho_{x,y}$) and nonlinear ($r_{x,y}$) correlation coefficients [$r_{x,y}$ is explained below]

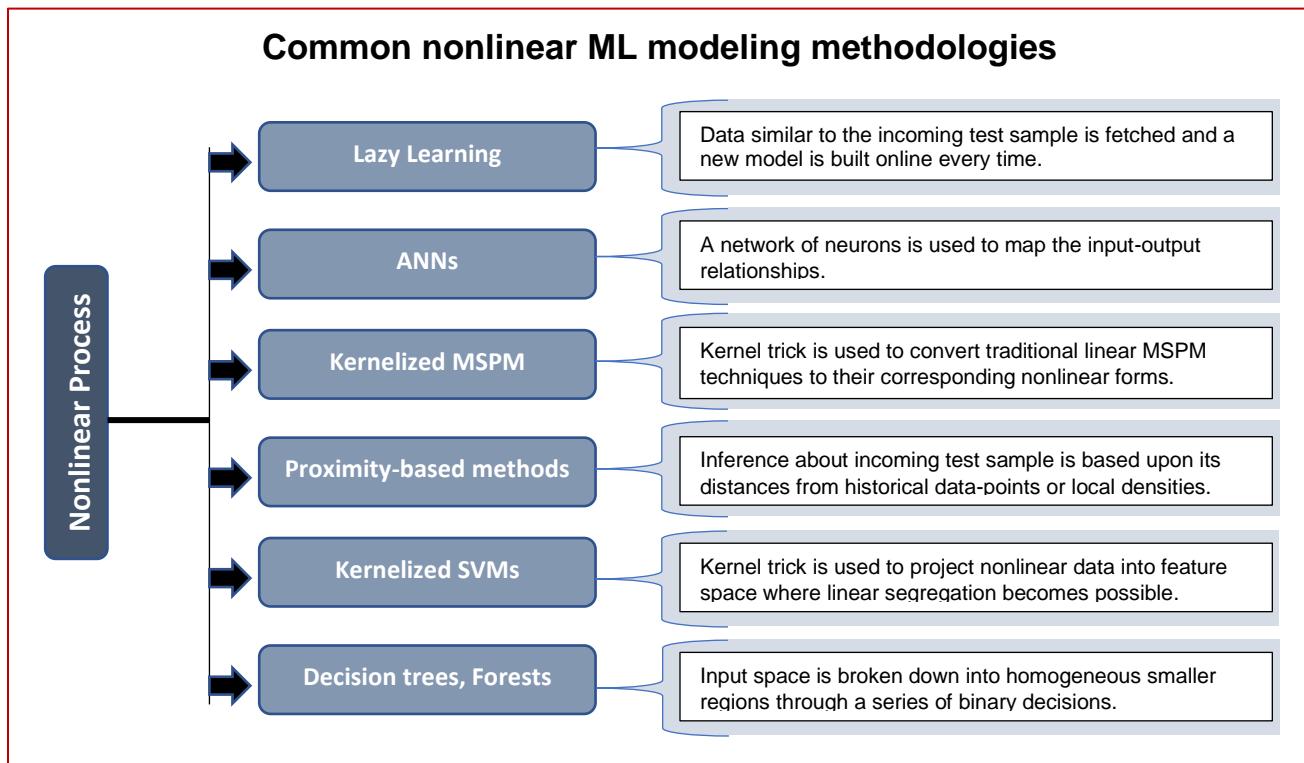
A popular metric to quantify generic dependency (both linear and nonlinear) is mutual information (MI), which is defined as

$$I(x,y) = H(x) + H(y) - H(x,y) \quad \text{eq. 1}$$

Where $H(x)$ is information entropy of variable x and $H(x,y)$ is the joint information entropy between x and y . The *ennemi*¹⁴ Python package allows convenient computation of MI; the package provides a normalized index termed as MI correlation coefficient which varies between 0 (no relationship) and 1, and is defined as

$$\rho_{I(x,y)} = \sqrt{1 - e^{-2I(x,y)}} \quad \text{eq. 2}$$

¹⁴ <https://pypi.org/project/ennemi/>



The MI correlation coefficient still does not tell us if two variables are strictly nonlinearly related. Towards this end, Zhang et al.¹⁵ provided a useful approach combining ρ and ρ_I to generate a nonlinearity coefficient¹⁶, $r_{x,y}$, defined as

$$r_{x,y} = \rho_I(x,y) (1 - |\rho_{x,y}|) \quad \text{eq. 3}$$

$\rightarrow \in [0,1]$
 $\rightarrow = 1 \Rightarrow x \text{ and } y \text{ have strong nonlinear relationship}$
 $\rightarrow = 0 \Rightarrow x \text{ and } y \text{ have no nonlinear relationship}$

Let us concretize our understanding through a synthetic example.

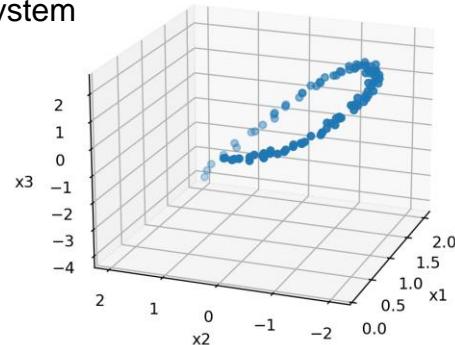
¹⁵ Zhang et al., A Novel Strategy of the Data Characteristics Test for Selecting a Process Monitoring Method Automatically. *Industrial & Engineering Chemistry Research*, 2016

¹⁶ $\rho_I(x,y)$ (and, therefore, $r_{x,y}$) may show near-zero negative values (<https://polys.github.io/enemi/potential-issues.html>) which can be interpreted as zero.

Example 3.1: Consider the following three-variable system

$$\begin{aligned}x_1 &= t + e_1 \\x_2 &= t^3 - 3t + e_2 \\x_3 &= -t^4 + 3t^2 + e_3\end{aligned}$$

$t \in (0,2]$ is sampled uniformly. Let's assess the system's nonlinearity. We will start with importing the required packages.

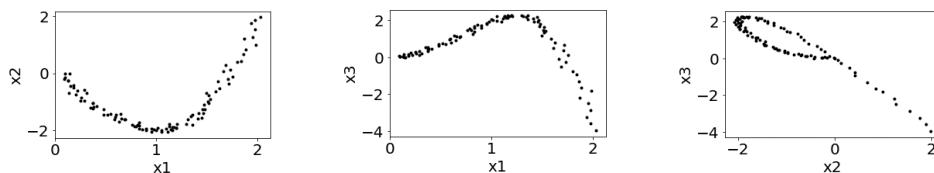


```
# import required packages
import numpy as np, matplotlib.pyplot as plt
from ennemi import pairwise_corr

# compute linear and Nonlinear correlation coefficients
data = np.hstack((x1,x2,x3)) # check online code to see how x1, x2, x3 vectors are generated
rho_xy = np.corrcoef(data, rowvar=False) # pair-wise linear correlation coefficients
rho_l_xy = pairwise_corr(data) # pair-wise normalized MI
rxy = rho_l_xy*(1-np.abs(rho_xy)) # pair-wise nonlinear correlation coefficients
```

$$\begin{array}{cc} r_{x,y} & \rho_{x,y} \\ \begin{matrix} x_1 & \left[\begin{matrix} nan & 0.61 & 0.78 \\ 0.61 & nan & 0.06 \\ 0.78 & 0.06 & nan \end{matrix} \right] \\ x_2 & \\ x_3 & \end{matrix} & \begin{matrix} x_1 & \left[\begin{matrix} 1 & 0.37 & -0.2 \\ 0.37 & 1 & -0.94 \\ -0.2 & -0.94 & 1 \end{matrix} \right] \\ x_2 & \\ x_3 & \end{matrix} \\ x_1 & x_2 & x_3 & x_1 & x_2 & x_3 \end{array}$$

As expected, the nonlinearity matrix indicates significant nonlinearity in the system (due to high nonlinearity coefficient values for the x_1 - x_2 and x_1 - x_3 pairs). The scatter plots below clearly corroborate this and also clarify why x_2 - x_3 pair appears as 'linearly' related.



For the entire dataset, Zhang et al. also proposed an overall nonlinear correlation coefficient defined as

$$r = \sqrt{\frac{\sum_{i,j=1}^m r_{ij}^2}{m^2-m}} \quad ; \text{ m is number of variables}$$

if no variable pair has nonlinear relationship, then $r = 0$

if all variable pairs have strong nonlinear relationship, then $r \rightarrow 1$

Since, linear methods can provide satisfactory monitoring performance for linear and weakly nonlinear dataset, Zhang et al. suggest a threshold between 0.3 and 0.5 for making the decision of using nonlinear models.



Industrial processes often operate around some optimal condition. Even though the actual process may be very complex and nonlinear, the data may exhibit only linear relationships among the variables. Therefore, do not jump the gun on choosing a nonlinear model; perform the nonlinearity check and then make an educated selection of an ML model for your system.

3.3 Gaussianity Assessment Techniques

We saw in Section 1 that wrong assumptions about Gaussianity of process data can lead to incorrect determination of NOC envelope. For multivariate Gaussianity assessment, an easy to implement method proposed by Zhang et al. will be presented here. The method uses the fact that if data is multivariate Gaussian distributed then the squared Mahalanobis distance (D) of samples from the center follow an F distribution¹⁷ as shown below¹⁸.

$$D(x) = (x - \bar{u})^T S_{cov}^{-1} (x - \bar{u}) \sim \frac{n(N^2-1)}{N(N-m)} F(m, N-m) \quad \text{eq. 4}$$

$x \in \mathbb{R}^m$ \bar{u} and S_{cov} are sample mean and covariance

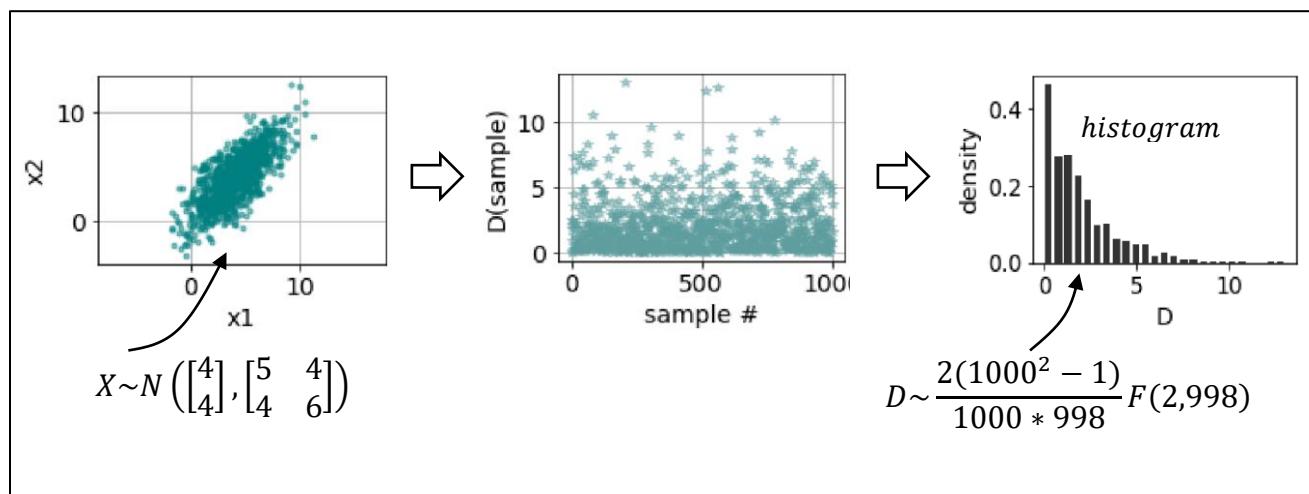


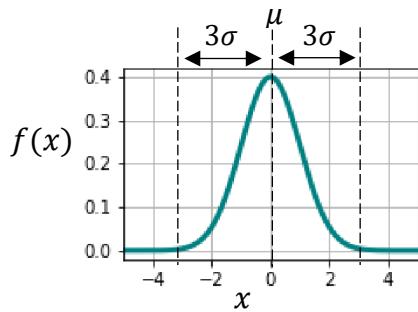
Figure 3.3: Distribution of Mahalanobis distances for a 2D Gaussian distribution

¹⁷ $F(m, N-m)$ imply an F distribution with m and $N-m$ degrees of freedom

¹⁸ Note that if S is singular then m is replaced by rank of S and S^{-1} by pseudo-inverse of S in Eq. (4)

Why Gaussianity is accorded so much importance

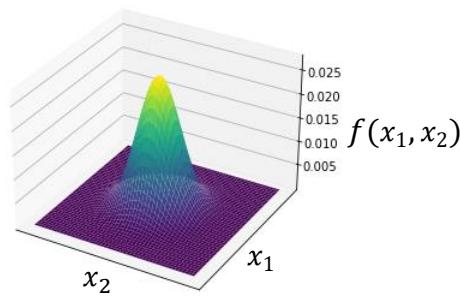
Illustration below shows the Gaussian distributions in 1D and 2D measurement space. Most of us are familiar with such histograms/distribution. But what is so special about these Gaussian distributed data?



$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

↓

$$P(\mu - 3\sigma < x < \mu + 3\sigma) = 0.9973$$



$$f(x) = \frac{1}{\sqrt{(2\pi)^m |\Sigma|}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}$$

Probability that x lies in the shown range

As shown in the expressions above, Gaussian signals are completely characterized by only two parameters, mean (μ) and covariance (σ^2/Σ), and therefore, are easy to describe and work with. Fortunately, most industrial processes operate around some optimal state and exhibit natural-cause Gaussian variations under the influence of random process disturbances. Therefore, many traditional MSPM and SPC techniques assume Gaussianity and correspondingly, many results established in their literature are valid strictly for Gaussian processes.

Once the Mahalanobis distances are available, the empirical cumulative distribution function (CDF) is compared against the CDF of the expected F distribution. The algorithm below summarizes the approach.

Algorithm: Multivariate Gaussianity test based on Mahalanobis distances

- Generate fractile-fractile plot of statistic D 's empirical distribution and expected F distribution

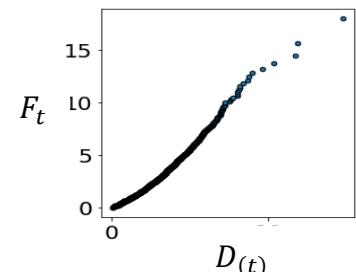
- Values of D are sorted such that

$$D_t: D_{(1)} \leq D_{(2)} \leq \dots \leq D_{(N)}$$

- Empirical CDF of D_t can be described as

$$CDF(D_{(t)}) \approx \frac{t-0.5}{N} = r_t \quad (t = 1, 2, \dots, N)$$

Here, $D_{(t)}$ is the fractile corresponding to probability r_t



- The fractile of expected F distribution corresponding to probability r_t is obtained as follows

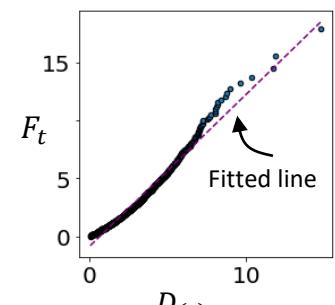
$$F_t \approx \frac{m(N^2-1)}{N(N-m)} F_{rt}(m, N-m)$$

- Plot $D_{(t)}$ vs F_t for $t = 1, 2, \dots, N$ and fit a straight line. If data follows multivariate Gaussianity then the fitted line should pass through origin with slope equal to 1

- Perform statistical tests:

- A significance test is performed to check if linear relationship between $D_{(t)}$ vs F_t is statistically significant.

The following condition is imposed



$$\left(\frac{S}{\bar{F}} \right) < 0.15 \Rightarrow \text{linear fit is not rejected}$$

$\bar{F} = \frac{1}{N} \sum F_t$

$$S = \sqrt{\frac{SSE}{N-2}}$$

- If the above condition is met, then the intercept and slope are compared against 0 and 1, respectively, via the following conditions

$$|intercept| < 0.05 * \bar{F}$$

$$|slope - 1| < 0.2$$

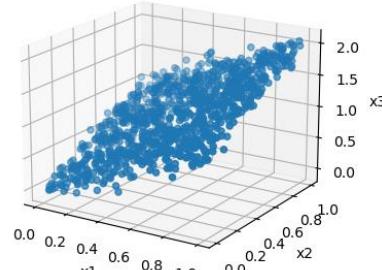
- If all the above three conditions are met, then multivariate Gaussian distribution is assumed

The example below shows the implementation steps¹⁹ of the above algorithm.

Example 3.2: Consider the following three-variable system

$$x_3 = x_1 + x_2 + e$$

x_1 and $x_2 \in [0,1]$ are sampled uniformly. Let's assess the system's non-Gaussianity.



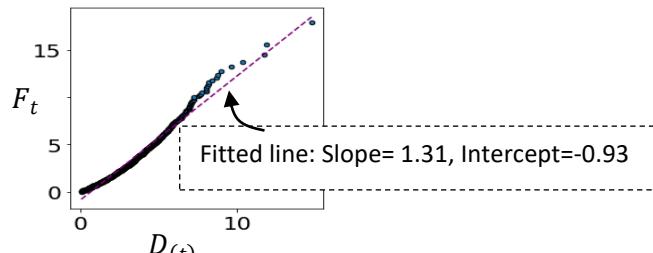
```
# import required packages
import numpy as np, matplotlib.pyplot as plt
import scipy.spatial, scipy.stats

# find data statistics
data = np.vstack((x1,x2,x3)).T # check online code to see how x1, x2, x3 vectors are generated
N, m = data.shape[0], data.shape[1]
mu, Scov = np.mean(data, axis=0), np.cov(data, rowvar=False, ddof=1)
Scov_inv = np.linalg.pinv(Scov)

# calculate D statistic and the fractiles
D = np.array([scipy.spatial.distance.mahalanobis(data[i,:], mu, Scov_inv)**2 for i in range(N)])
Dt, rt = np.sort(D), [(t-0.5)/N for t in range(1,N+1)]
Ft = (m*(N**2-1)/(N*(N-m)))*np.array([scipy.stats.f.ppf(p, m, N-m) for p in rt])

# fit a straight line
linearFit = scipy.stats.linregress(Dt, Ft)
intercept, slope = linearFit[1], linearFit[0]

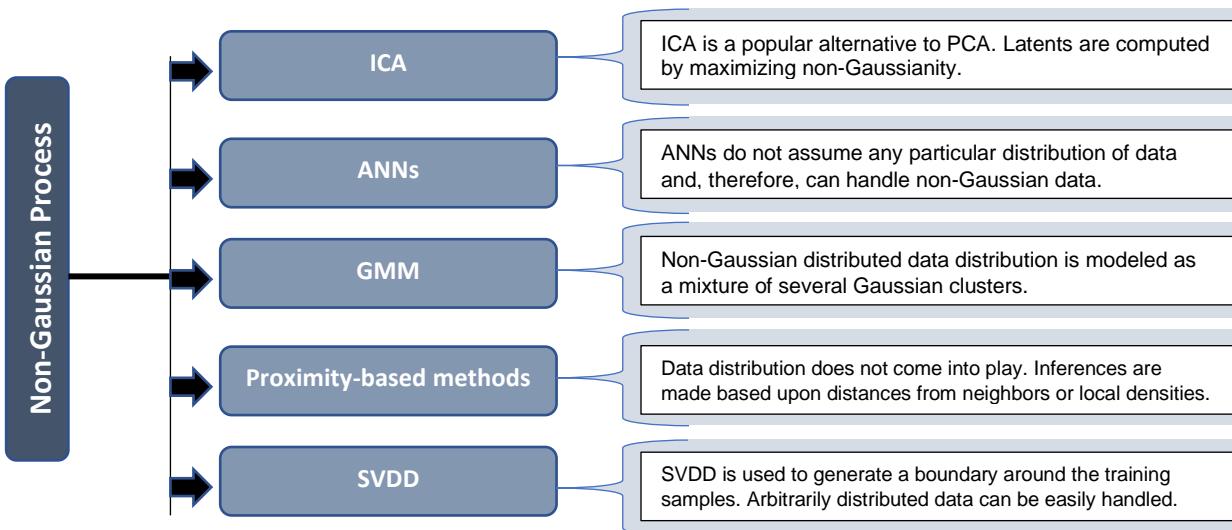
# perform significance tests
Fbar = np.mean(Ft)
Sfit = np.sqrt(((Ft-(intercept+slope*Dt))**2).sum()/(N-2))
Sfit_by_Fbar = Sfit/Fbar
if Sfit_by_Fbar > 0.15:
    gaussianity = False
else:
    if np.abs(slope-1) < 0.2 and np.abs(intercept) < Fbar*0.05:
        gaussianity = True
    else:
        gaussianity = False
```



This condition fails \Rightarrow Gaussianity= False

¹⁹ Code adapted from KydLIB package (<https://github.com/afraniomelo/KydLIB/>)

Common ML modeling methodologies for multivariate non-Gaussian data



3.4 Dynamics Assessment Techniques

In a dynamic dataset, the recorded samples are serially correlated, i.e., the current measurements are not independent of the past measurements. We saw in Section 1 that models designed to work with steady state data can fail to capture the temporal relationships which lead to poor fault detection performance. The unmodeled serial correlations show up in model residuals as illustrated in Figure 3.4. The presence of serial correlations in model residuals is assessed through autocorrelation function (ACF) which computes the linear correlation coefficients for the residual time series for different lags. If no significant dynamics are present, then the ACF values should lie close to 0 for lags > 0 . This rationale is used by Sun & Braatz in the SPA package to automatically decide the need for a dynamic model.

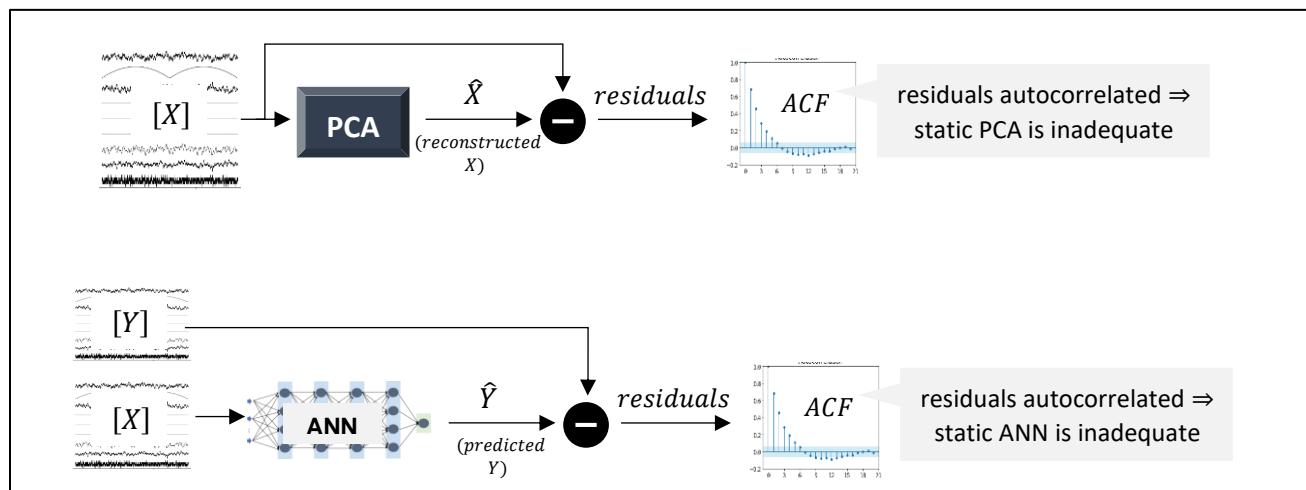


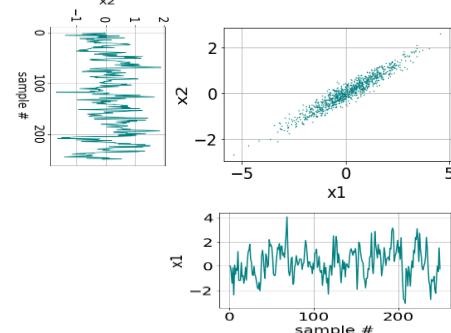
Figure 3.4: Serial correlations in model residuals upon usage of static models

The previous approach entails creation of a static model to generate residuals. An alternative approach followed by Melo et al. in KydLIB package uses ACF over the raw measurement itself and not the residuals. The ‘time necessary’ to reach an autocorrelation coefficient value of 0.5 was plotted for all the variables to assess the dynamic behavior of the dataset. A low mean or median time required to achieve 0.5 autocorrelation signifies low level of dynamics in the data.

Example 3.3: Consider the following 2D system that we saw in Section 1

$$\begin{aligned}x_1(t) &= 0.7x_1(t-1) + e_1(t) \\x_2(t) &= 0.5x_1(t) + e_2(t)\end{aligned}$$

e_1 and e_2 are Gaussian noise. Let's assess the system's dynamics.

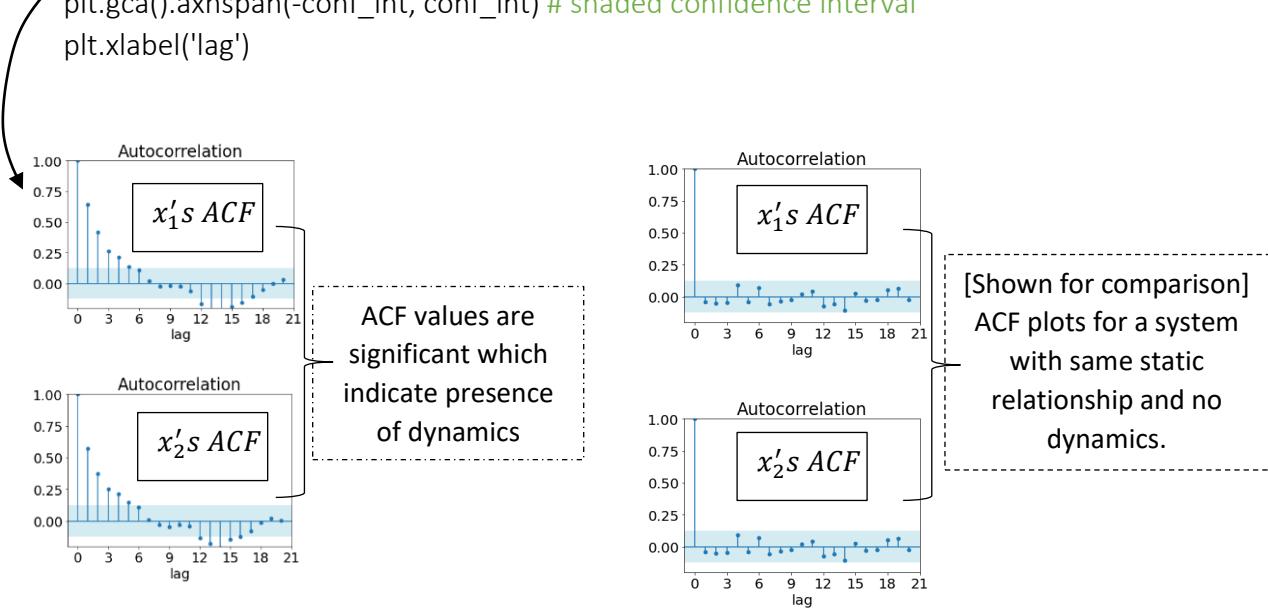


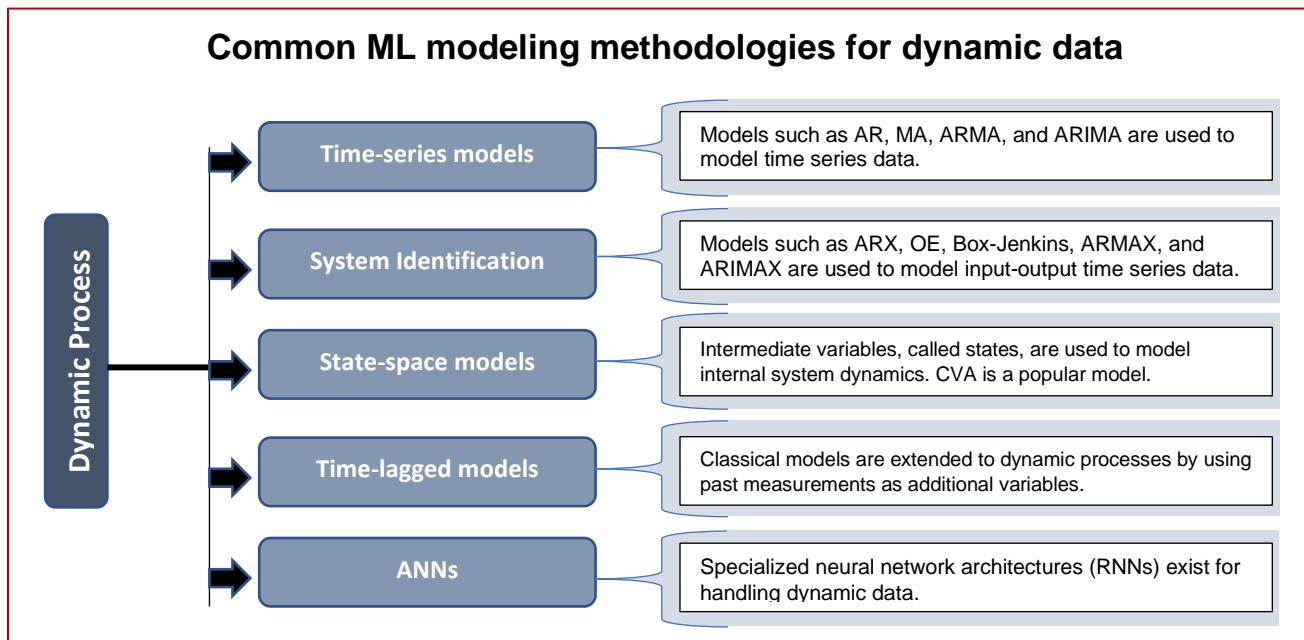
```
# import required packages
import numpy as np, matplotlib.pyplot as plt

# check for dynamics via ACF plot
from statsmodels.graphics.tsaplots import plot_acf
conf_int = 2/np.sqrt(len(x1)) # check online code to see how x1 and x2 vectors are generated

plot_acf(x1, lags= 20, alpha=None)
plt.gca().axhspan(-conf_int, conf_int) # shaded confidence interval
plt.xlabel('lag')

plot_acf(x2, lags= 20, alpha=None)
plt.gca().axhspan(-conf_int, conf_int) # shaded confidence interval
plt.xlabel('lag')
```

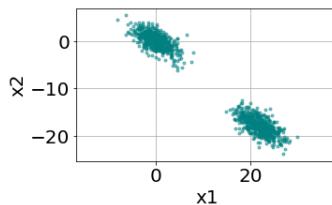




3.5 Multimode/Multi-clustered Distribution Assessment Techniques

The most common approach for determining the presence of multimodality in data is to divide the dataset into distinct clusters and look for the optimal number of clusters. Techniques such as k-means clustering and Gaussian mixture modeling exist for data clustering. We will defer a detailed discussion on these techniques to Chapter 11. Nonetheless, we show in the example below how the dataset shown in Figure 3.1 can be assessed for multimodality. In Chapter 16, you will see a neural network-based visualization technique called Self Organizing Maps that can provide a 2D view of high-dimensional dataset and, therefore, help to check for the presence of clusters.

Example 3.4: Consider the following 2D multi-clustered data distribution



Let's see how we can automatically find the number of operation modes/clusters.

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

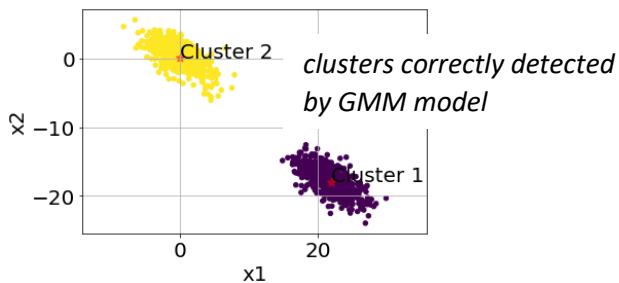
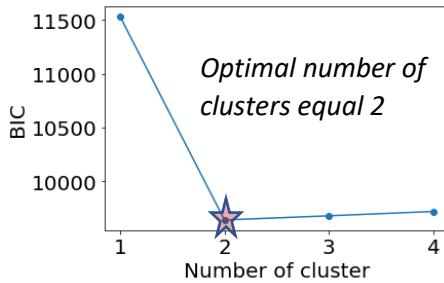
# check for clusters via Gaussian Mixture Modeling using Bayesian Information Criterion
from sklearn.mixture import GaussianMixture

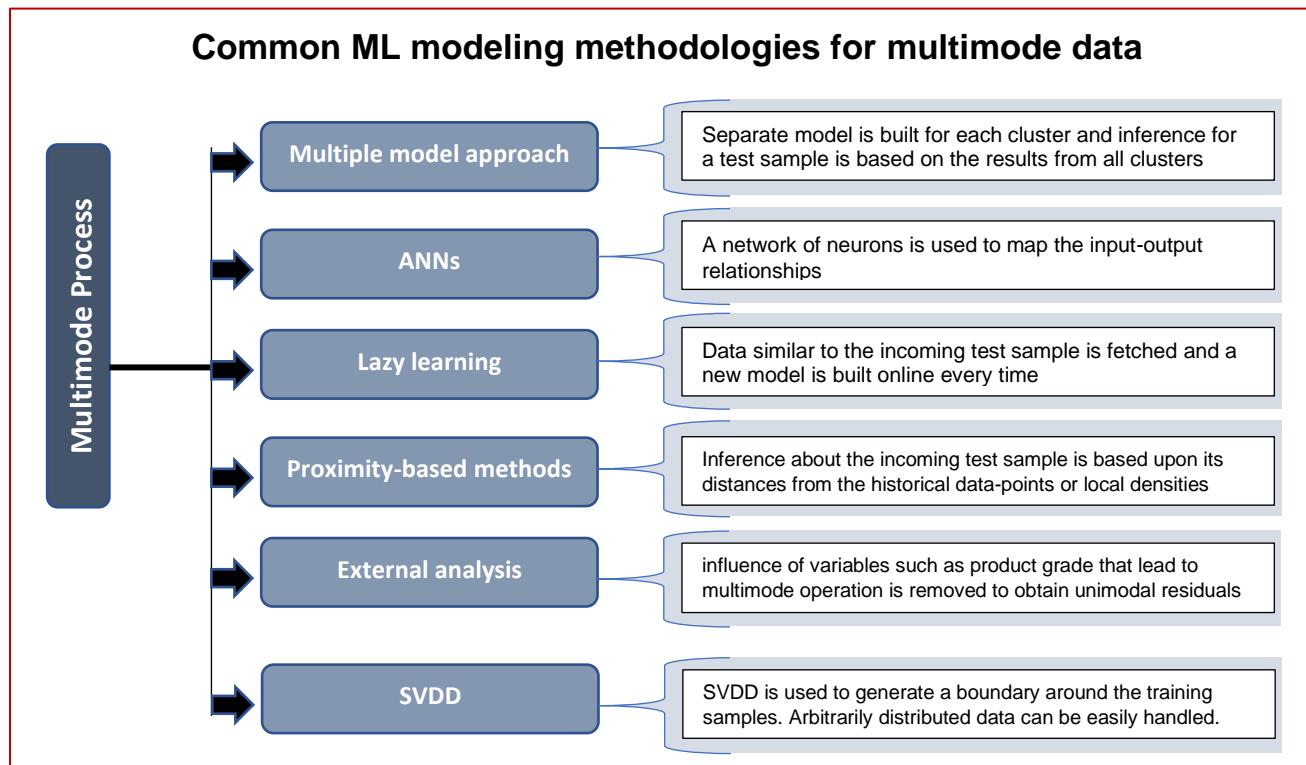
BICs = []
lowestBIC = np.inf
for n_cluster in range(1, 5):
    gmm = GaussianMixture(n_components = n_cluster, random_state = 100).fit(data)
    BIC = gmm.bic(data) # check online code to see how x1 and x2 vectors are generated
    BICs.append(BIC)

    if BIC < lowestBIC:
        optimal_n_cluster = n_cluster
        lowestBIC = BIC

print (optimal_n_cluster)

>>> 2
```





3.6 Data Characteristics Investigation of Tennessee Eastman Process Dataset

In this case study, we will bring together the concepts learnt to analyze a celebrated dataset, the Tennessee Eastman Process (TEP) dataset. If you work in the area of process monitoring, then you are very likely to encounter research papers using TEP dataset to demonstrate efficacy of their FDD techniques. Therefore, let's use this to put the concepts learnt into practice. Melo et al. have analyzed the nonlinearity, non-Gaussianity, and dynamics characteristics of this dataset. We will try to reproduce those results and also assess the presence of multimodal distribution.

The TEP dataset²⁰ comes from simulation of a large-scale industrial chemical plant. The plant consists of several unit operations: a reactor, a condenser, a separator, a stripper, and a recycle compressor²¹. There are 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. The NOC training

²⁰ available at <https://github.com/camaramm/tennessee-eastman-profBraatz>

²¹ Detailed information about the process and the faults can be obtained from the original paper by Downs and Vogel titled 'A plant-wide industrial process control problem'.

data file (*d00.dat*) contains 500 samples. Since this file is often used to build models, we will analyze it in detail. We will exclude the composition measurements from our analysis. Figure 3.5 and 3.6 shows the process flowsheet and the line plots of the NOC data, respectively.

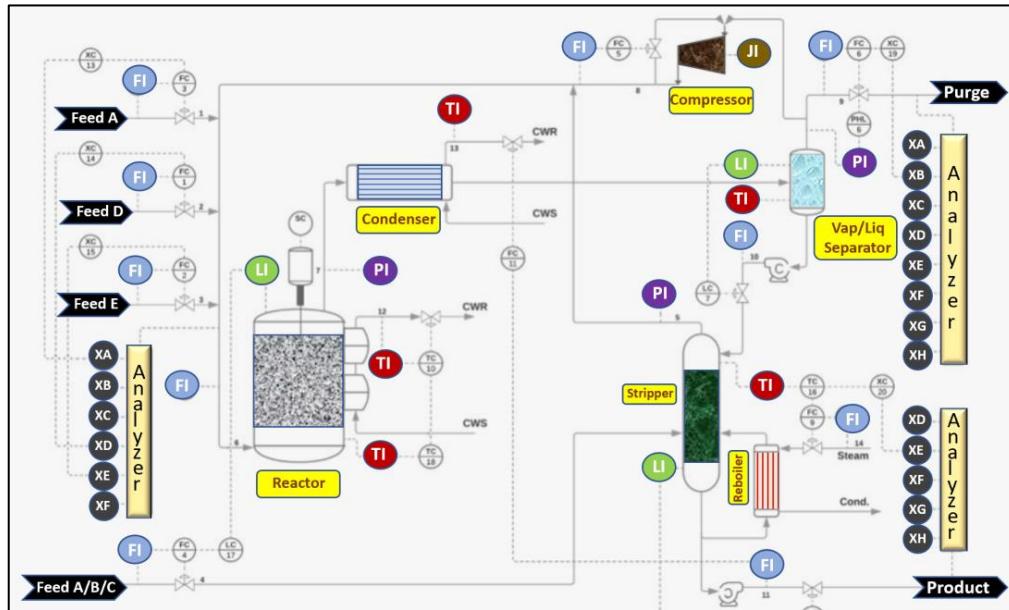


Figure 3.5: Tennessee Eastman process flowsheet²²

Let's load the data to be analyzed.

```
# read TEP NOC training data
import numpy as np
TEdata_noFault_train = np.loadtxt('d00.dat').T
xmeas = TEdata_noFault_train[:,0:22] # continuous measured
xmv = TEdata_noFault_train[:,41:52] # manipulated variables
data = np.hstack((xmeas, xmv))
print(data.shape)

>>> (500, 33)
```

²² Adapted from the original flowsheet by Gilberto Xavier (<https://github.com/gmxavier/TEP-meets-LSTM>) provided under Creative-Commons Attribution 4.0 International License (<https://creativecommons.org/licenses/by/4.0/>).

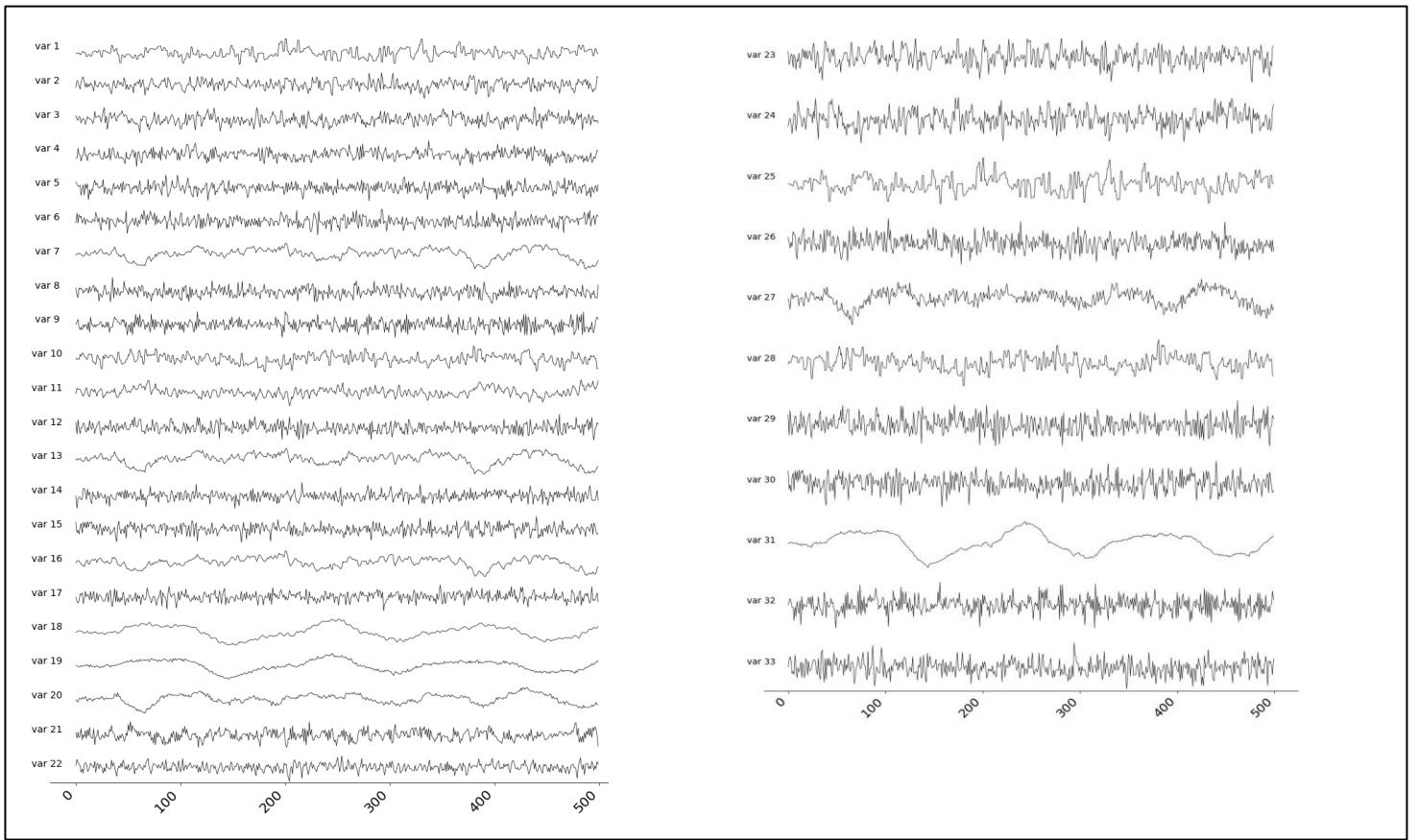
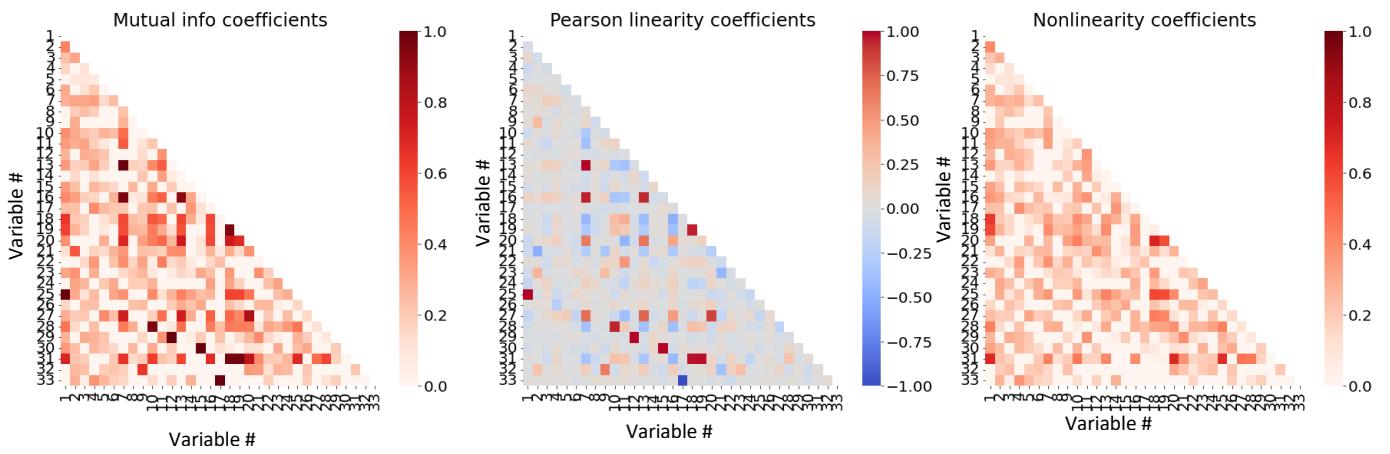


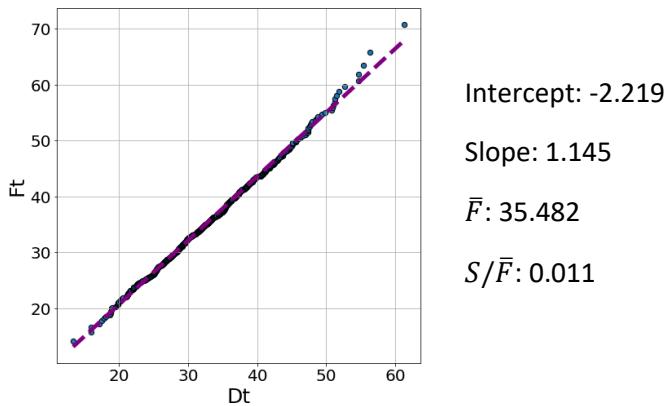
Figure 3.6: Line plots of continuous process measurements and manipulated variables

The plots below show the heat maps for the different correlation coefficients. The code for the computation of the coefficients is the same as that shown in Section 3.2 and therefore not reproduced here²³. The overall linearity and nonlinearity coefficient comes out to be 0.21 and 0.20, respectively, indicating low levels of both linear and nonlinear correlations.

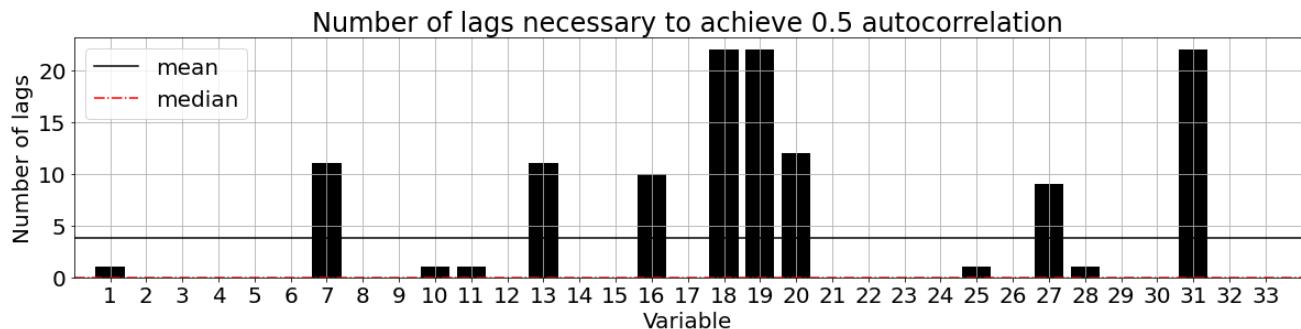


²³ The complete code used for TEP data assessment can be obtained from the GitHub repository

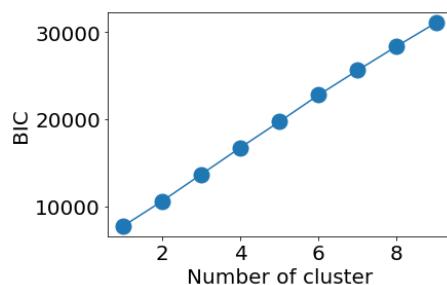
The fractile-fractile plot and the fitted straight line for the Gaussianity assessment are shown below. Based on the statistics obtained, the dataset is assessed to be non-Gaussian.



The plot below shows the 'lag necessary' to reach an autocorrelation coefficient value of 0.5 for each of the 33 variables. The low value of median lag required signifies overall low level of dynamics in the data.



The BIC curve from GMM modeling of the TEP data indicates that the data is unimodal distributed.



The above exercise indicates that the analyzed TEP dataset exhibits linear, static, non-Gaussian, and unimodal characteristics. Correspondingly, you will see an application of ICA

technique (which is well-suited for such datasets) employed for TEP fault detection in Part 3 of the book.

Summary

In this chapter, we looked at methods to assess the different characteristics of process data; specifically, we focused on the presence of nonlinearity, non-Gaussianity, dynamics, and multi-modality. We also performed an investigation of the process data characteristics from Tennessee Eastman Chemical plant simulation mimicking a real-scale complex industrial system. With the techniques learnt in this chapter, you will no longer be working in the dark while selecting a model suitable for your dataset. In the next chapter, we will look at some more guidelines on best practices for ML model development for plant health management.

Chapter 4

Machine Learning for Plant Health Management: Workflow and Best Practices

Whether you are building a ML solution for fault detection, fault classification, or fault prognosis, model development is the most critical task. Inarguably, obtaining a good ML model is not a trivial task. You cannot obtain a good model by just dumping all the available raw data in an off-the-shelf machine learning module. Incorrectly specify one hyperparameter and your model will return garbage results; provide insufficiently ‘rich’ training dataset and even the most carefully chosen ML model will prove incapable of providing meaningful insights. Unfortunately, an automated procedure for ML model development that works for all types of problems does not exist. Nonetheless, there is no cause for despair. The trick to successful model development lies in being actively involved in the several model development stages and making use of several useful guidelines that the ML community has come up with over the years. We already saw in the previous chapter the importance of acquiring a good understanding of data for correct model selection. In this chapter, we will learn several other guidelines and the best practices.

We will not cover the best practices associated with generic machine learning workflow. Concepts like feature extraction, feature engineering, cross-validation, regularization, etc., have already been covered in detail in our first book of the series. Albeit we will touch upon topics that are specific to plant health management applications. In this chapter, our focus will be on aspects that you should not ignore to ensure that you are not unknowingly setting your model up for failure. Specifically, we will cover these topics

- ML model development workflow
- Data selection and pre-processing to obtain good training dataset
- Assessment of monitoring performance
- Best practices for model selection and tuning

4.1 ML Model Development Workflow

The prime objective of the ML modeling task for building process modeling solutions is to obtain a model that provide high fault sensitivity (i.e., the model is able to detect process faults in incipient stages) and low false alarm rate (i.e., the model does not report a process fault when process is operating normally). Balancing the trade-off between these two requirements is not easy and requires careful attention to varied aspects during model development. It definitely takes more than just executing a ‘model = <some ML_model>.fit()’ command on the available data. In Chapter 1, we saw an overview of the typical steps involved in a ML model development exercise. In this chapter, we will look at the different components of the workflow in more details. Figure 4.1 lists the subtasks that we will touch upon. While separate books can be written on each of these subtasks, we will focus on the aspects that may get overlooked by an inexperienced process data scientist.

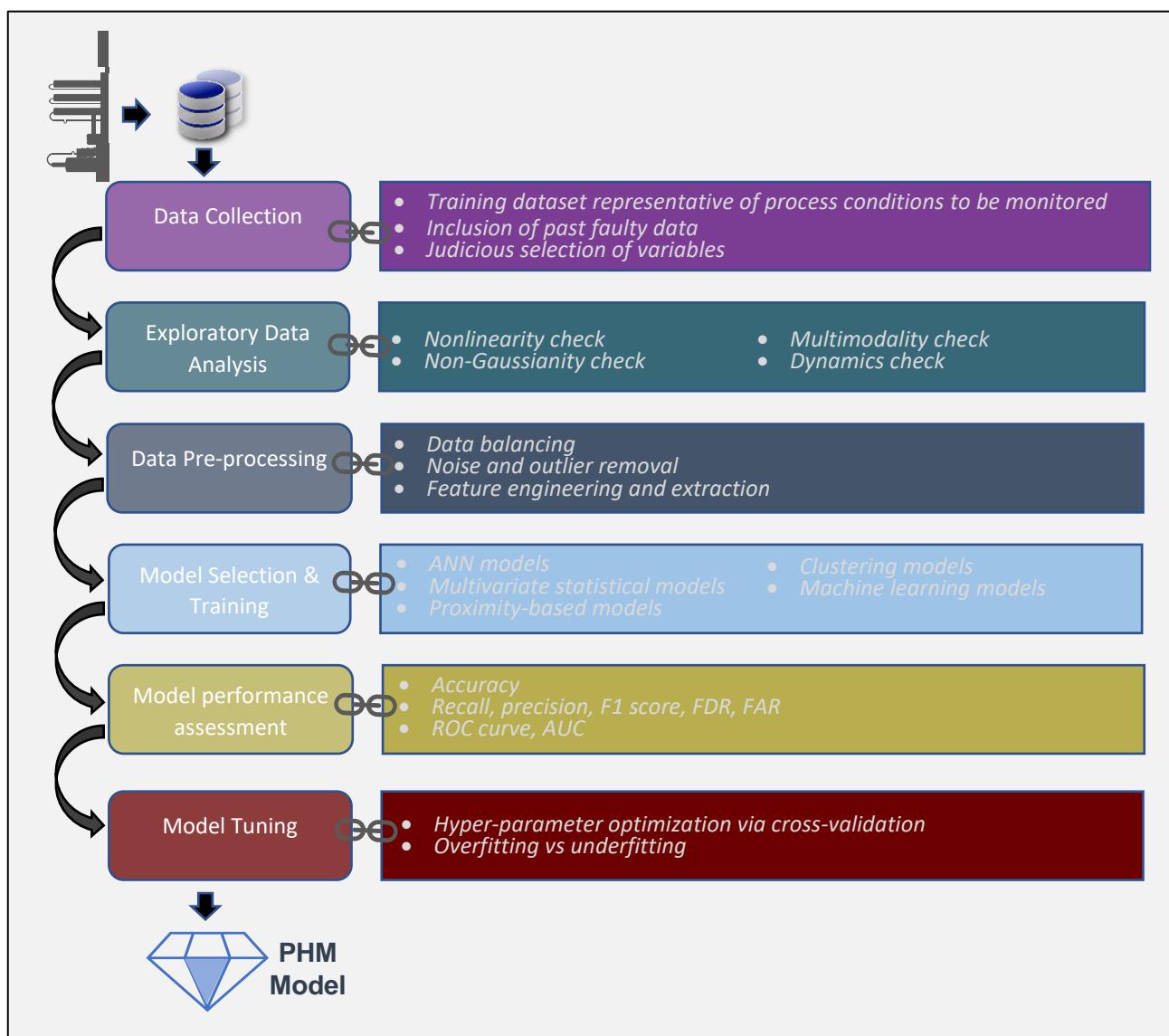


Figure 4.1: ML model development workflow

The workflow starts with data collection. Although most often you won't have much control over the data (process data collected in historical database) that is available for model building, you can take a few steps to ensure that the 'right' data get passed down the workflow. For example, you can ensure that only the NOC samples that are representative of the process conditions that will be monitored are selected for further processing; this ensures that the model learns the correct fault-free process behavior. Thereafter, you move onto the exploratory data analysis step wherein you get a 'feel' of the data. In the previous chapter, we saw in great detail how EDA can help us select the right class of model. In the next step, data are pre-processed to increase the information content in the data. This step can involve data cleaning (e.g., removing outliers and noise), data transformation (generating new features and/or modifying existing ones), etc. After this step, we are ready to fit out chosen model.

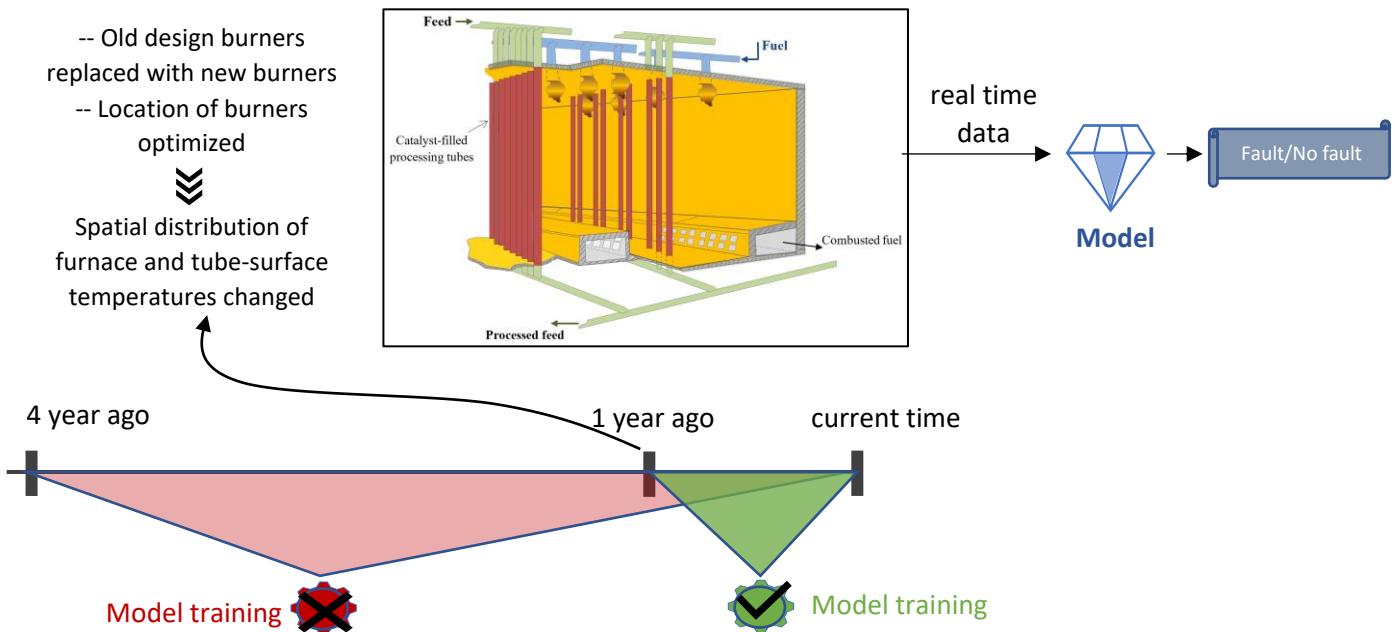
A practical guideline for model selection is Occam's razor which advises selection of simplest model that meets the modeling objectives. In Chapter 1, we saw several factors that influence model selection. These factors along with inferences from the EDA step will help you shortlist your candidate models. After model fitting, an evaluation of the model's fault detection and fault diagnosis performance is conducted. Diverse set of metrics has been devised to quantify different aspects of FDD performance. We will look at these metrics in this chapter. The last step before you obtain your coveted model is model tuning where you play with your model's hyperparameters. We will study how you can use cross-validation for this step. With this broad overview of the workflow, let's take a closer look into some of the aspects.

4.2 Data Selection

At the 'data selection' step of the ML model development workflow, as a process modeler, you need to take care that you are not accidentally providing a dataset that 'confuses' the ML model! To understand this further, let's take a look at a couple of example scenarios.

Training data representative of normal operating conditions

Most of the ML models used for process monitoring applications work by 'learning' the normal process behavior. Now consider the illustration below. The furnace underwent significant design changes a year ago due to which the normal values of temperatures inside the furnace has changed. Now imagine if historical data over the last four years are utilized for model training.



Any model will struggle to adequately describe the furnace temperatures during the last 1 year. The model-plant mismatch will add significantly to the ‘noise’ in the dataset which will severely impact the fault sensitivity of the final model. The model performance is bound to be disappointing. A sensible approach would be to use only the last 1 year of data as the training data.

Avoiding inclusion of unnecessary variables

A general tendency among inexperienced process data scientists is to dump all the available process variables into model training module. They rely upon feature selection to weed out irrelevant variables. However, feature selection may not always do a decent job; for example, if you are building a PCA model and enough historical faulty samples are not available then inferences regarding variable relevancy are difficult to make algorithmically. It is always wiser to pay careful attention upfront and make educated selection of variables.



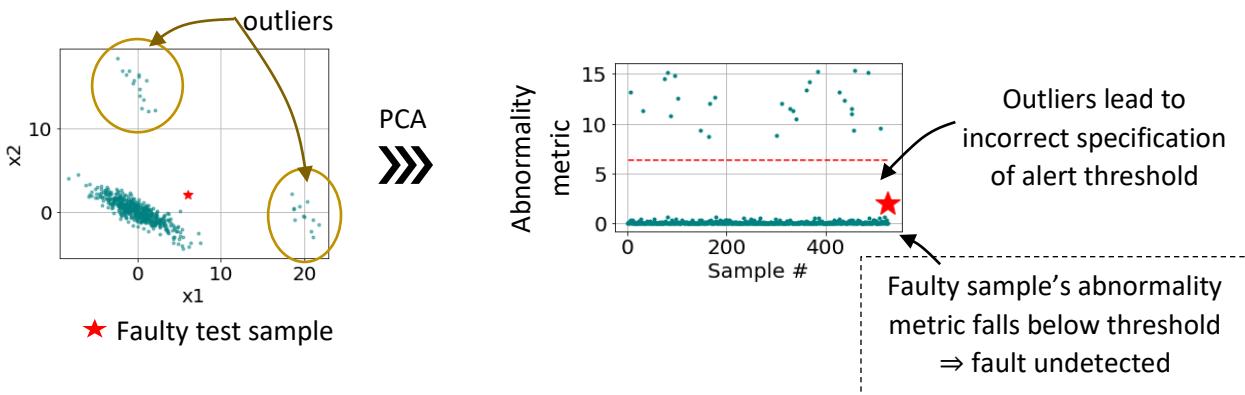
The takeaway message is that if you are not an ‘insider’ to the plant operations then it would serve you well to sit down with the plant manager (and plant engineer) to obtain as much information about the process as possible; use the information gained to curate the training dataset.

4.3 Data Pre-processing

As alluded to in Chapter 1, the overall objective of this step is to increase the ‘information content’ of training dataset so that the model’s ability to distinguish between normal and faulty operations is bolstered. For example, model’s performance will suffer if training dataset is corrupted with outliers or does not have adequate number of faulty samples. During pre-processing step, training dataset is treated to remove these deficiencies. Let’s look at a couple of deficiencies that are easy to get overlooked (/not handled properly) in pursuit of ‘quick’ results resulting in misleading inferences.

Handling outliers

In simple words, outliers are corrupted training samples that are not consistent with the normal training samples. As should be obvious, outliers can ‘confuse’ models during extraction of the true normal behavior of the process from training data. The illustration below shows how unhandled outliers can lead to failures going undetected.

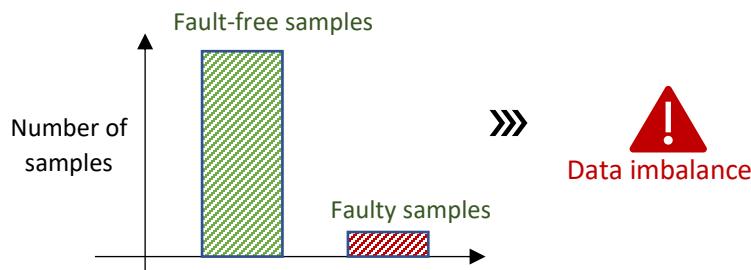


In Chapter 4 of Book 1 of this series, we covered in detail several techniques for removal of outliers; interested readers are referred to it for more details. For the dataset in the above illustration, the correct modeling approach would be to either explicitly remove the outliers prior to model fitting or implement robust PCA that is less susceptible to outliers and choose control limits with correct significance level (such as 90% control limit instead of 95%).

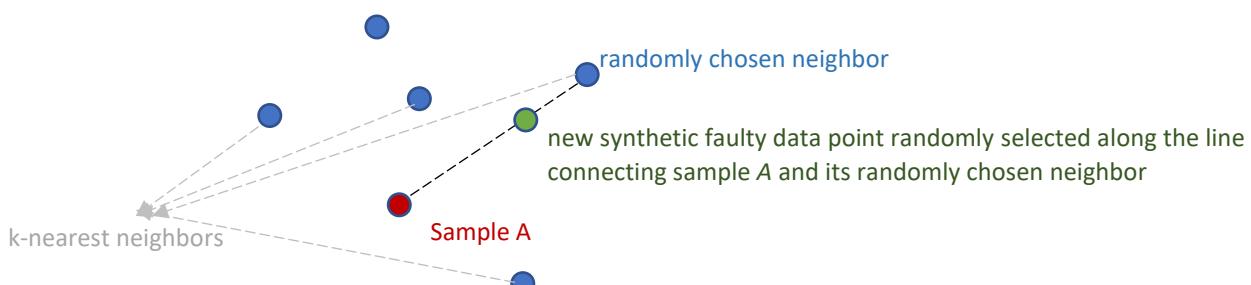
In high-dimensional dataset, it is not trivial to detect the presence of outliers from just a ‘quick glance’ and therefore, you may be tempted to skip outlier handling. However, as a best practice, your default assumption must always be that the training dataset is potentially infested with outliers and appropriate measures, as deemed suitable for the system at hand, must be put in place to handle outliers.

Handling data imbalance

If you are building a fault classification model, then you will invariably encounter the problem of data imbalance, wherein the number of data samples corresponding to faulty operations is much fewer compared to the number of NOC samples. Data imbalance is undesirable because it leads to the ML model training being dominated by NOC samples, resulting in the model not able to learn sufficiently about the faulty samples and eventually failing in correctly detecting (/classifying) process faults.



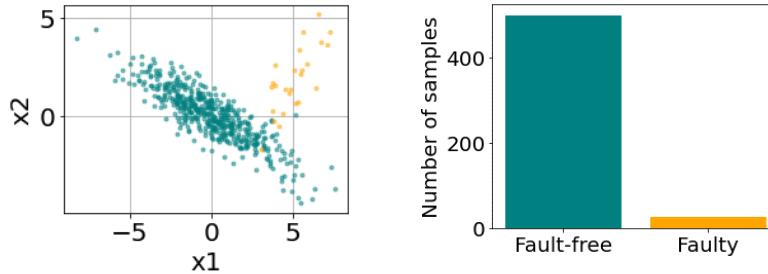
How can data imbalance be rectified? One trivial strategy could be to discard most of the NOC samples (also termed undersampling) but this obviously leads to loss of crucial process information. The other approach would be to add more faulty samples to training dataset, but plant managers will not let you inject faults in their processes just so that your model can be trained. A popular mechanism to handle data imbalance is SMOTE (synthetic minority oversampling technique) wherein synthetic faulty samples are generated. The basic idea is illustrated below. Here, a random faulty sample (*A*) is selected and a synthetic data point is generated as illustrated. This procedure is repeated to create as many synthetic faulty samples as desired.



Example 4.1: We will generate an imbalanced dataset of NOC and faulty samples. Then we will learn how to use the imbalanced-learn package to balance the dataset.

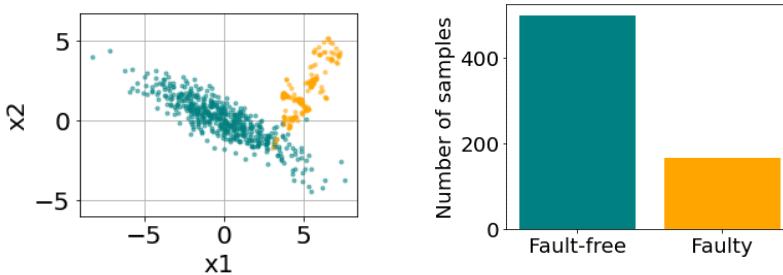
```
# import required packages
import numpy as np, matplotlib.pyplot as plt
from imblearn.over_sampling import SMOTE
```

```
# generate data
cov = np.array([[6, -3], [-3, 2]])
pts_NOC = np.random.multivariate_normal([0, 0], cov, size=500)
cov = np.array([[1, 1], [1, 2]])
pts_Faulty = np.random.multivariate_normal([5, 2], cov, size=25)
X = np.vstack((pts_NOC, pts_Faulty))
y = np.vstack((np.zeros((500,1)), np.ones((25,1)))) # labels [0=>NOC; 1=>Faulty]
```



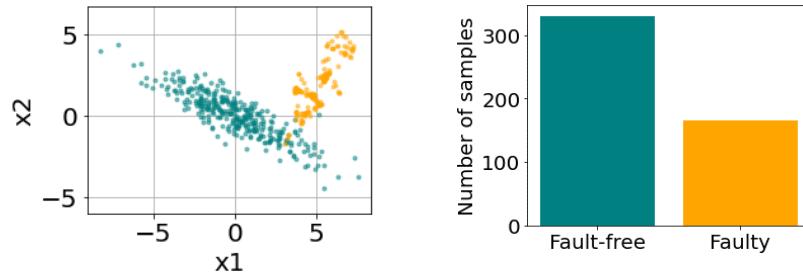
We will oversample faulty samples using SMOTE and specify the ratio of faulty to fault-free samples to be 0.33.

```
# Oversampling
overSampler = SMOTE(sampling_strategy=0.33)
X_smote, y_smote = overSampler.fit_resample(X, y)
```



A common practice is to augment SMOTE with undersampling of the NOC samples. The *RandomUnderSampler* class of *imblearn* package can be utilized for this. In Example 4.1's dataset, we will increase ratio of faulty to fault-free samples to be 0.5 as shown below.

```
# Undersampling
from imblearn.under_sampling import RandomUnderSampler
underSampler = RandomUnderSampler(sampling_strategy=0.5)
X_balanced, y_balanced = underSampler.fit_resample(X_smote, y_smote)
```



Now that your training dataset has been enriched, you can expect your fault classifier to perform much better!

4.4 Model Evaluation

Post model fitting, we need some means of assessing the model performance to quantify how well the model detects faults. The most straightforward metric is *accuracy* which simply communicate the ratio of true predictions to the total number of predictions. Accuracy, however, can be misleading for imbalanced datasets. For example, if your model always trivially predicts ‘no fault’ and your dataset has 95% NOC samples then the model’s accuracy will be 95%, which is deceptively good! Therefore, several other metrics have been devised that quantify different aspects of fault detection capabilities of a model. Before we introduce these metrics, let’s first understand a confusion matrix. As shown below, a confusion matrix for a fault detection (binary classification) model provides a comprehensive overview of how well the model has correctly (and incorrectly) classified samples belonging to positive (faulty) and negative (normal/fault-free) classes.

Positive \Rightarrow Faulty Negative \Rightarrow Fault-free	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2"></th> <th colspan="2" style="background-color: #e0e0e0;">Predicted Class</th> </tr> <tr> <th colspan="2"></th> <th style="background-color: #e0e0e0;">Positive</th> <th style="background-color: #e0e0e0;">Negative</th> </tr> <tr> <th rowspan="2" style="writing-mode: vertical-rl; transform: rotate(180deg);">True Class</th> <th style="background-color: #e0e0e0;">Positive</th> <td style="background-color: #003366; color: white; padding: 5px;">(TP) True Positive 67</td> <td style="background-color: #e0e0e0; padding: 5px;">(FN) False Negative 1</td> </tr> </thead> <tbody> <tr> <th style="background-color: #e0e0e0;">Negative</th> <td style="background-color: #e0e0e0; padding: 5px;">(FP) False Positive 4</td> <td style="background-color: #e0e0e0; padding: 5px;">(TN) True Negative 21</td> </tr> </tbody> </table>			Predicted Class				Positive	Negative	True Class	Positive	(TP) True Positive 67	(FN) False Negative 1	Negative	(FP) False Positive 4	(TN) True Negative 21	Actual Positive (AP) = TP + FN Actual Negative (AN) = FP + TN
		Predicted Class															
		Positive	Negative														
True Class	Positive	(TP) True Positive 67	(FN) False Negative 1														
	Negative	(FP) False Positive 4	(TN) True Negative 21														

Using the terms from the confusion matrix, the table below summarizes the varied performance metrics.

False Alarm Rate (FAR) or False Positive Rate (FPR)	Fault Detection Rate (FDR) or True Positive Rate (TPR) or Sensitivity or Recall	Missed Detection Rate or False Negative Rate (FNR)
<i>Proportion of negatives that are flagged as positives</i>	<i>Proportion of positives that are correctly flagged</i>	<i>Proportion of positives that are flagged as negatives</i>
$FAR = \frac{FP}{AN} = \frac{FP}{FP + TN}$	$FDR = \frac{TP}{AP} = \frac{TP}{TP + FN}$	$FNR = \frac{FN}{AP} = \frac{FN}{FN + TP}$
Precision or Positive Predictive Value (PPV)	F1 Score	Accuracy (ACC)
<i>Proportion of positive predictions that are correct</i>	<i>Harmonic mean of precision and recall</i>	<i>Proportion of correct predictions</i>
$Precision = \frac{TP}{TP + FP}$	$F1 = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$	$ACC = \frac{TP + TN}{TP + TN + FP + FN}$

Table 1: Performance metrics for a fault detection model

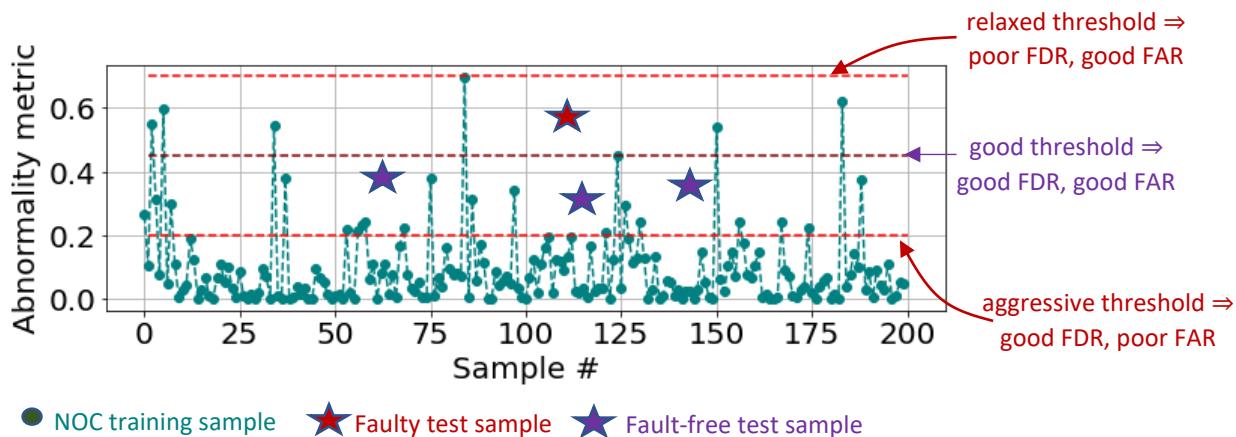
You will encounter the terms precision and recall commonly in the anomaly detection literature. Precision metric returns the ratio of number of correct positive predictions to the total number of positive predictions by the model while recall returns the ratio of number of correct positive predictions to the total number of positive samples in the data. For a process monitoring tool (where positive label implies presence of a process fault), recall denotes model's ability to detect process fault, while precision denotes accuracy of model's prediction of process fault. A model can have high recall if it detects process faults successfully, but occurrence of lots of false alarms will lower its precision. In other scenario, if the model can detect only a specific type of process fault and gives very few false alarms then it will have low recall and high precision. A perfect model will have high values (close to 1) for both precision and recall. However, we just saw that it is possible to have one of these two metrics high and the other low. Therefore, both metrics need to be reported. However, if only a single metric is desired then F1 score is utilized which returns a high value if both precision and recall are high and a low value if either precision or recall is low.

In process industry, recall (FDR) and false alarm rate (FAR) are more commonly employed to assess a process monitoring tool's performance. A plant operator is mainly concerned about two things: does the model detect a fault when it occurs (given by FDR) and does the model trigger false alerts under NOC (given by FAR). As would be obvious, FAR close to 0 is

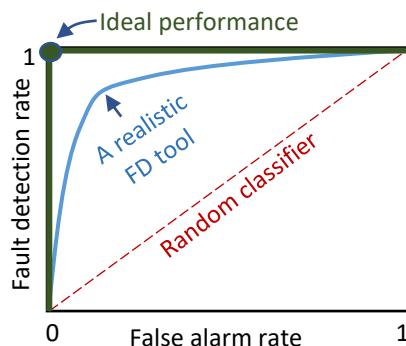
desirable. A well-designed monitoring tool has high FDR and low FAR values. Low FDR and high FAR values are both undesirable because while low FDR value leads to safety issues and economic losses due to delayed/no abnormality detection, high FAR value leads to loss of user's confidence in the tool and eventually tool's demise. If you make your model very sensitive to detect faults promptly at incipient stages, it will very likely generate lots of false alerts. As a process data scientist, you will very often find yourself tuning the model to balance the trade-off between FDR and FAR.

ROC and area under curve (AUC)

Most of the fault detection models compute abnormality metric(s) in one form or the other (Q and T^2 in PCA, distance from the center in SVDD, etc.). Based on the abnormality metrics of the training samples, an alerting threshold is chosen such that metric values beyond the threshold imply presence of faults as shown in the illustration below.



As indicated, while a very 'relaxed' threshold will likely fail to detect faulty samples (FDR low, FAR low), a very aggressive threshold will lead to normal samples being flagged as faulty (FAR high, FDR high). This interplay between FDR and FAR for different alerting threshold is graphically presented via a ROC curve shown below. While an ideal fault detection tool will provide a perfect performance (FDR=1, FAR=0), in real scenarios, you will have to tune your alert threshold to provide acceptable levels of FDR and FAR.

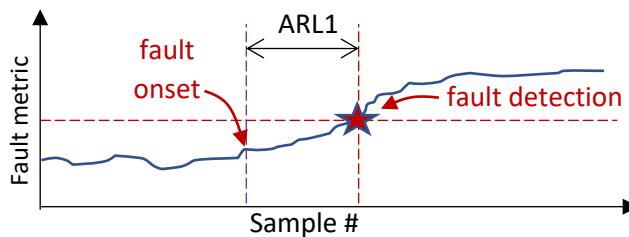


In general, the closer a curve is to the ideal detector or the more lift it has above the diagonal line, the better performance it provides over a random classifier. To compare two models using their ROC curves, a metric called AUC (area under curve) has been devised. AUC is obtained by integrating the ROC curve from FAR=0 to FAR=1. An ideal detector has AUC=1 and higher AUC values imply better fault detection performance.

Fault sensitivity vs fault detection speed



Previously, we focused on a fault detection model's ability to detect a fault. Another important characteristic to assess (fault detection) FD performance is 'speed of detection'. The metric commonly used to quantify the detection speed is called average run length (ARL). Specifically, ARL1 refers to the number of samples (on an average) after the onset of fault that a FD model detects it.



In Chapter 5, we will see the trade-off between fault sensitivity and fault detection speed. Specifically, a technique called CUSUM will be introduced that is able to detect faults of small magnitudes (\Rightarrow high fault sensitivity) but shows higher ARL1 compared to classical 3-sigma technique (which is widely used for univariate signal monitoring). Another metric called ARL0 denotes the average time or number of samples a FD model takes to flag a (false) alarm when no fault exists in the process.

4.5 Model Tuning

Every ML model comes with a set of hyperparameter (e.g., the number of neurons in ANNs, number of retained latent components in PCAs) that can be adjusted to improve the performance metrics. However, care must be taken to avoid overfitting the model. The illustration below shows SVDD modeling of NOC training samples for different values of the bandwidth.

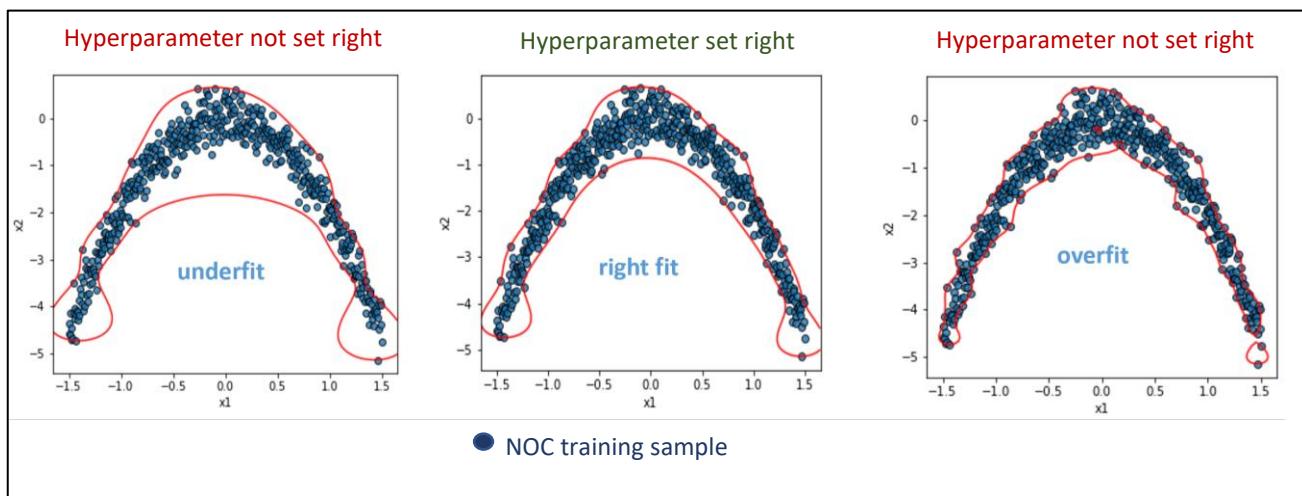


Figure 4.2: SVDD hyperparameter tuning

For high-dimensional complex datasets, it is not easy to judge whether we are overfitting the data while tuning your model. Therefore, the standard approach is to use separate sets of datasets to use for fitting, tuning, and assessing the model as shown in the figure below.

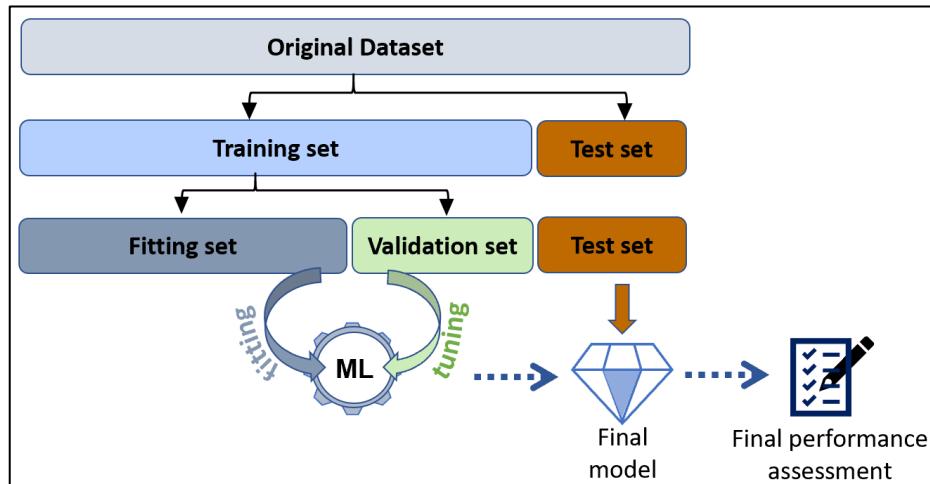


Figure 4.3: 3-way holdout methodology

An overfitted model is an unnecessarily complex model that ends up fitting the noise in the training dataset and therefore, it can be expected to provide poor performance (high FAR) on validation dataset. On the other hand, the underfitted model fails to capture even the systematic structure in the training dataset leading to undesirable low FDR on both fitting and validation datasets. Therefore, the hyperparameter values that lead to the least complex model providing acceptable performance on validation dataset is often chosen as the final model. This practice of using validation dataset to assess model performance during model tuning is termed cross-validation.

Cross-validation with semi-supervised learning



If you had the luxury of having adequate number of faulty samples in your validation dataset, then you could directly use metrics such as FDR or MDR to assess how well your model does on validation dataset. However, as would often be the case, you will work only with NOC samples. In such cases, a common practice is to select hyperparameters such that it gives a certain level of FAR on validation dataset. For example, in Figure 4.2, an SVDD was fitted and FAR computed on validation dataset. The bandwidth value is selected such that it gives an FAR of 5% (and no lower) during cross-validation. A lower validation FAR would indicate potential underfit and higher than 5% FAR would imply overfit.

Summary

In this chapter, we covered some basic but crucial aspects of building a fault detection model that you should pay careful attention to ensure satisfactory model performance. We touched upon the tasks of data selection, data pre-processing, model evaluation, and model tuning. Starting from the next chapter, we will start the deep-dive into the world of developing process monitoring models.

Part 2

Univariate Signal Monitoring

Chapter 5

Control Charts for Statistical Process Control

Before machine learning engulfed the process industry, simple plotting of key plant variables with statistically chosen upper and lower thresholds used to be the norm for detecting process abnormalities. These plots, called control charts, formed the major component of statistical process control or statistical quality control. Although control charts have lost some of their shine due to the advent of advanced multivariate process monitoring tools, they are still widely employed by plant management to monitor crucial KPIs, for example, product quality, process efficiency, etc. Simple concept, easy interpretation, and quick implementation are some of the reasons for their continued popularity.

Shewhart charts (which includes the popular 3-sigma charts) are the earliest, simplest, and most commonly used control charts. These, however, show poor performance for detection of faults that cause small deviations. Therefore, alternatives such as CUSUM charts and EWMA charts have been devised. In this chapter, we will learn these techniques and become familiar with how to implement them in practice. We will conclude with some discussion on the ways to overcome the shortcomings of univariate control charts. Specifically, the following topics are covered.

- Introduction to Shewhart control charts
- Introduction to CUSUM control charts
- Introduction to EWMA control charts
- Statistical process control of aeration tank via CUSUM chart
- Strategies for overcoming limitations of univariate statistical process control

5.1 Control Charts: Simple and Time-tested Process Monitoring Tools

Control charts are one of the seven²⁴ pillars of statistical process control that are traditionally used to monitor key production or product quality metrics in order to detect unexpected deviations. When the process is ‘in-control’, the monitored variables are expected to exhibit only natural cause variations around some target or mean values. As shown in Figure 5.1, a control chart is simply a display of measurements of a single process variable plotted against time or sample number. Additionally, these charts include a centerline or the expected mean value, and a couple of limit lines called UCL (upper control limit) and LCL (lower control limit). Most industrial process variables show natural variations due to random disturbances affecting the process. The control limits are statistically designed in such a way that under natural cause variations, the monitored variable remains within the control limits with certain desired probability. The breach of the control limits indicates potential process fault or an ‘out-of-control’ situation. Proper specification of the limits is therefore essential to ensure minimal false alarms and rapid detection of faults.

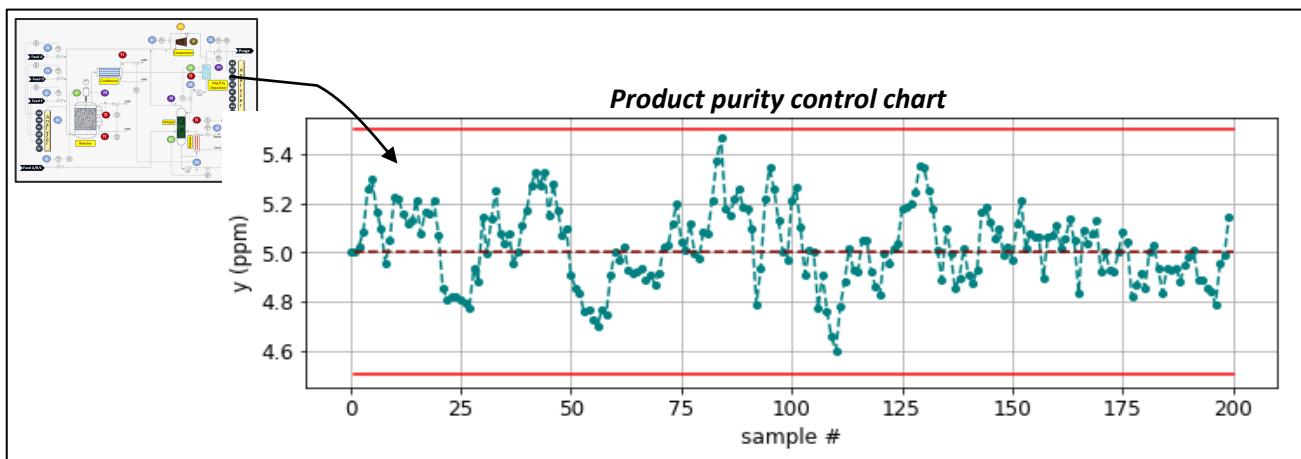


Figure 5.1: Representative control chart for product purity

The traditional control charts take three different forms, viz, Shewhart charts, CUSUM charts, and EWMA charts. While a Shewhart chart plots only the current measurements on the control chart, the other two plot some combination of current and past measurements. You will soon learn how usage of past measurements allow detection of incipient or low magnitude faults which may not get detected by Shewhart charts. Control charts are not limited to tracking process measurements only; any metric that is expected to exhibit only random fluctuations around some mean or target can be monitored via control charts. Correspondingly, control charts are also employed for monitoring model residuals, latent variables (for example, in PCA), etc. Let’s first get started with Shewhart charts.

²⁴ <https://asq.org/quality-resources/statistical-process-control>

5.2 Shewhart Charts: An Introduction

Shewhart chart is the simplest and widely used control chart wherein the process measurement at each sampling instant is plotted against the sample number and the control limits are often set at $\mu \pm 3\sigma$, where μ and σ are the in-control process variable mean and standard deviation. The centerline is obviously set at μ . Shewhart charts assume Gaussian distribution and therefore, the 3σ control chart amounts to a false alarm rate of 0.27% as shown in Figure 5.2

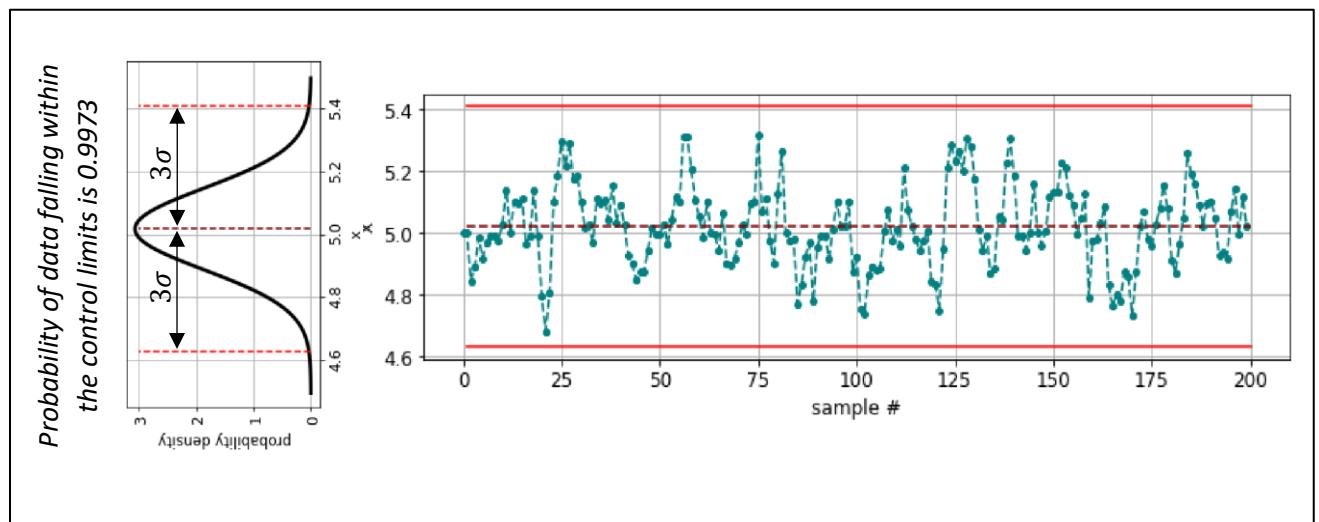


Figure 5.2: In-control measurements on a Shewhart chart

The statistics μ and σ are often not known exactly and are estimated from historical in-control data. Let x_1, x_2, \dots, x_N be the observations from historical good operation period, then the sample statistics can be estimated as²⁵

$$\hat{\mu} = \bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \quad ; \quad \hat{\sigma} = \frac{s}{c_4}$$

where s is sample standard deviation ($\sqrt{\frac{\sum(x_i - \bar{x})^2}{N-1}}$) and c_4 is a correction factor²⁶ (which tends to 1 for large N) to account for finite samples. This leads to the following control limits

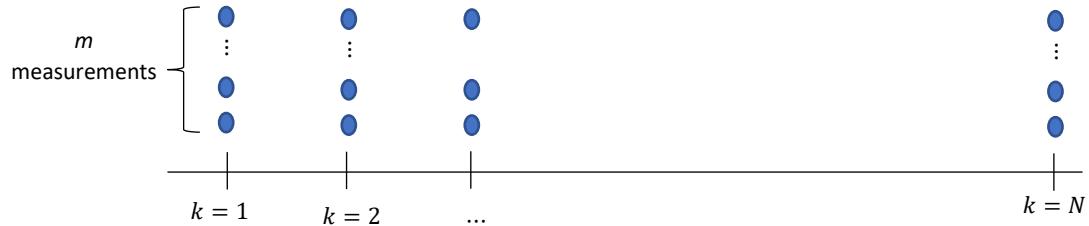
$$LCL = \bar{x} - 3 \frac{s}{c_4} \quad ; \quad UCL = \bar{x} + 3 \frac{s}{c_4}$$

²⁵ <https://www.itl.nist.gov/div898/handbook/pmc/section3/pmc322.htm>

²⁶ <https://www.itl.nist.gov/div898/handbook/pmc/section3/pmc32.htm>

Subgroup of measurements at each sample

The standard Shewhart chart assumes that a subgroup of, say, m measurements is available at each sampling instant as illustrated below.



It is the mean of the subgroup that is tracked on the standard Shewhart chart. Infact, the Shewhart chart with $m=1$ is referred to as an individual Shewhart chart. Let x_{ij} denote the j^{th} measurement in the i^{th} subgroup. The sample statistics for the subgroup mean can be obtained as follows using historical IC (in-control) data.

$$\hat{\mu}_{\bar{x}} = \bar{\bar{x}} = \frac{1}{N} \sum_{i=1}^N \bar{x}_i$$

$\bar{x}_i = \frac{1}{m} \sum_{j=1}^m x_{ij}$

i^{th} subgroup mean

$$\hat{\sigma}_{\bar{x}} = \frac{1}{c_4} \frac{\bar{S}}{\sqrt{m}} = \frac{1}{c_4} \frac{1}{\sqrt{m}} \frac{\sum_{i=1}^N S_i}{N}$$

$S_i = \sqrt{\frac{1}{m-1} \sum_{j=1}^m (x_{ij} - \bar{x}_i)^2}$

i^{th} subgroup standard deviation

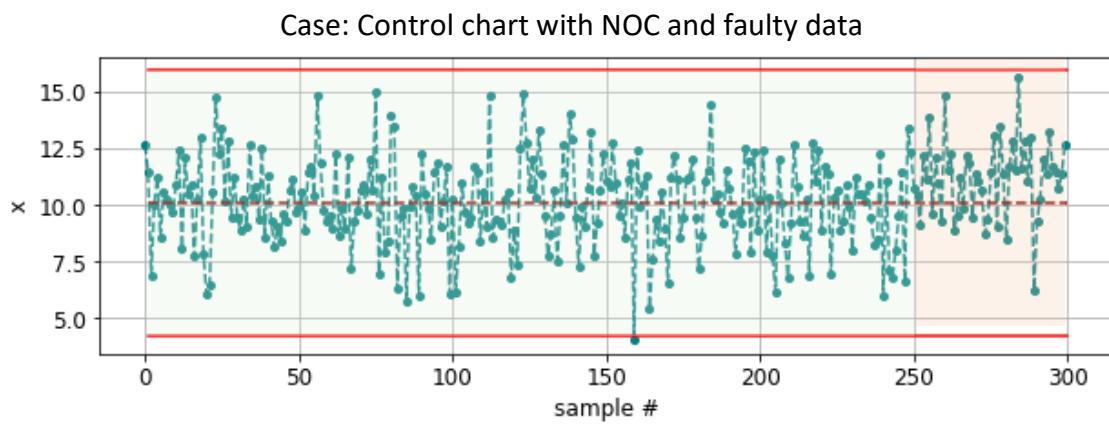
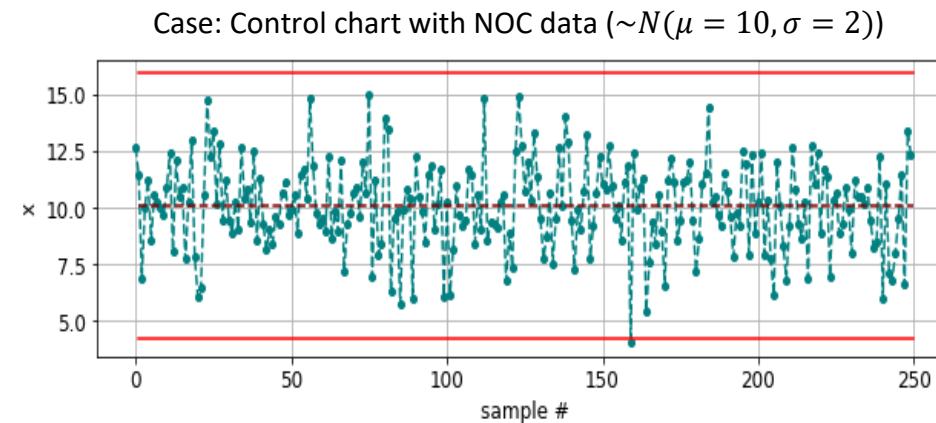
▼▼▼

$$LCL = \bar{\bar{x}} - 3 \frac{1}{c_4} \frac{\bar{S}}{\sqrt{m}}$$

$$UCL = \bar{\bar{x}} + 3 \frac{1}{c_4} \frac{\bar{S}}{\sqrt{m}}$$

For continuous processes, both individual and standard Shewhart charts find utility. For example, suppose that plant efficiency measurements are available every minute. Plant operators, if desired, can track the minute-level efficiencies using individual control chart. Alternatively, efficiency measurements may be averaged over an hour ($\Rightarrow m=60$) and the hourly-averaged efficiency values may be tracked

While Shewhart charts are easy to interpret and implement, their performance is poor when used to detect small deviations ($< 1.5\sigma$ shifts) in process variable mean. The following illustration clarifies this aspect.



- Last 50 samples generated with 0.5σ shift in mean
- Fault could not get detected by Shewhart chart
- Even visually, the fault is barely perceptible

Figure 5.3: Shewhart chart's failure to detect small mean shifts

To understand how Shewhart plots can be generated, let us work through the code used to generate the plots in Figure 5.3.

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

# generate data
x0 = np.random.normal(loc=10, scale=2, size=250) # NOC samples
x1 = np.random.normal(loc=11, scale=2, size=50) # faulty samples
x = np.hstack((x0,x1)) # combined data
```

```
# fit Shewhart model using NOC samples
mu, sigma = np.mean(x0), np.std(x0)
UCL, LCL = mu + 3*sigma, mu - 3*sigma

# plot control chart for combined data
plt.plot(x,'--',marker='o', color='teal')
plt.plot([1,len(x)],[UCL,UCL], color='red'), plt.plot([1,len(x)],[LCL,LCL], color='red')
plt.plot([1,len(x)],[mu, mu], '--', color='maroon')
plt.xlabel('sample #'), plt.ylabel('x')
```

5.3 CUSUM Charts: An Introduction

Consider the control chart shown below for the dataset shown in Figure 5.3. It is apparent that unlike the Shewhart chart, this control chart easily flags the out-of-control situation. This chart is called CUSUM chart and is known to be more effective at detecting small mean shifts. At the i^{th} sampling instant, the CUSUM statistic (S_i) that is plotted on the CUSUM chart is the ‘Cumulative SUM’ of deviations of each of the available measurements from mean (or target). When a process is in control, the random deviations around the mean cancel out and S_i wanders around zero; however, in presence of a mean shift of, say, Δ at r^{th} sampling instant, an additional Δ term gets added to S_i at every instant $\geq r$ which leads to a continuous upward (or downward) trend as seen in Figure 5.4.

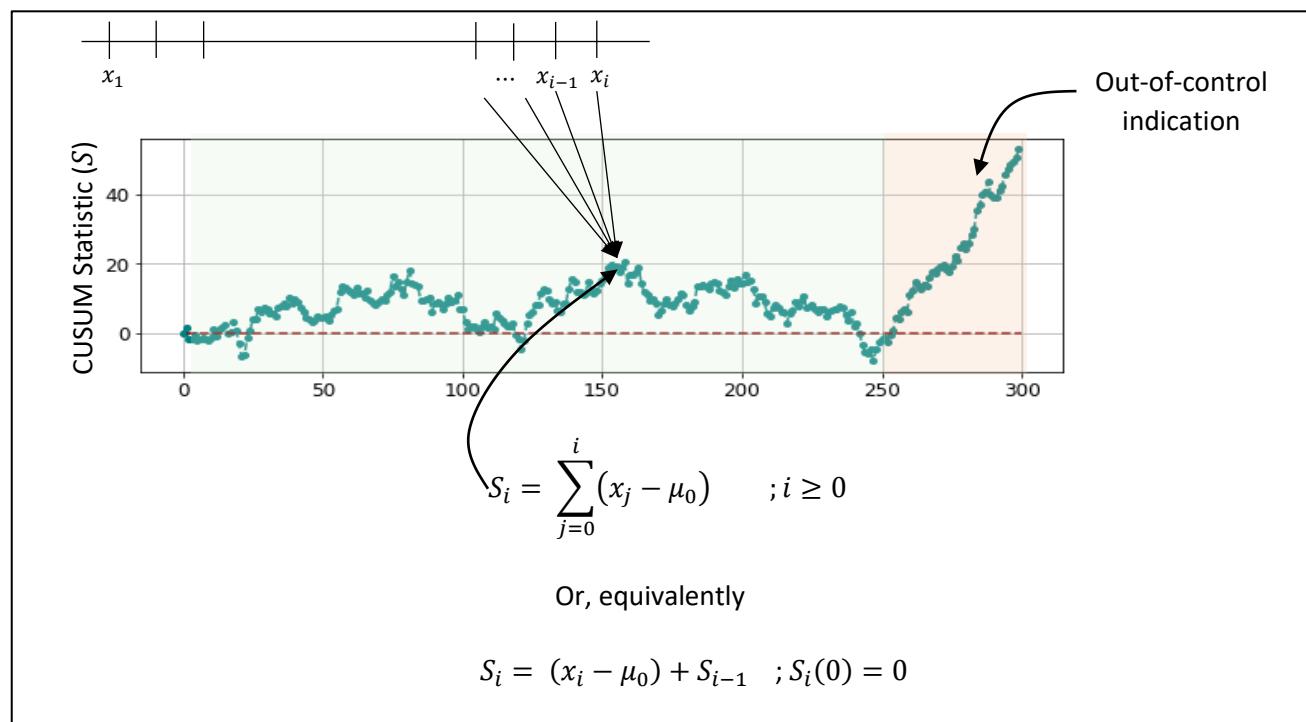


Figure 5.4: CUSUM control chart

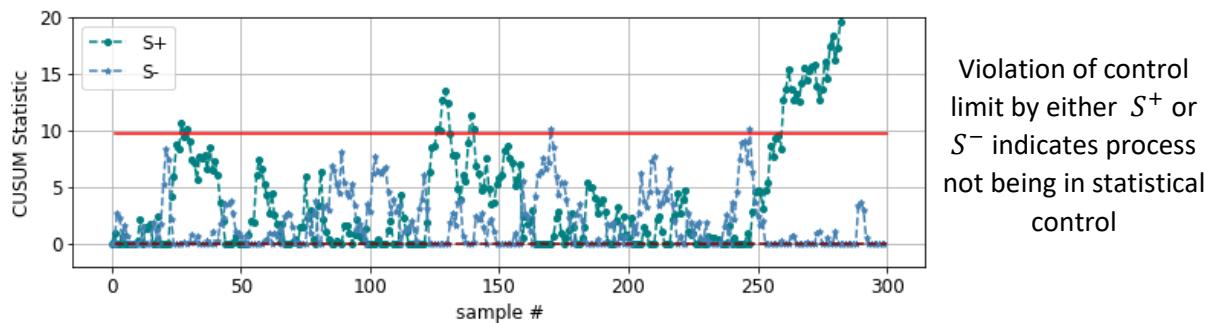
Fault detection using CUSUM charts used to be done graphically by looking at the slope of the trends using the so-called V-masks. However, in modern statistical software systems, the following two-sided CUSUM charts are utilized.

$$S_i^+ = \max[0, x_i - (\mu_0 + k) + S_{i-1}^+] \rightarrow \text{tracks positive shifts in mean}$$

$$S_i^- = \max[0, (\mu_0 - k) - x_i + S_{i-1}^-] \rightarrow \text{tracks negative shifts in mean}$$

- $k \geq 0$
- Deviation from mean (or target) greater than k increases S^+ or S^-
- k is usually set to be $\frac{1}{2}$ the size of mean shift that we want to detect quickly

Both S^+ and S^- are plotted together on the same chart along with a specified control limit (or threshold) H as shown below.



The implementation of CUSUM chart requires specification of k , H , and mean μ_0 (or target T). While μ_0 is simply the sample mean of historical IC data, determination of control limit H is slightly more involved compared to Shewhart charts. H is often selected based on the ARL(0) and ARL(1) values for different levels of mean shifts²⁷. Usually, H is set at $4\hat{\sigma}$ or $5\hat{\sigma}$ where $\hat{\sigma}$ is estimated the same way as done for Shewhart charts.

Change Point Detection



In the area of anomaly detection, you may encounter the term ‘change point detection’ (CPD). CPD is the problem of finding when significant changes in properties (mean, standard deviation, etc.) of a signal occurs. CUSUM and EWMA are two popular techniques employed for CPD.

²⁷ <https://www.itl.nist.gov/div898/handbook/pmc/section3/pmc3231.htm>

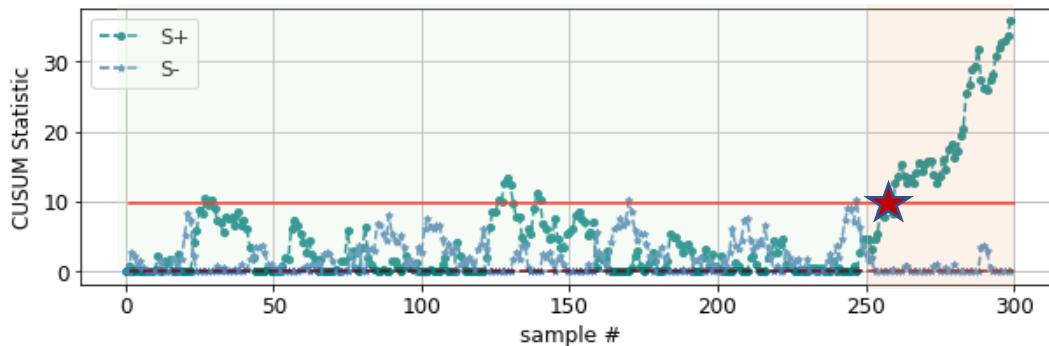
Example 5.1: We will take the data from Figure 5.3 and learn how to build CUSUM control charts. We will also see the impact of k on chart performance.

```
# CUSUM chart
mu, sigma = np.mean(x0), np.std(x0)
k, H = 0.25*sigma, 5*sigma

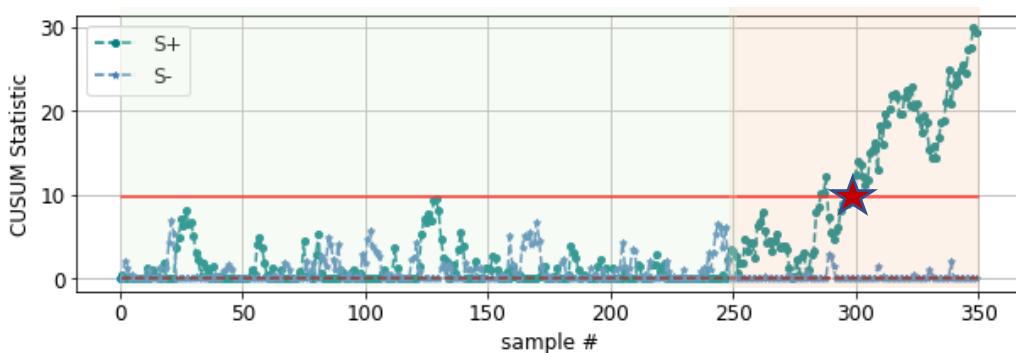
S_positive, S_negative = np.zeros((len(x),)), np.zeros((len(x),))
S_positive[0], S_negative[0] = 0, 0

for i in range(1,len(x)):
    S_positive[i] = np.max([0, x[i]-(mu+k) + S_positive[i-1]])
    S_negative[i] = np.max([0, (mu-k)-x[i] + S_negative[i-1]])

plt.plot(S_positive,'--',marker='o', color='teal', label='S+')
plt.plot(S_negative,'--',marker='*', color='steelblue', label='S-')
plt.plot([1,len(x)],[H,H], color='red'), plt.plot([1,len(x)],[0,0], '--', color='maroon')
```



The CUSUM chart can promptly detect the shift in mean. However, we also see some false alerts in NOC data. This is because our chart is very sensitive (designed to capture just 0.5σ shifts). The figure below shows the impact of increasing k to 0.5σ – the incidence of false alerts has decreased, but there is considerable delay in fault detection (~ 50 samples).



CUSUM charts can also detect large shifts in process mean; however, the detection is slower (i.e., the ARL(1) is higher) than Shewhart charts. EWMA provides a good compromise between detection of small mean shifts and quick detection of large shifts.

5.4 EWMA Charts: An Introduction

Like CUSUM, EWMA charts are control charts with memory. Here, at any i^{th} sampling instant, the statistic (z_i) plotted on the EWMA control chart is a weighted combination of the i^{th} process measurement and the previous statistic z_{i-1} ; this translates to EWMA statistic being an exponentially weighted average of all the past measurements as shown in Figure 5.5. We can see that EWMA chart provides a balance between Shewhart (only current observation is considered) and CUSUM (equal weightage given to all available observations) charts; accordingly, this leads to a good balance between detection delay and fault sensitivity.

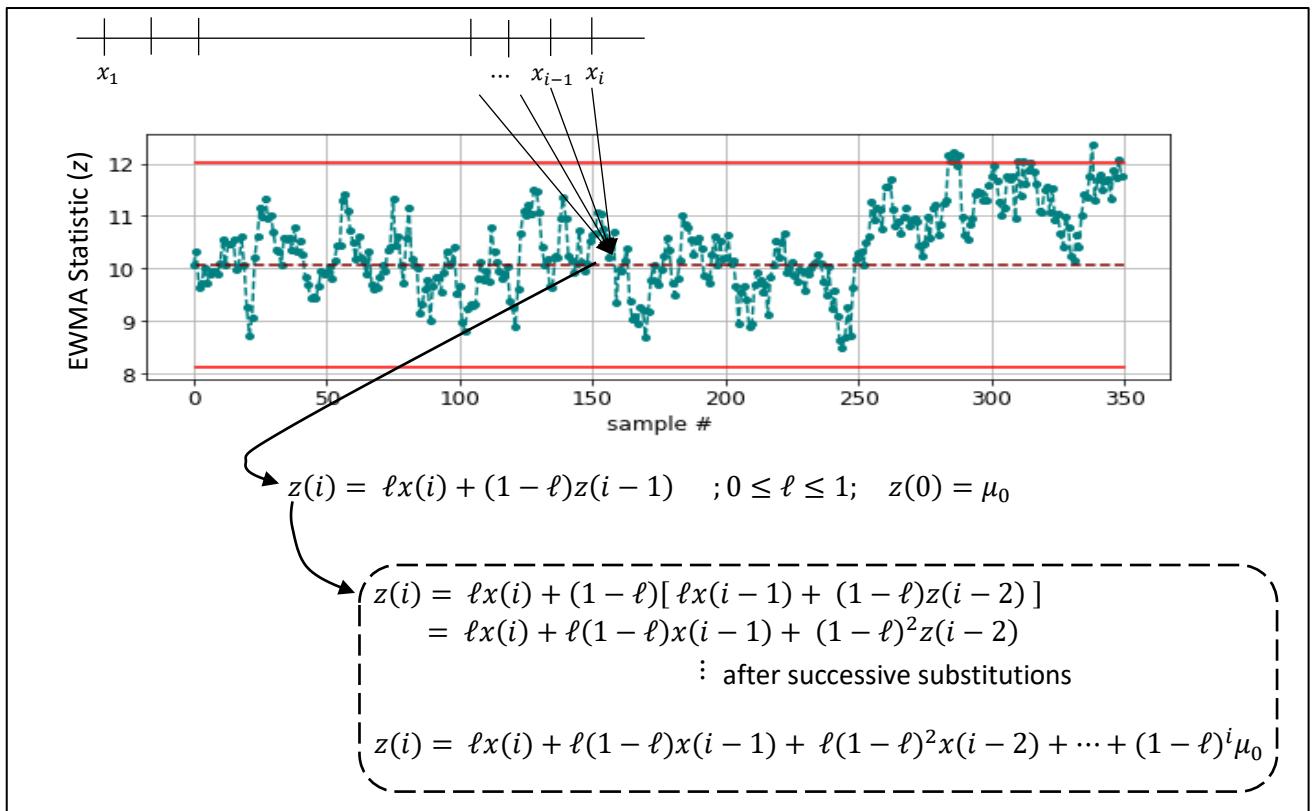


Figure 5.5: EWMA control chart

The hyperparameter λ determines the weightage given to past observations. For $\lambda = 1$, EWMA chart reduces to Shewhart chart and as $\lambda \rightarrow 0$, CUSUM-like properties begin to emerge. A usual recommendation is to have $0.2 \leq \lambda \leq 0.3$. The 3-sigma control limits are given as follows.

$$LCL = \hat{\mu} - 3\hat{\sigma}\sqrt{\frac{\lambda}{2-\lambda}} \quad ; \quad UCL = \hat{\mu} + 3\hat{\sigma}\sqrt{\frac{\lambda}{2-\lambda}}$$

where $\hat{\sigma}$ is the historical IC standard deviation as calculated for Shewhart charts.

Example 5.1 continued: We will again take the data from Figure 5.3 and learn how to build EWMA control charts.

EWMA chart

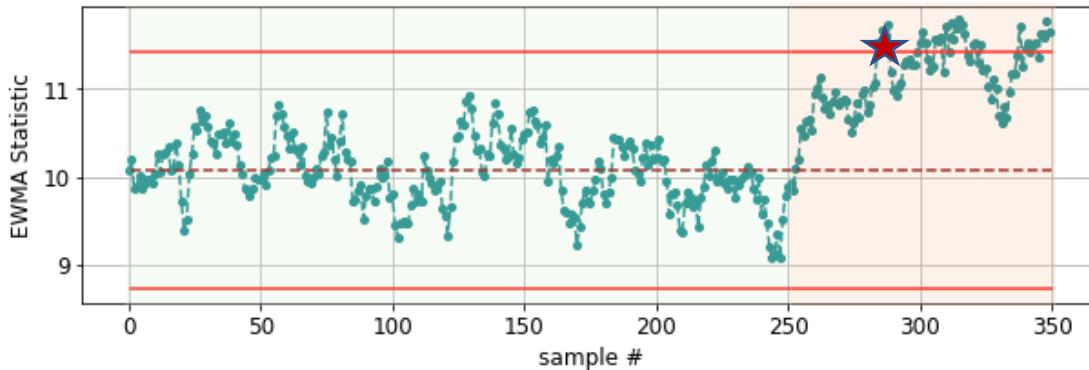
```
mu, sigma = np.mean(x0), np.std(x0)
smoothFactor = 0.1
```

```
LCL = mu - 3*sigma*np.sqrt(smoothFactor/(2-smoothFactor))
UCL = mu + 3*sigma*np.sqrt(smoothFactor/(2-smoothFactor))
```

```
z = np.zeros((len(x),))
z[0] = mu
```

```
for i in range(1,len(x)):
    z[i] = smoothFactor*x[i] + (1-smoothFactor)*z[i-1]
```

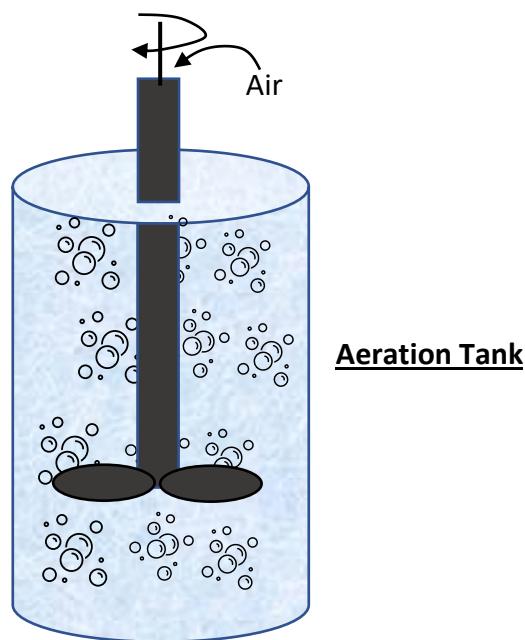
```
plt.plot(z,'--',marker='o', color='teal')
plt.plot([1,len(x)],[LCL,LCL], color='red'), plt.plot([1,len(x)],[UCL,UCL], color='red')
plt.plot([1,len(x)],[mu,mu], '--', color='maroon')
plt.xlabel('sample #')
```



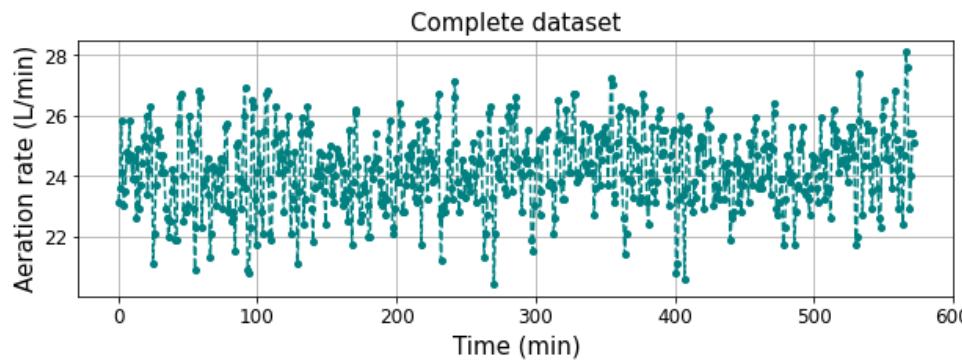
The balance between the performances of Shewhart and CUSUM chart is immediately apparent in the above EWMA chart. The EWMA statistic breaches (although barely) the alert threshold (\Rightarrow better performance than Shewhart chart) and detection delay is lower than that of CUSUM as well (\Rightarrow better performance than CUSUM chart).

5.5 Case Study: Monitoring Air Flow in an Aeration Tank

To illustrate an industrial application of control charts, we will consider data²⁸ from an aeration tank shown below. An aeration tank uses small air bubbles to keep solid particles in suspension. Excessive foaming and loss of valuable solid product occurs if too much air is blown into the tank. If too little air is blown into the tank, the particles sink and drop out of suspension. The simulated dataset contains 573 observations, where each observation equals the total liters of air added to the tank in a one-minute interval.

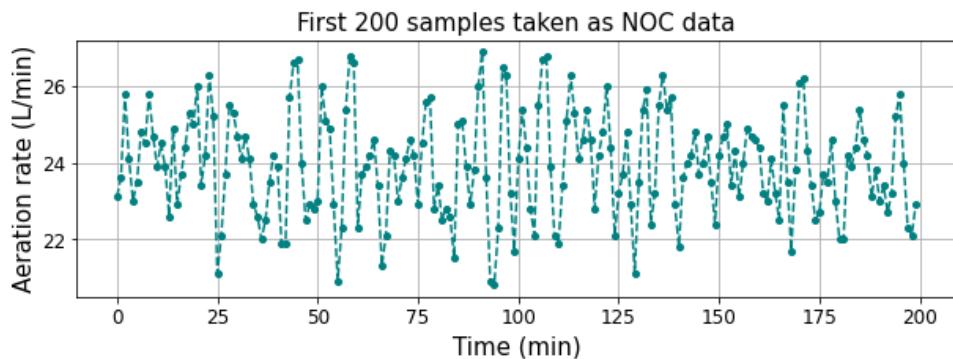


All 573 observations are plotted below. You won't be judged if you fail to notice a slight upward shift around 300 minutes in the first glance.

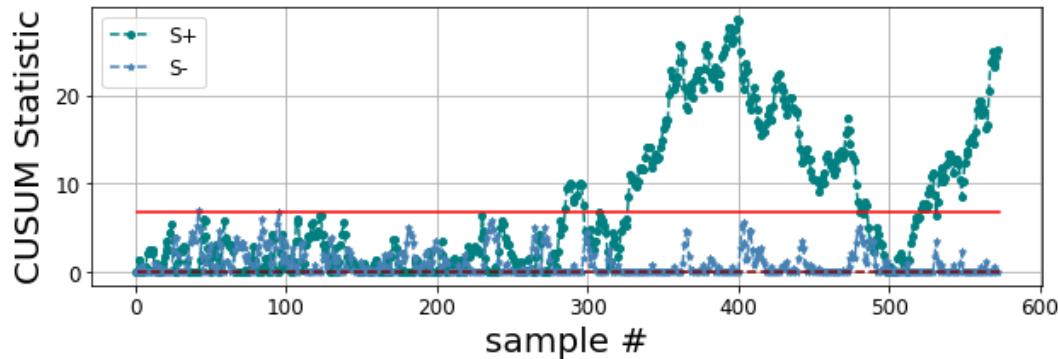


²⁸ The *aeration rate dataset* and description is publicly available at <https://openmv.net/info/aeration-rate>.

However, we have learnt that CUSUM charts excel in capturing small mean shifts. Let's see how nicely it works for this dataset. We will take first 200 observations as NOC data which is plotted below. The mean and standard deviation is computed from these samples.



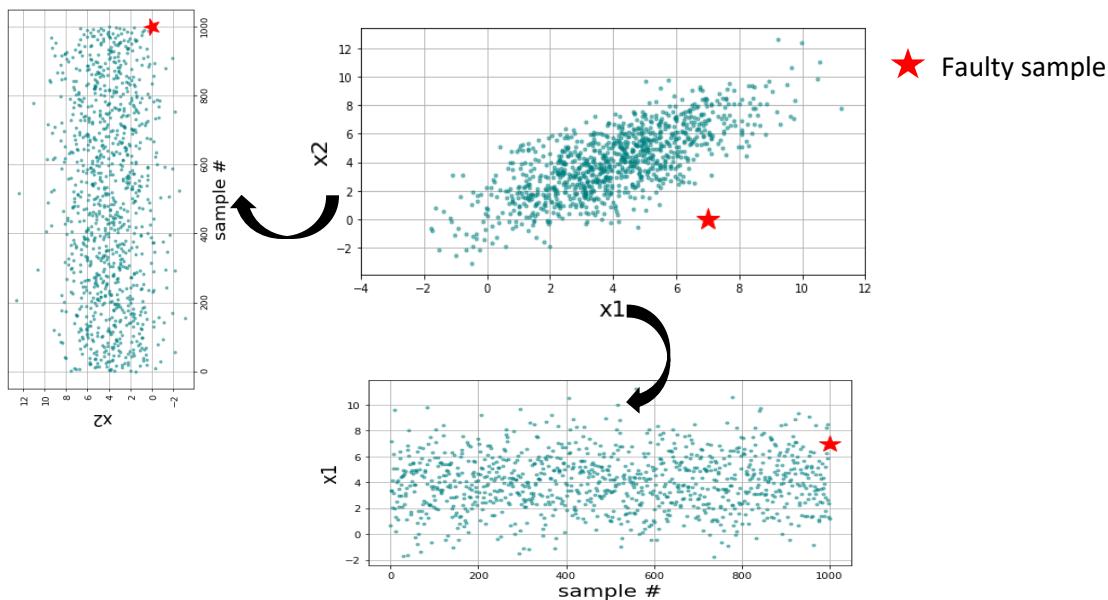
To generate the CUSUM chart, k is taken to be equal to 0.25σ . Using the code shown in Example 5.1, the CUSUM control chart²⁹ shown below is generated. It is impressive that, without much false alarms, the small mean shift has now been made very 'obvious' to see!



²⁹ The complete code is provided in the online GitHub repository.

5.6 Pitfalls of Univariate Control Charts and Alternative Solutions

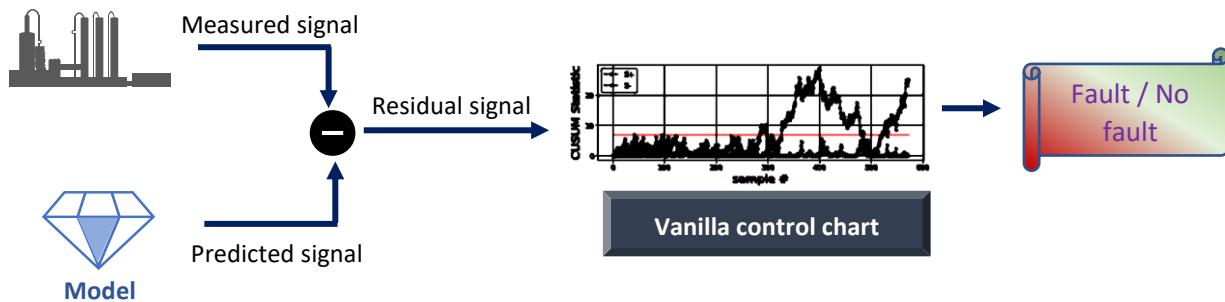
The obvious shortcoming of the techniques learnt in this chapter is their inability to effectively detect faults in multivariate systems as illustrated below. The faulty sample is an obvious outlier in the 2D space; however, if the variables x_1 and x_2 are analyzed individually, the faulty sample appears within the NOC ranges.



Multivariate Shewhart chart (Hotelling T^2 chart), multivariate CUSUM (MCUSUM), and multivariate EWMA (MEWMA) schemes have been devised, but they do not enjoy the wide acceptance that their univariate counterparts enjoy. This is due to the difficulty encountered while handling correlated high dimensional data. Popular alternatives for handling multivariate data include PCA, kernel PCA, PLS, etc. which you will study in the next part of the book.

Another shortcoming of the vanilla control charts that we have studied is the assumption of independence among the data samples. If you are dealing with a signal that shows autocorrelation then a better strategy as illustrated below is to build a model (e.g., a time-series model³⁰) and compute residuals (between observed and predicted values). For NOC data and carefully built model, the residuals will be uncorrelated and have Gaussian distribution, and therefore, the vanilla control charts can be built for the residuals to indirectly monitor the original signal.

³⁰ Time series modeling techniques are covered in detail in Book 2 of the series.



Another popular strategy to deal with autocorrelated signals is to update the model using the latest observations and monitor the model parameters.

Summary

In this chapter, we familiarized ourselves with statistical process control charts for monitoring univariate signals. We looked at the three popular techniques, viz, Shewhart charts, CUSUM charts, and EWMA charts. We learnt how to build these charts and understood the pros and cons of these different methods. In the next chapter, we will continue our study of univariate signal monitoring and look at pattern matching-based methodologies.

Chapter 6

Process Fault Detection via Time Series Pattern Matching

Imagine you are a plant operator newly put in charge of running a plant and you observe an interesting pattern in one of the process variable: occasional spiky fluctuations without the signal violating the DCS alarm limits. A natural line of investigation would be to find if such patterns have occurred in the past and are a leading indicators of underlying process faults. However, how do you quickly sift through years of historical data to find similar patterns? Consider another scenario where you are responsible for quality control of a batch process. To check if the latest batch went smoothly, you may want to compare it with known reference/golden batch. However, batches may show normal variations due to different batch durations or abnormal deviations due to process fault. How do you train an algorithm to smartly call out a faulty batch? One thing common in both these scenarios is that we are not looking at abnormality of a single measurement; instead, abnormality of a sequence of successive values (also called collective anomalies) is of interest.

Time series pattern matching is a mature field in the area of time series classification and recent algorithmic advances now allow very fast sequence comparisons to find similar or abnormal patterns in historical data. Unsurprisingly, pattern matching is being offered as prime feature in commercial process data analytic software (such as Aspen's Process Explorer, SEEQ, etc.). In this chapter, we will work through some use-cases of pattern-matching-based process monitoring. Specifically, the following topics are covered

- Introduction to time series anomalies
- Pattern matching-based fault detection: use-cases in process industry
- Fault detection via historical pattern search for steam generator process
- Fault detection via discord discovery

6.1 Time Series Anomalies and Pattern Matching

In anomaly detection literature, anomalies in univariate time series or dynamic signals are categorized into three categories: point anomalies, contextual anomalies, and collective anomalies. Figure 6.1 illustrates these anomalies for a valve (%) opening signal. As depicted in Figure 6.1b, if a single measurement deviates significantly from the rest of the sensor readings, then a point anomaly is said to have occurred. Contextual anomaly occurs when a measurement is not anomalous in an ‘overall sense’ but only in a specific context. For example, in Figure 6.1c, point ‘B’ is abnormal when taken in the context of operation mode 1 only; valve opening goes close to 80% under normal operation but not when the process is in mode 1. The last category of collective anomaly occurs when a group/sequence of successive measurements jointly show abnormal behavior, although the individual measurements may not violate NOC range. While control charts can be built to detect point and contextual anomalies, more specialized approaches are needed to detect collective anomalies. Therefore, this chapter is devoted to study of approaches for collective anomaly detection.

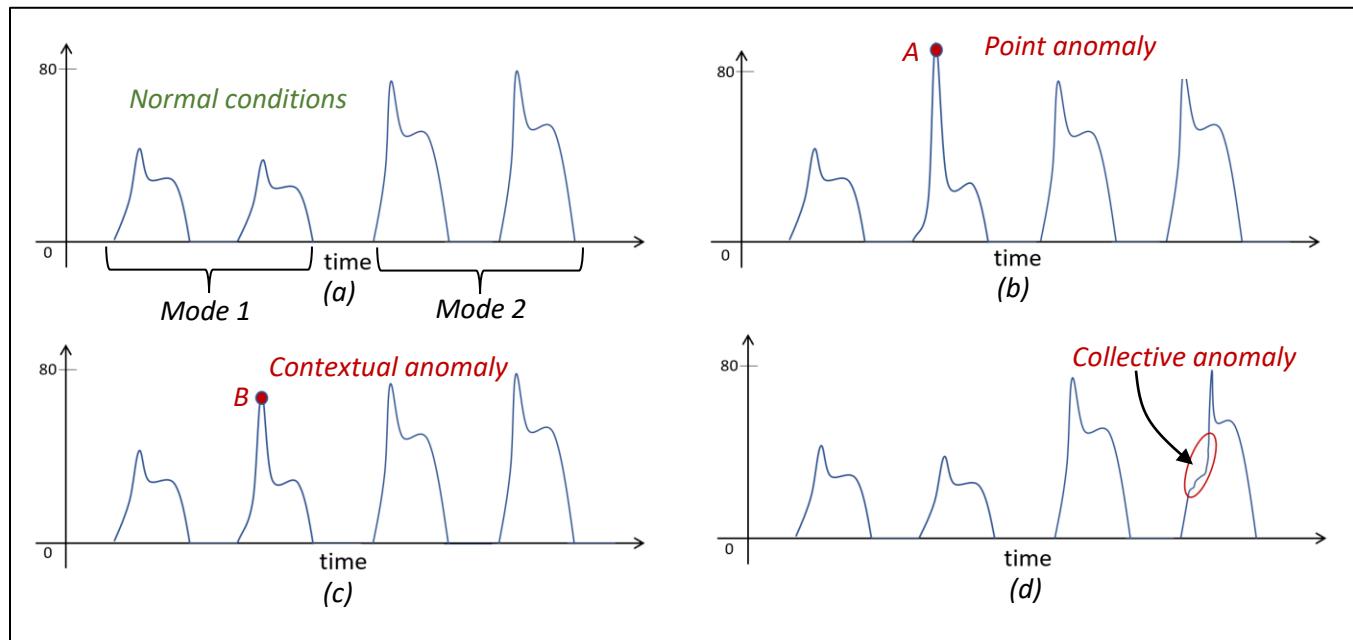


Figure 6.1: Time series anomalies: representative illustrations

The need for (sub) sequence-based pattern matching for FDD show up in different forms in process industry; Figure 6.2 illustrates some of the use-case scenarios. Let’s work through some of these use-cases to understand the underlying techniques and available resources.

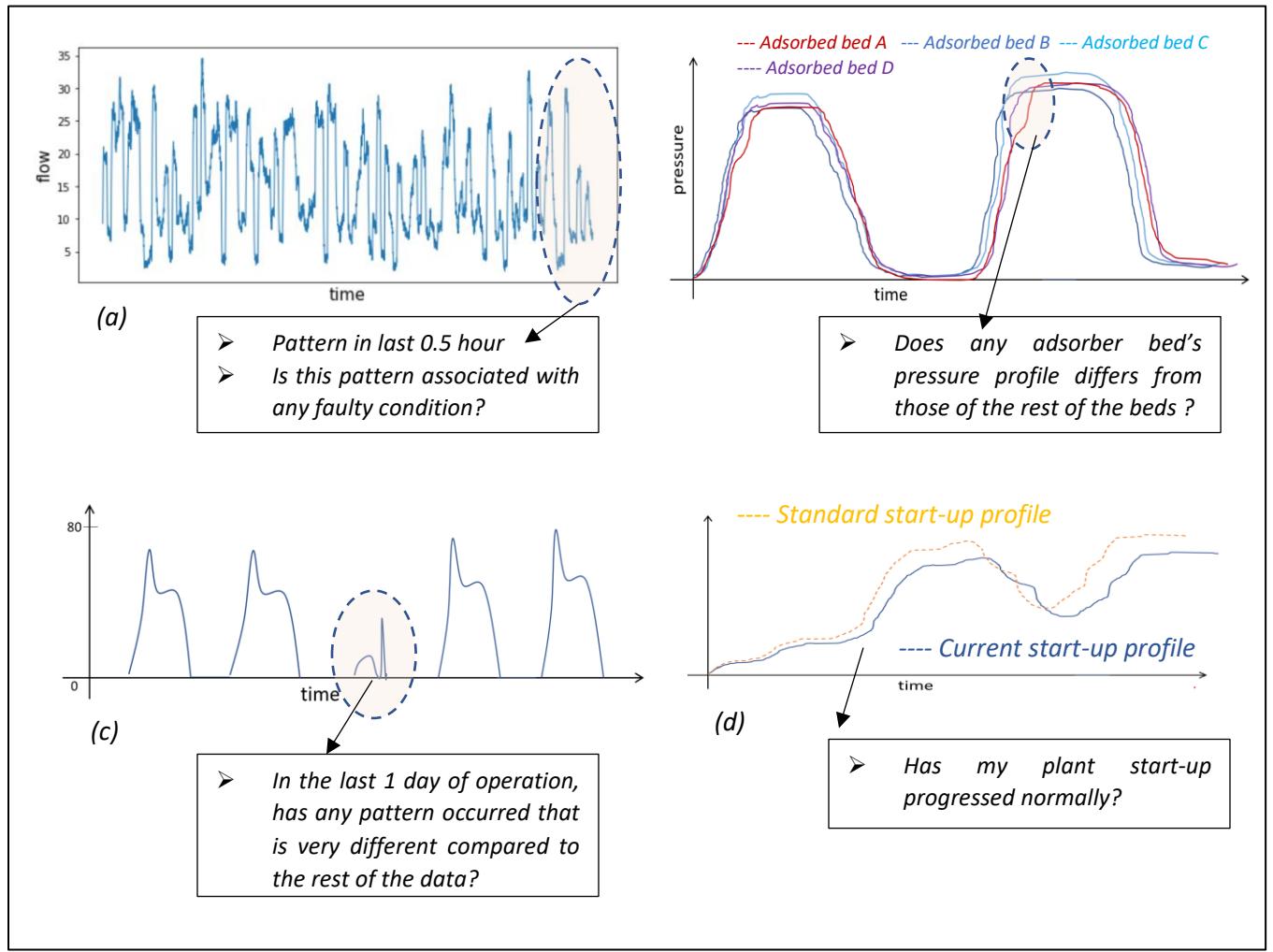


Figure 6.2: Sample use-case scenarios of time series pattern matching-based fault detection

We will work through the case scenarios (a) and (c). Both of these use-cases involve finding similarity of a ‘query’ subsequence with several other subsequences taken from the same time series or another time series. In order to accomplish this in a time-efficient manner, a library called STUMPY³¹ will be utilized. Let’s learn how to utilize STUMPY for our time series data mining tasks.

³¹ <https://stumpy.readthedocs.io/en/latest/index.html>.

S.M. Law, STUMPY: A Powerful and Scalable Python Library for Time Series Data Mining. Journal of Open Source Software, 2019.

6.2 Fault Detection via Historical Pattern Search

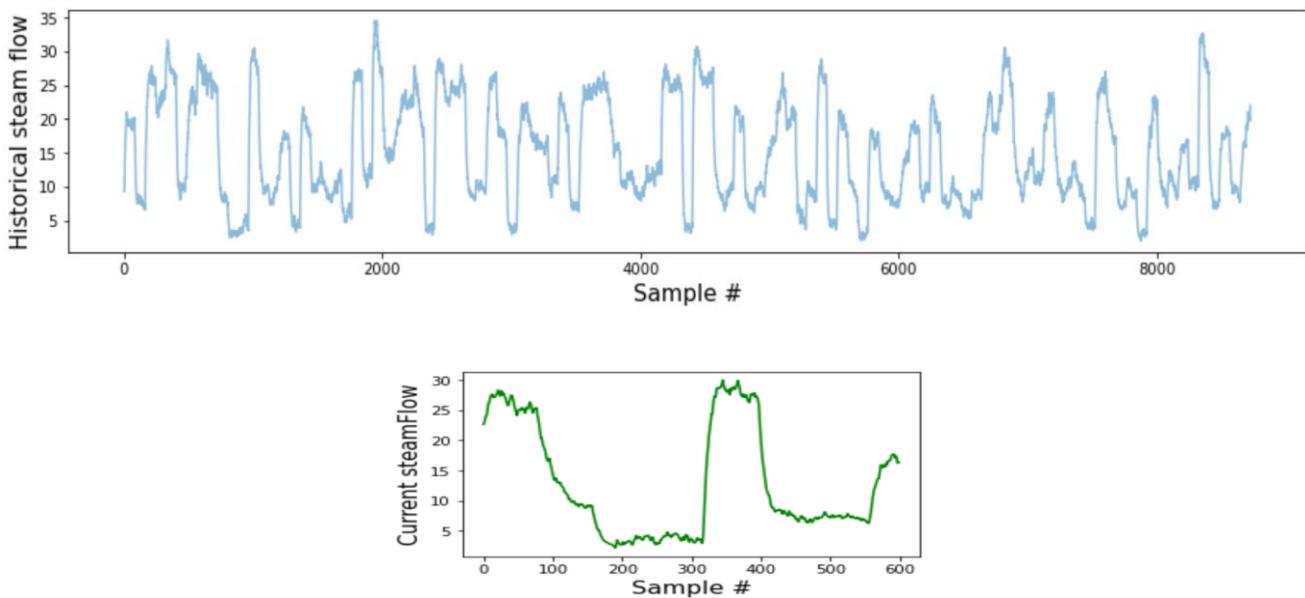
To answer the question posed in Figure 6.2a, the following task can be performed: using the real-time data in the last 30 mins, the most similar subsequence can be found in the historical data; thereafter, one can simply check if a process upset had followed the found subsequence in the history. We will use steam flow data³² obtained from a model of a Steam Generator at Abbott Power Plant in Champaign IL. The provided dataset contains 9600 samples (sampling time 3 seconds) of several process variables including steam flow. We created two separate datafiles, *historical_steamFlow.txt* and *current_steamFlow.txt*, to contain the historical data and the real time data that will be used in this case study. Let's start with loading the data.

```
# import requisite libraries
import stumpy
numpy as np, matplotlib.pyplot as plt

# fetch data
historical_steamFlow = np.loadtxt('historical_steamFlow.txt')
current_steamFlow = np.loadtxt('current_steamFlow.txt')

plt.figure(), plt.plot(historical_steamFlow)
plt.ylabel('Historical steam flow'), plt.xlabel('Sample #')

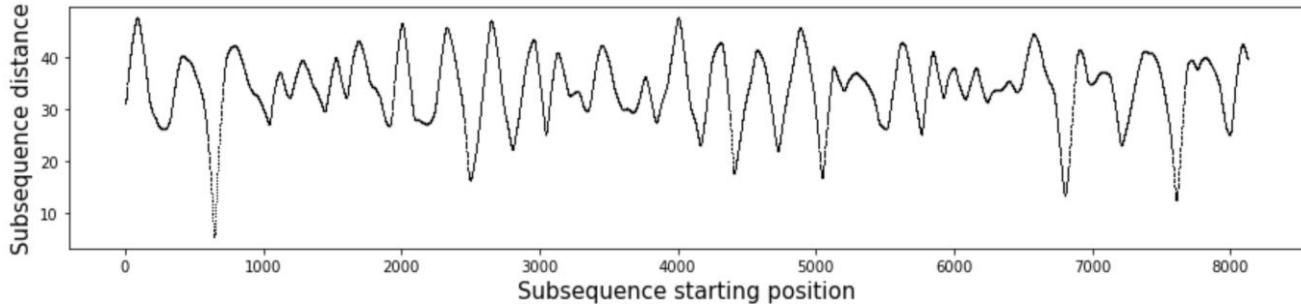
plt.figure(), plt.plot(current_steamFlow, color='green'),
plt.ylabel('Current steamFlow'), plt.xlabel('Sample #')
```



³² De Moor B.L.R. (ed.), DaISy: Database for the Identification of Systems, Department of Electrical Engineering, ESAT/STADIUS, KU Leuven, Belgium, URL: <http://homes.esat.kuleuven.be/~smc/daisy/>

```
# use stumpy library
distance_profile = stumpy.mass(current_steamFlow, historical_steamFlow)

plt.figure(), plt.plot(distance_profile, '*', color='black')
plt.ylabel('Subsequence distance'), plt.xlabel('Subsequence starting position')
```



`distance_profile` contains pairwise distances between `current_steamFlow` and every subsequence (of the same length as `current_steamFlow`) within `historical_steamFlow`. For example, `distance_profile[0]` gives the Euclidean distance between `current_steamFlow` and a subsequence of length `len(current_steamFlow)` starting at position 0 in the `historical_steamFlow` time series. We can notice in the above plot that a subsequence (with start position between 0 and 1000) exists that is at a much less distance from `current_steamFlow` than any other subsequence in `historical_steamFlow`. This most similar subsequence from `historical_steamFlow` can be obtained by finding the smallest distance in `distance_profile` and extracting its positional index as shown below.

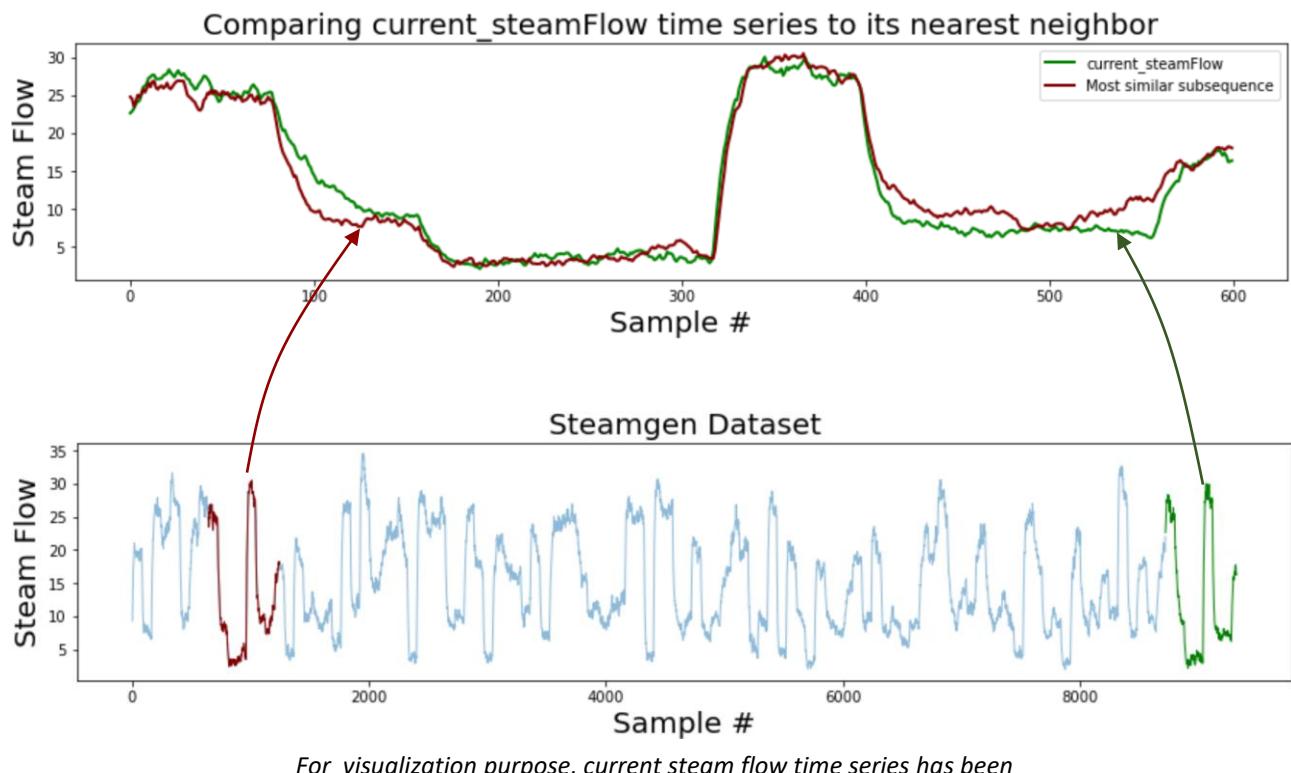
```
# find the most similar historical subsequence
position = np.argmin(distance_profile)
print('The subsequence most similar to current_steamFlow is located at position :', position)

>>> The subsequence most similar to current_steamFlow is located at position : 643
```

Let's see how the query and target subsequence compare to each other.

```
# compare the found pattern to current_steamFlow
historicalPattern = historical_steamFlow[position:position+len(current_steamFlow)]

plt.figure()
plt.plot(current_steamFlow, color="green", label="current_steamFlow")
plt.plot(historicalPattern, color="maroon", label="Most similar subsequence")
plt.xlabel('Sample #'), plt.ylabel('Steam Flow'), plt.legend()
```

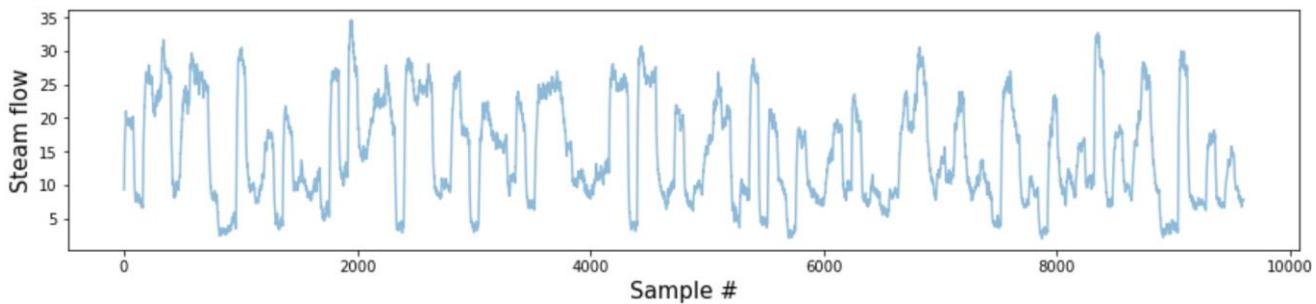


6.3 Fault Detection via Discord Discovery

The question posed in Figure 6.2c can be solved by finding a subsequence which is at a large distance from any other subsequence in a given time series. For this case-study, we will use the steam flow time series provided in the original steam generator dataset. Let's start with loading the data.

```
# import requisite libraries
import stumpy, numpy as np, matplotlib.pyplot as plt

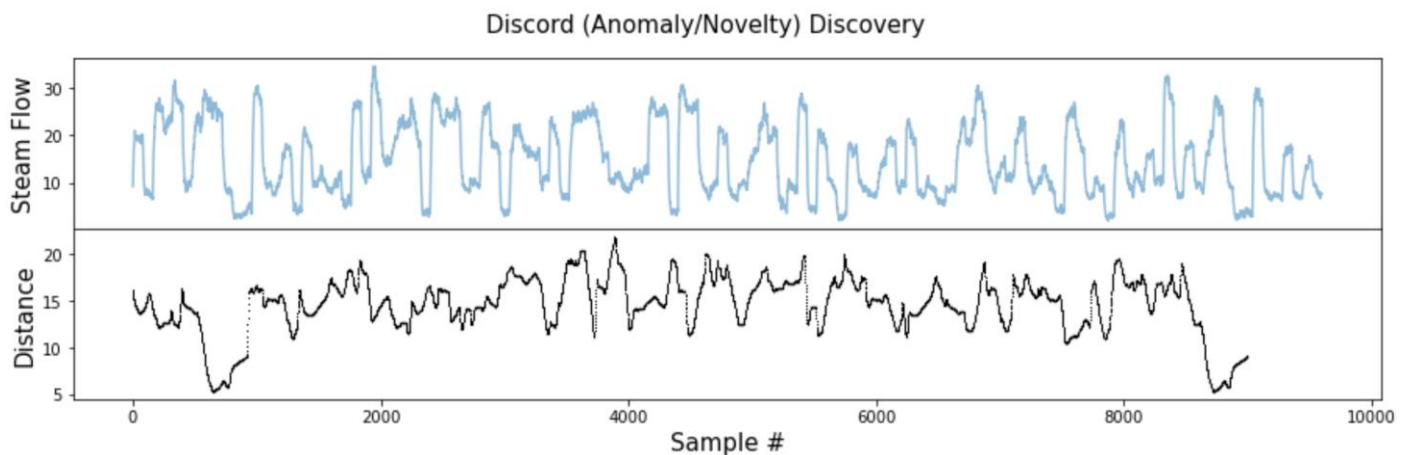
# fetch data
data = np.loadtxt(steamgen.dat')
steamFlow = data[:,8]
plt.figure(), plt.plot(steamFlow)
plt.ylabel('Steam flow'), plt.xlabel('Sample #')
```



```
# use stumpy library
sequenceLength = 600 # 30 mins
matrix_profile = stumpy.stump(steamFlow, sequenceLength)
```

The 'stump' function does the following: for every subsequence (of length `sequenceLength`) within the `steamFlow` time series, it automatically identifies its corresponding nearest-neighbor subsequence of the same length³³. The first column of `matrix_profile` array contains the distance values and the nearest neighbor subsequence's indices are in the 2nd and 3rd columns. Let's see how the distances vary.

```
# plot matrix profile distances
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})
plt.suptitle('Discord (Anomaly/Novelty) Discovery')
axs[0].plot(steamFlow), axs[0].set_ylabel('Steam Flow')
axs[1].plot(matrix_profile[:, 0], '*', color='black')
axs[1].set_xlabel('Sample #'), axs[1].set_ylabel('Distance')
```



To find the most unusual subsequence which can be a potential anomaly, we just need to find the subsequence position in the `steamFlow` time series that has the largest value from its nearest neighbor subsequence.

³³ Check out the nice visualization at <https://stumpy.readthedocs.io/en/latest/index.html>.

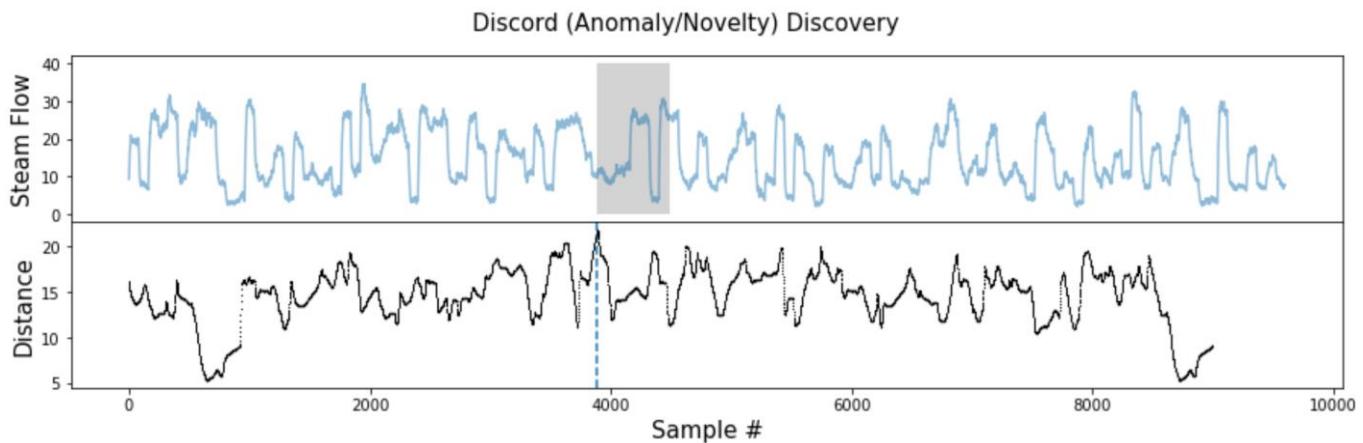
```
# find discord
discord_position = np.argsort(matrix_profile[:, 0])[-1]
print('The discord is located at position: ', discord_position)

>>> The discord is located at position: 3890

# show the discord in the time series
from matplotlib.patches import Rectangle
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})
plt.suptitle('Discord (Anomaly/Novelty) Discovery')

axs[0].plot(steamFlow), axs[0].set_ylabel('Steam Flow')
rect = Rectangle((discord_position, 0), sequenceLength, 40, facecolor='lightgrey')
axs[0].add_patch(rect)

axs[1].plot(matrix_profile[:, 0], '*', color='black')
axs[1].set_xlabel('Sample #'), axs[1].set_ylabel('Distance')
axs[1].axvline(x= discord_position, linestyle="dashed")
```



We won't be surprised if you are already thinking of the different ways you could use time series pattern matching for your own process systems. Libraries like STUMPY have made finding patterns in huge volumes of data easy and therefore, we encourage you to let your imaginations run wild with the potential use cases!

Summary

In this chapter, we looked at time series data mining techniques for fault detection in process data. Specifically, we worked with a steam generator dataset, and looked at pattern matching and discord discovery problems. We learnt how to use a powerful library called STUMPY to solve these problems.

Part 3

Multivariate Statistical Process Monitoring

Chapter 7

Multivariate Statistical Process Monitoring for Linear and Steady-State Processes: Part 1

It is not uncommon to have hundreds of process relevant variables being measured at manufacturing facilities. However, 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. Several smart techniques have been derived to find these hidden latent variables. Latent variable-based techniques allow characterization of ‘normal’ process noise affecting the process during NOC. Process monitoring methods based on latent space monitor the values of latent variables and process noise in real-time to infer the presence of process faults. Sounds complicated? Don’t worry! This chapter will show you how this is accomplished while retaining focus on conceptual understanding and practical implementation.

PCA and PLS are among the most popular latent variable-based process monitoring tools and have been reported in several successful industrial process monitoring applications. This chapter provides a comprehensive exposition of the PCA and PLS techniques and teaches you how to apply them for fault detection. Furthermore, we will learn how to identify the faulty process variable using the popular contribution analysis methodology. Specifically, the following topics are covered

- Introduction to PCA and PLS
- Process fault detection via PCA and PLS
- Fault isolation in PCA- and PLS-based process monitoring applications
- Process monitoring of polymer manufacturing process via PCA
- Process monitoring of polyethylene manufacturing process via PLS

7.1 PCA: An Introduction

Principal component analysis (PCA), in essence, 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 7.1. 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 in data 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.

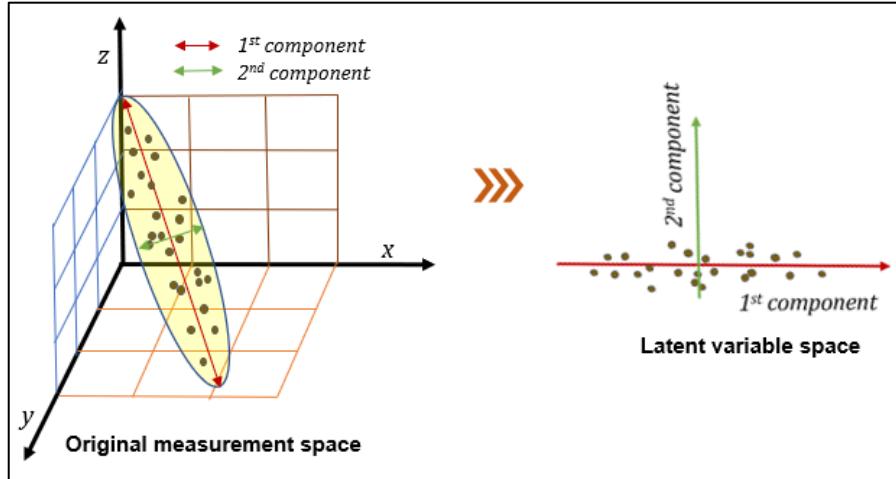


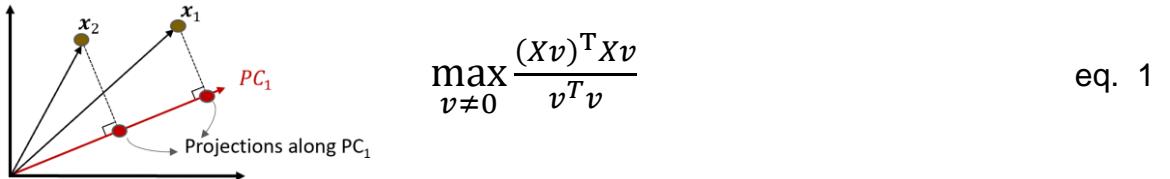
Figure 7.1: PCA illustration

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 PCA³⁴. 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. PCA-based fault detection goes further and compresses all the information in the PC space and the process noise into a couple of control charts. You will soon learn how to generate and use these control charts.

³⁴ 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 $X \in \mathbb{R}^{N \times m}$ 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 $v \in \mathbb{R}^m$ represent the ‘loading’ vector that projects data-points along PC1; it can be found by solving the following optimization problem



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

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

Above is the form you will find commonly in PCA literature. The columns of eigenvector matrix $V \in \mathbb{R}^{m \times m}$ are the loading vectors that we need. The diagonal eigenvalue matrix Λ equals $\text{diag}\{\lambda_1, \lambda_2, \dots, \lambda_m\}$, where $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_m$ 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 $k \lambda_s$) are taken to form the loading matrix $P \in \mathbb{R}^{m \times k}$. Transformed data in the PC space can now be obtained

$$\xleftarrow{\text{Projected values along } j^{\text{th}} \text{ PC}} t_j = X p_j \text{ or } T = X P \quad \leftarrow \begin{array}{c} \text{bulky } X \\ \boxed{\vdots \vdots \vdots \vdots} \\ N \times m \end{array} \ggg \begin{array}{c} \text{lean } T \\ \boxed{\vdots \vdots \vdots} \\ N \times k \end{array} \quad \text{eq. 3}$$

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

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

Note that because we discarded the loading vectors corresponding to insignificant λ_s , $\hat{X} \neq X$. The difference $E = X - \hat{X}$ 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 facility³⁵. The dataset contains 33 variables and 92 hourly samples (Figure 7.2).

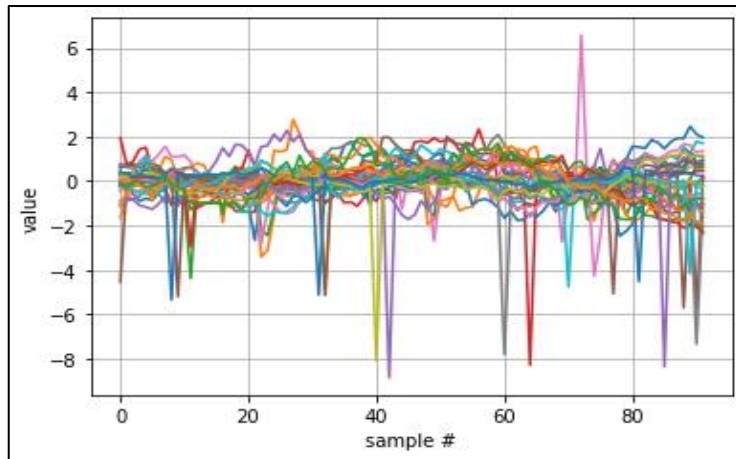


Figure 7.2: Process data from a polymer manufacturing plant. Each colored curve corresponds to a process variable.

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, 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)
```

³⁵ Data was originally available at <https://landing.umetrics.com> (unfortunately, this link no longer seems to work). Dataset is also referenced at https://www.academia.edu/38630159/Multivariate_data_analysis_wiki. Data file is made available in this book's GitHub repository.

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.  0. -0.]
 [ 0.  1.0  0.]
 [-0.  0.  1.0]]
```

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 7.3 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()
```

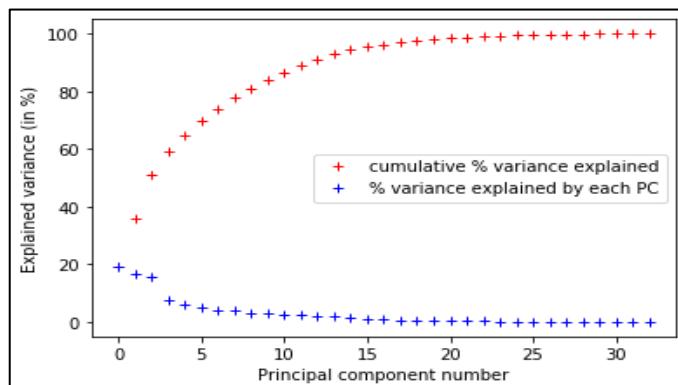


Figure 7.3: Variance explained by principal components

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 7.4 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
```

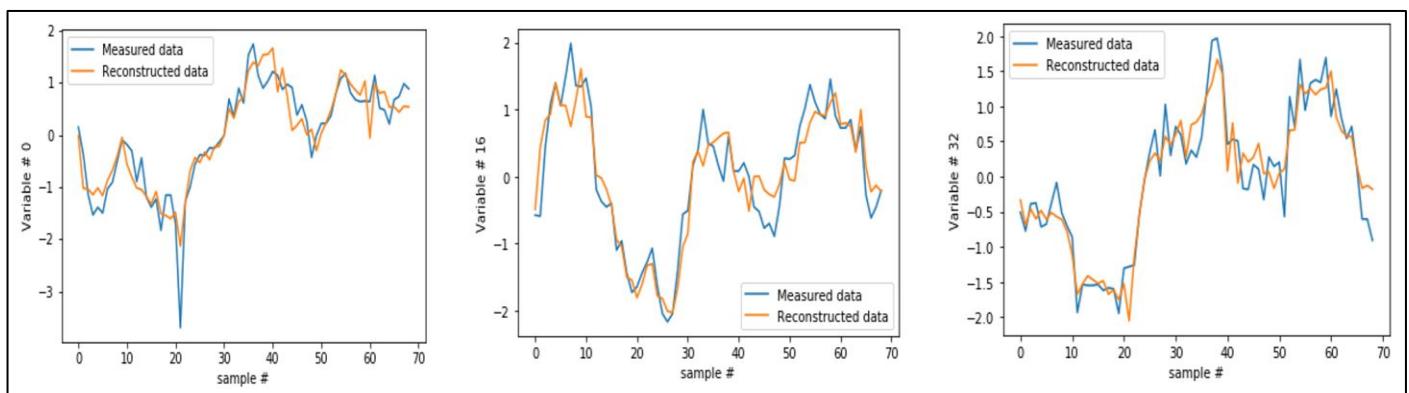


Figure 7.4: Comparison of measured and reconstructed values for a few variables

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
```

7.2 Fault Detection via PCA: Polymer Manufacturing Case Study

In Figure 7.2, 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.

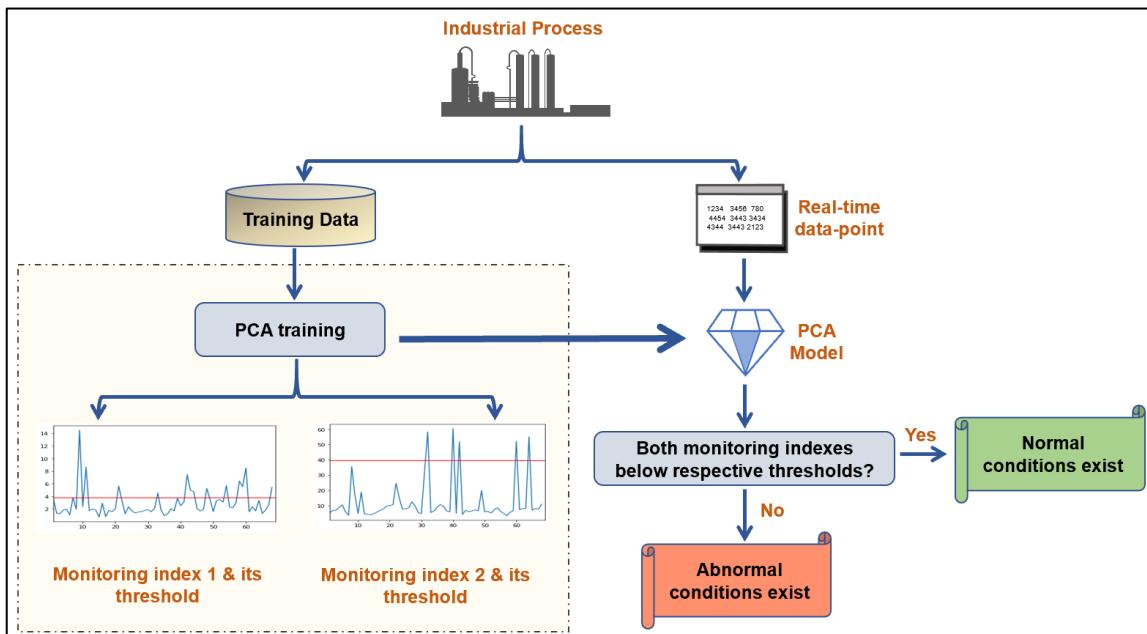


Figure 7.5: PCA-based fault detection workflow

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 7.5. 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.

Fault detection indices

Hotelling's T^2 and SPE (also called Q) statistics are the two monitoring indices utilized for fault detection. Both the statistics are computed for each data-point. [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.]

Hotelling's T^2

Let 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 follows

$$T^2 = \sum_{j=1}^k \frac{t_{i,j}^2}{\lambda_j} = t_i \Lambda_k^{-1} t_i^T \quad \text{eq. 5}$$

Note that $\Lambda_k = \text{diag}\{\lambda_1, \lambda_2, \dots, \lambda_k\}$. 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 control limit for a specified false alarm rate of α^{36} (often set at 0.05 or 0.01) is given as follows

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

$F_{k,N-k}(\alpha)$ is the $(1-\alpha)$ 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.

SPE/Q

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

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

³⁶ In statistical terms, α is the Type I error rate. In colloquial terms, it corresponds to $100*(1-\alpha)\%$ control limit.

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

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

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

z_α is the $(1-\alpha)$ percentile of a standard Gaussian distribution.

Other expressions for control limits



If the number of NOC samples are large (which implies that the mean and covariance of NOC data can be estimated accurately), then the T^2 statistic follows a χ^2 distribution with k degrees of freedom and the control limit is obtained as

$$T_{CL}^2 = \chi_\alpha^2(k)$$

The statistical expressions for the control limits presented so far assume that the NOC data follow a multivariate Gaussian distribution. If this assumption is not true, then the control limits can be estimated via kernel density estimation (demonstrated in Book 1 of the series) or the percentile method (see Chapter 8).

We now have all the information required to generate control charts for the fault indices for training data. We will continue our case study on the polymer manufacturing dataset.

```
# calculate T2 for training data
N, m, k = data_train_normal.shape[0], data_train_normal.shape[1], n_comp

lambda_k = np.diag(pca.explained_variance_[0:k]) # eigenvalue = explained variance
lambda_k_inv = np.linalg.inv(lambda_k)
T2_train = np.zeros((N,))

for i in range(N):
    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
```

```

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_

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 control chart
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 control chart
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')

```

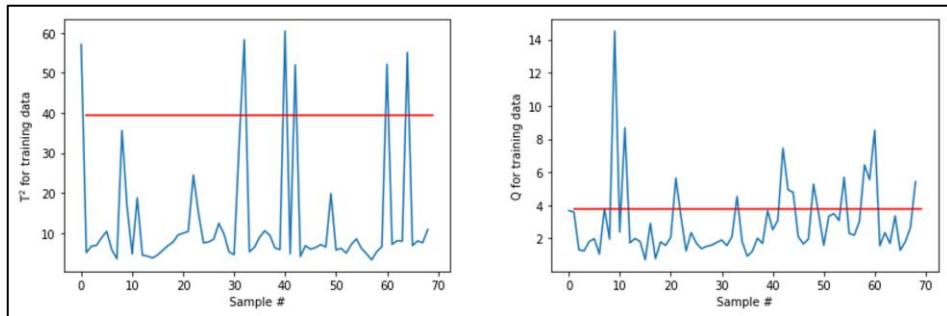


Figure 7.6: Monitoring charts for training data

Figure 7.6 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, as alluded to before, 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.

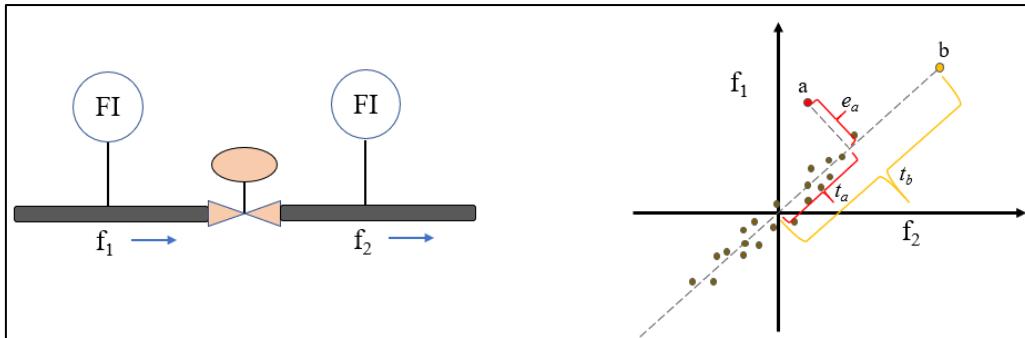


Figure 7.7: (Left) Flow measurements across a valve (Right) Mean-centered flow readings with two abnormal instances (samples a and b)

For example, consider the scenario in Figure 7.7. 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 on test data

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:k]
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 7.8 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.

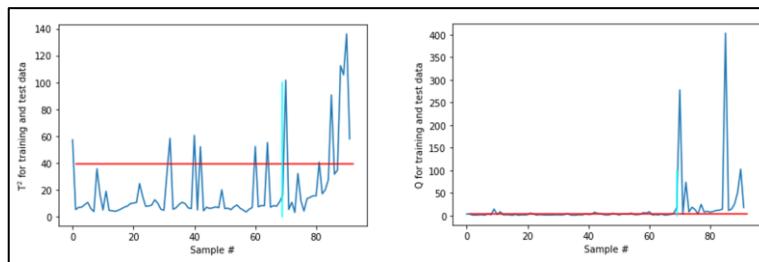


Figure 7.8: Monitoring charts for training and test data. Vertical cyan-colored line separates training and test data

7.3 Fault Isolation via Contribution Analysis for PCA

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.

SPE contributions

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.

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

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.

T^2 contributions

For T^2 contributions, calculations are not as straightforward. Several expressions have been postulated in literature³⁷. The commonly used expression below was proposed by wise et al.³⁸

$$\begin{aligned} T^2 \text{ contribution of variable } j &= \left(j^{\text{th}} \text{ element of } (D^{1/2}x) \right)^2 \\ D &= P\Lambda_k^{-1}P^T \end{aligned} \quad \text{eq. 10}$$

Note that these contributions³⁹ 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.bar(['var ' + str(i+1) for i in range(len(T2_contri))], T2_contri)

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

plt.figure(), plt.bar(['var ' + str(i+1) for i in range(len(SPE_contri))], SPE_contri)
```

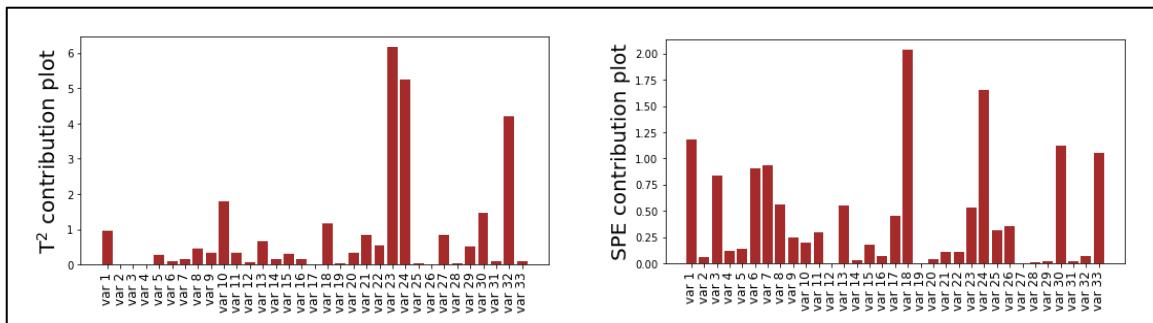


Figure 7.9: T^2 and SPE contribution plots for sample 85

³⁷ S. Joe Qin, Statistical process monitoring: basics and beyond, Journal of Chemometrics, 2003

³⁸ Wise et. al., PLS toolbox user manual, 2006

³⁹ A quick derivation (Alcalá & Qin, Analysis and generalization of fault diagnosis methods for process monitoring. *Journal of Process Control*, 2011) is as follows ($T^2 = x^T P \Lambda_k^{-1} P^T x = \|P \Lambda_k^{-1} P^T x\|^2 \equiv \|D^{1/2}x\|^2 = \sum_{j=1}^m (\xi_j^T D^{1/2} x)^2 = \sum_{j=1}^m T_j^2$). Here, ξ_j is j^{th} column ($\xi_j = [0 \dots 1 \dots 0]^T$) of the identity matrix of size $m \times m$ and T_j^2 is j^{th} variable contribution.

Variable # 24 makes large contributions to both the indices and in Figure 7.10 we can see 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.

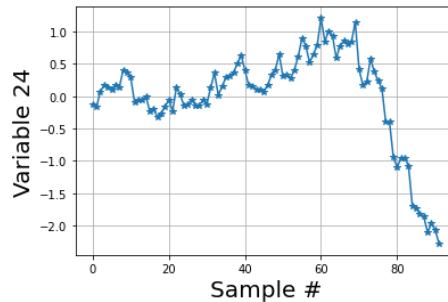


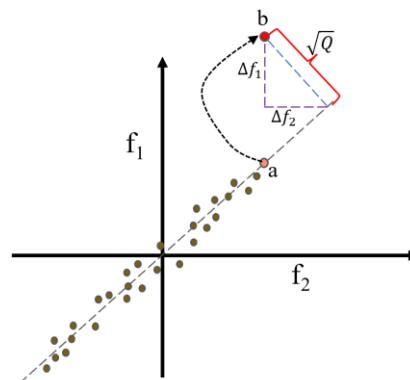
Figure 7.10: Temporal evolution of variable 24

Note that contribution plots do not explicitly identify the underlying cause of the fault; rather, they only isolate the variables that have been most affected by the fault or are most inconsistent with the NOC behavior.

Smearing effect



While contribution analysis is a widely employed method for fault isolation, it is prone to errors – variables that are not impacted by fault can end up having significant contributions. For example, in the simple illustration below, assume that an actual process state denoted by point 'a' is measured as point 'b' due to faulty f_1 sensor. However, as shown, variable f_2 makes significant contribution to the Q metric and therefore, f_2 will be inferred to be faulty as well. The take-home message is that caution must be exercised during interpretation of the contribution plot and the actual data for the highlighted variables should be checked before reporting the faulty variables.



Other approaches for fault isolation have been explored such as the reconstruction-based approach by Alcala & Qin (see the 2009 article title 'Reconstruction-based approach for process monitoring' published in Automatica).

7.4 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 7.11 provides a conceptual comparison of PLS methodology with those of other popular linear regression techniques, principal component regression (PCR) and multivariate linear regression (MLR).

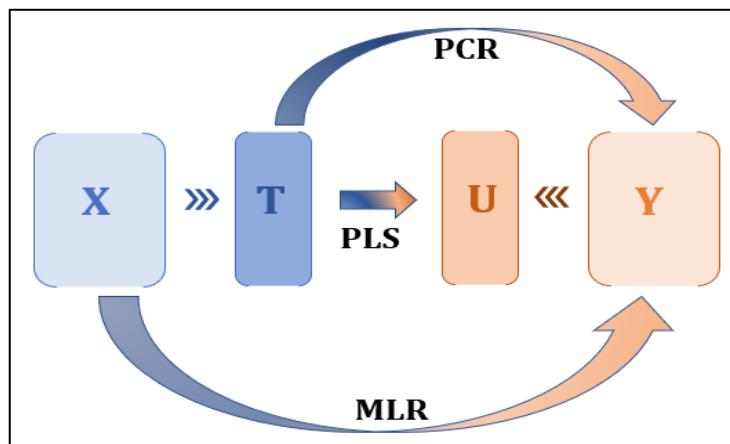


Figure 7.11: PLS, PCR, MLR methodology overview. Note that the score matrix, T , for PLS and PCR can be different.

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 $X \in \mathbb{R}^{N \times m}$ 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, $Y \in \mathbb{R}^{N \times p}$. 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:

$$t_1 = Xw_1 \text{ and } u_1 = Yc_1 \quad \text{eq. 11}$$

The vectors w_1 and c_1 , termed weight vectors, are computed such that the covariance between t_1 and 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.

$$\text{Cov}(t_1, u_1) = \text{Correlation}(t_1, u_1) * \sqrt{\text{Var}(t_1)} * \sqrt{\text{Var}(u_1)}$$

In the next step, with the computed pairs $\{X, t_1\}$ and $\{Y, u_1\}$, loading vectors (p_1 and q_1), are found via least squares regression

$$X = t_1 p_1^T + E_1 \text{ and } Y = u_1 q_1^T + F_1 \quad \text{eq. 12}$$

In Eq. 12, E and F are called residual matrices and represent the part of X and Y that have not yet been captured. To find the next component scores, the above three steps are repeated with matrices E_1 and F_1 replacing X and Y . Note that the maximum number of possible components equals m . For each component, the weight vectors are found via iterative procedures like NIPALS (algorithm shown later) or SIMPLS. The final PLS decomposition looks like the following

$$X = TP^T + E = \sum_{i=1}^k t_i p_i^T + E$$

$$Y = UQ^T + F = \sum_{i=1}^k u_i q_i^T + F \quad \text{eq. 13}$$

where k is the number of latent components computed. The expressions in Eq. (11) are referred to as the outer relations for the X and Y blocks, respectively. An inner relation is also estimated for each pair $\{t_i, u_i\}$ via linear regression

$$u_i \approx b_i t_i$$

or $U = [u_1, u_2, \dots, u_k] \approx [b_1 t_1, b_2 t_2, \dots, b_k t_k] = TB$

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

$$\hat{Y} = \hat{U}Q^T = TBQ^T \quad \text{eq. 14}$$

If these algorithmic details appear intimidating, do not worry⁴⁰. 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 fault detection tool.

Algorithm: PLS model fitting via NIPALS

- 1) Initialize $i = 1$, $X_1 = X, Y_1 = Y$
- 2) Set score vector $u_i (\in \mathbb{R}^{N \times 1})$ to any column of Y_i
- 3) Calculate weight vector $w_i = X^T u_i$
- 4) Compute score vector $t_i (\in \mathbb{R}^{N \times 1}) = X_i w_i$
- 5) Scale t_i to unit length: $t_i = t_i / \|t_i\|$
- 6) Calculate weight vector $c_i = Y_i^T t_i$
- 7) Update score vector $u_i = Y_i c_i$
- 8) Scale u_i to unit length: $u_i = u_i / \|u_i\|$
- 9) Repeat steps 3 to 8 until convergence

10) Deflate matrices:

$$X_{i+1} = X_i - t_i t_i^T X_i$$

$$Y_{i+1} = Y_i - t_i t_i^T Y_i$$

- 11) Set $i = i + 1$ and return to step 2. Stop when $i > k$ [number of latents to be retained]
- 12) Form the matrices $T = [t_1, t_2, \dots, t_k]$
 $U = [u_1, u_2, \dots, u_k]$

⁴⁰ We included the NIPALS algorithm details so that you can follow the mathematical background of KPLS algorithm easily in Chapter 10.

Computing score matrix T directly from X

In PCA, we could directly obtain the score matrix T via the relation $T = XP$ (Eq. (3)). Therefore, it is natural to wonder if we can extend Eq. (11) to write $T = XW$? Unfortunately, this wouldn't be correct as the weight vectors (w) operate on the deflated matrices (see NIPALS algorithm). Nonetheless, there exist another set of vectors called the r vectors⁴¹ or the R matrix which can provide us the scores directly from the original predictor matrix X .

$$t_i = Xr_i \text{ and } T = XR \quad \text{eq. 15}$$

where, $R = W(P^T W)^{-1}$ whose i^{th} column equals the vector r_i . An obvious relationship is that $r_1 = w_1$.

7.5 Fault Detection via PLS: Polyethylene Manufacturing Case Study

Although soft sensing is the predominant use of PLS in process industry, the 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) production process using a simulated model that was tuned to match a typical industrial process⁴². As shown in the figure below, the process consists of a multi-zonal tubular reactor. 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 7.13).

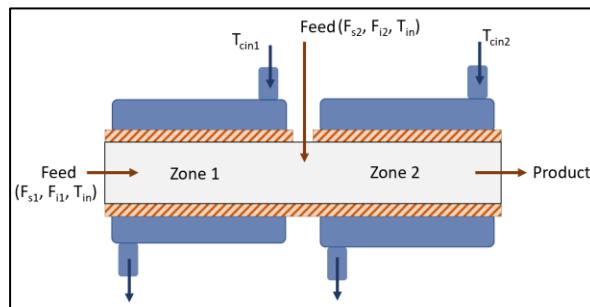


Figure 7.12: LDPE reactor

⁴¹ <https://learnche.org/pid/latent-variable-modelling/projection-to-latent-structures/how-the-pls-model-is-calculated>

⁴² MacGregor et al., Process monitoring and diagnosis by multiblock PLS methods, Process Systems Engineering, 1994. This dataset can be obtained from <https://openmv.net>.

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.

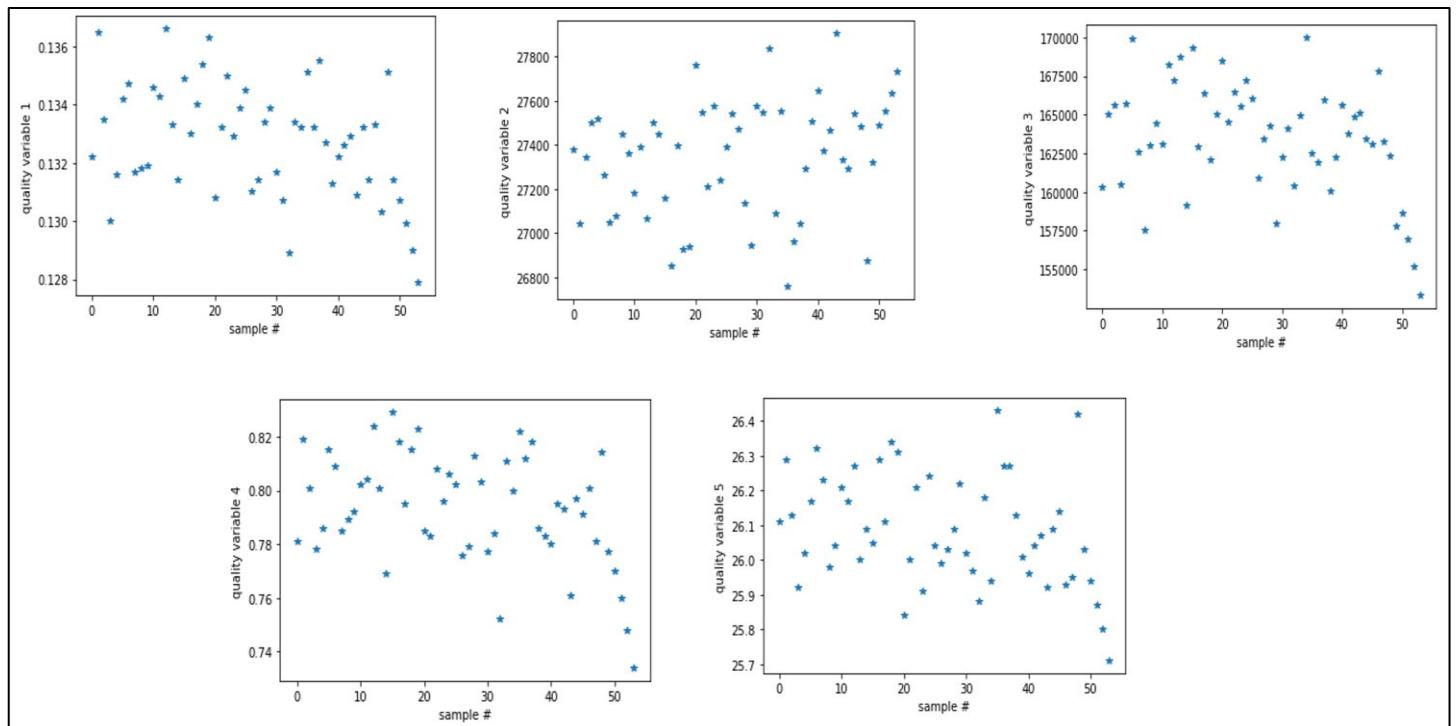


Figure 7.13: Plot of quality variables in LDPE dataset

We begin by building a PLS model using 3 latent components⁴³.

```
# 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)
```

⁴³ Kourtogi & MacGregor, Process analysis, monitoring and diagnosis, using multivariate projection methods, Chemometrics and Intelligent Laboratory Systems, 1995

```
# 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)
```

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.

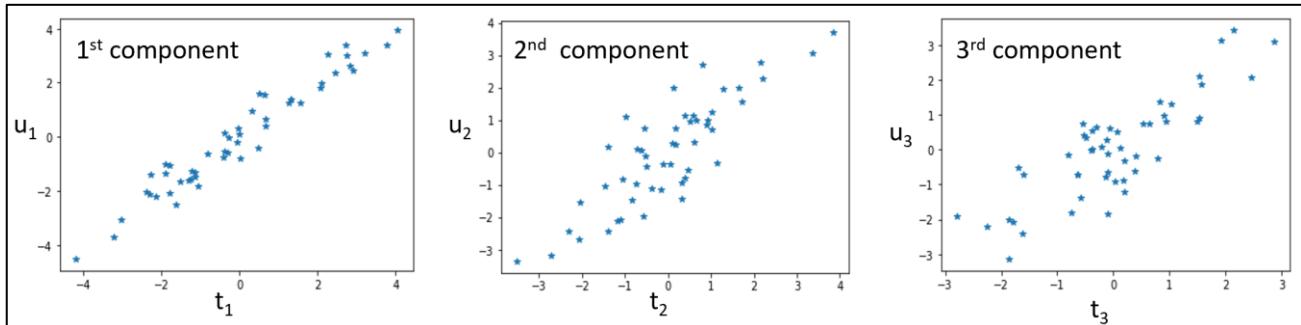


Figure 7.14: X-scores vs Y-scores. Here t_j and u_j refer to the j^{th} columns of T and U matrices, respectively.

Fault detection indices

For PLS, three monitoring indices are computed: one T^2 statistic from t-scores, and two SPE statistics from X and Y residuals.

T^2 statistic

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 t_i denote the i^{th} row of T . The T^2 index for this data-point is given by

$$T^2 = \sum_{j=1}^k \frac{t_{ij}^2}{\sigma_j} = t_i \Lambda_k^{-1} t_i^T$$

Λ_k , a diagonal matrix, is the covariance matrix of T with σ_j (variance of j^{th} component scores) as its diagonal elements. T_{CL}^2 for a false rate of α is obtained by the following expression

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

SPE_x and SPE_y statistics

The second and third indices, SPE_x and SPE_y , represents the residuals or the unmodelled part of X and Y , respectively. Let e_i and f_i denote the i^{th} row of E and 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 and large number of training samples, the control limit for SPE statistic is given by the following expression

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

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

χ^2_α is the $(1-\alpha)$ percentile of a chi-squared distribution⁴⁴ with h degrees of freedom; μ denotes the mean value and σ denotes the variance of the SPE statistic. Let's now generate the control charts for the fault detection indices.

```
# monitoring indices for training data
# T2
N = data_train_normal.shape[0]
k = 3

T_cov = np.cov(Tscores.T)
T_cov_inv = np.linalg.inv(T_cov)
T2_train = np.zeros((N,))
for i in range(N):
    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
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
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)
```

⁴⁴ Yin et al., A review of basic data-driven approaches for industrial process monitoring, IEEE Transactions on Industrial Electronics, 2014

```

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

```

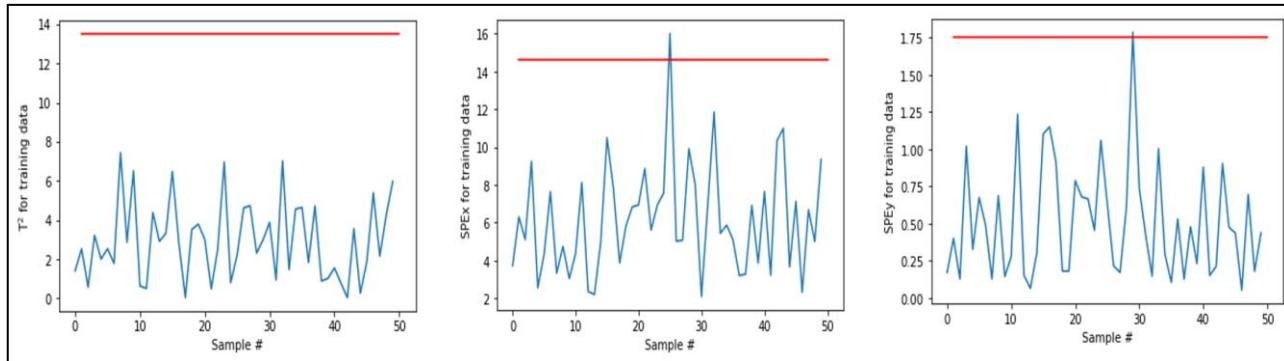


Figure 7.15: Monitoring charts for LDPE training data

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. Which specific input variable is the culprit? The fault isolation exercise in the next section answers this question.

```

# 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)

```

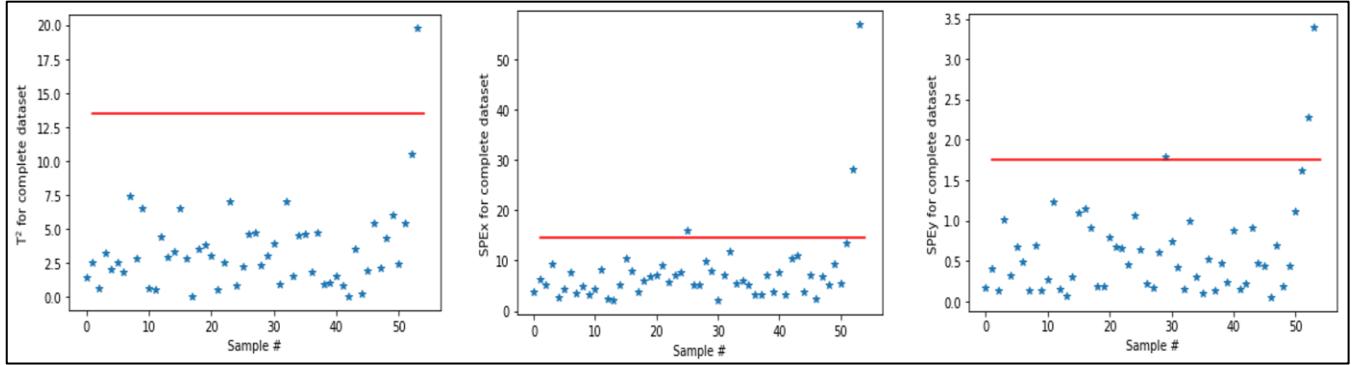


Figure 7.16: Monitoring charts for the complete LDPE dataset

7.6 Fault Isolation via Contribution Analysis for PLS

Contribution analysis-based fault isolation for PLS model proceed along the similar lines as that for PCA. Each monitoring statistic is broken down into contributions from individual variables and the variables with highest contributions are flagged as potentially faulty variables.

SPE_x and SPE_y contributions

For SPE statistics, the following decompositions take place.

$$SPE_x = \sum_{j=1}^m e_j^2 = \sum_{j=1}^m SPE_{x,j}$$

$$SPE_y = \sum_{j=1}^p f_j^2 = \sum_{j=1}^p SPE_{y,j}$$

T² contributions

To find T^2 contributions, the following approach is adopted⁴⁵. Let $x^{sample} \in \mathbb{R}^{m \times 1}$ and $t^{sample} \in \mathbb{R}^{k \times 1}$ denote the input vector and the score vector, respectively, for a candidate sample. Then,

$$\begin{aligned} T^2 &= (t^{sample})^T \Lambda_k^{-1} t^{sample} = \|\Lambda_k^{-0.5} t^{sample}\|^2 \\ &= \|\Lambda_k^{-0.5} R^T x^{sample}\|^2 = \|\Gamma x^{sample}\|^2 \end{aligned}$$

⁴⁵ Choi & Lee, Multiblock PLS-based localized process diagnosis. *Journal of Process Control*, 2005.

$$\Rightarrow T^2 = \left\| \sum_{j=1}^m \Gamma(:, j) x^{sample}(j) \right\|^2$$

where, $\Gamma = \Lambda_k^{-0.5} R^T$, $\Gamma(:, j)$ is the j^{th} column of Γ , and $x^{sample}(j)$ is the value of the j^{th} input variable in the sample. The contribution of the j^{th} input variable for T^2 is given as $\|\Gamma(:, j)x^{sample}(j)\|^2$. Let's now find which variables need to be further investigated at the 54th sample.

```
# SPEx contribution
sample = 54
data_point = np.transpose(data_normal[sample-1,])

x_error_test_sample = x_error_test[sample-1,]
SPEx_contri = x_error_test_sample*x_error_test_sample # vector of contributions

plt.figure(), plt.bar(['var ' + str((i+1)) for i in range(len(SPEx_contri))], SPEx_contri,
plt.xticks(rotation = 90), plt.ylabel('SPEx contribution plot')

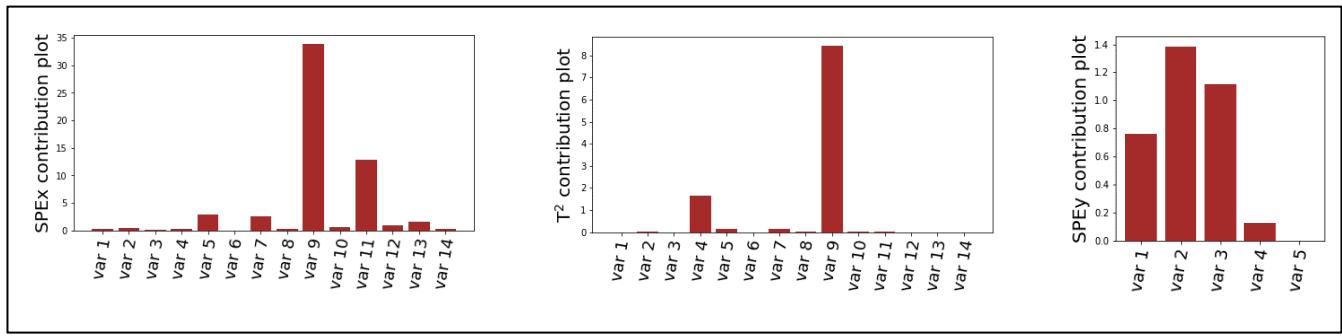
# SPEy contribution
y_error_test_sample = y_error_test[sample-1,]
SPEy_contri = y_error_test_sample*y_error_test_sample # vector of contributions

plt.figure(), plt.bar(['var ' + str((i+1)) for i in range(len(SPEy_contri))], SPEy_contri,
plt.xticks(rotation = 90), plt.ylabel('SPEy contribution plot')

# T2 contribution
W = pls.x_weights_
P = pls.x_loadings_
R = np.dot(W, np.linalg.inv(np.dot(P.T, W)))
Ghe = np.dot(scipy.linalg.sqrtm(T_cov_inv), R.T)

T2_contri = np.zeros((X_train_normal.shape[1],))
for i in range(X_train_normal.shape[1]):
    vect = Ghe[:,i]*data_point[i]
    T2_contri[i] = np.dot(vect, vect)

plt.figure(), plt.bar(['var ' + str((i+1)) for i in range(len(T2_contri))], T2_contri,
plt.xticks(rotation = 90), plt.ylabel(T$^2$ contribution plot')
```

Figure 7.17: T^2 and SPE contribution plots for sample 54

Variable # 9 makes large contributions to both SPE_x and T^2 indexes and in Figure 7.10 we can see that there was a sharp increase in its value towards the end of the sampling period. Variable # 9 was also reported as the most contributing variable in the work of McGregor et al⁴³.

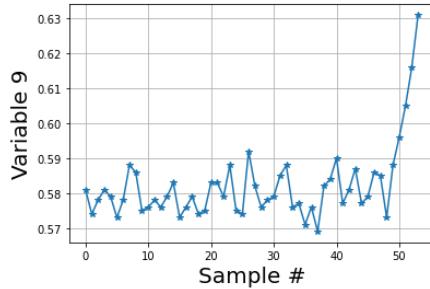


Figure 7.18: Temporal evolution of variable 24

With this, we have come to the end of our study of PCA- and PLS-based monitoring solution development. You probably now understand why these two techniques are so popular. However, notwithstanding the powerful capabilities of PCA and PLS, you may encounter problems where vanilla-PCA and vanilla-PLS fail to provide satisfactory performance. Process data exhibiting severe nonlinearity and/or dynamics require some modifications to the vanilla techniques. We will study these variants in the upcoming chapters.

Summary

With this chapter we have reached a significant milestone in our ML-based PM/PdM 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 many of the process 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.

Chapter 8

Multivariate Statistical Process Monitoring for Linear and Steady-State Processes: Part 2

By now you must be very impressed with the powerful capabilities of PCA and PLS techniques. These methods allowed us to extract latent variables and monitor systematic variations in latent space and process noise separately. However, you may ask, “Are these the best latent variable-based techniques to use for all problems?”. We are glad that you asked! Other powerful methods do exist which may be better suited in certain scenarios. For example, independent component analysis (ICA) is preferable over PCA when process data is not Gaussian distributed. It can provide latent variables with stricter property of statistical independence rather than only uncorrelatedness. Independent components may be able to characterize the process data better than principal components and thus may result in better monitoring performance.

In another scenario, if your end goal is to classify process faults into different categories for fault diagnosis, then, maximal separation between data from different classes of faults would be your primary concern rather than maximal capture of data variance. Fisher discriminant analysis (FDA) is preferred for such tasks.

In this chapter, we will learn in detail the properties of ICA and FDA. We will apply these methods for process monitoring and fault classification for a large-scale chemical plant. Specifically, the following topics are covered

- Introduction to ICA
- Process monitoring of non-Gaussian processes
- Introduction to FDA
- Fault classification for large scale processes.

8.1 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 8.1 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 accurately⁴⁶.

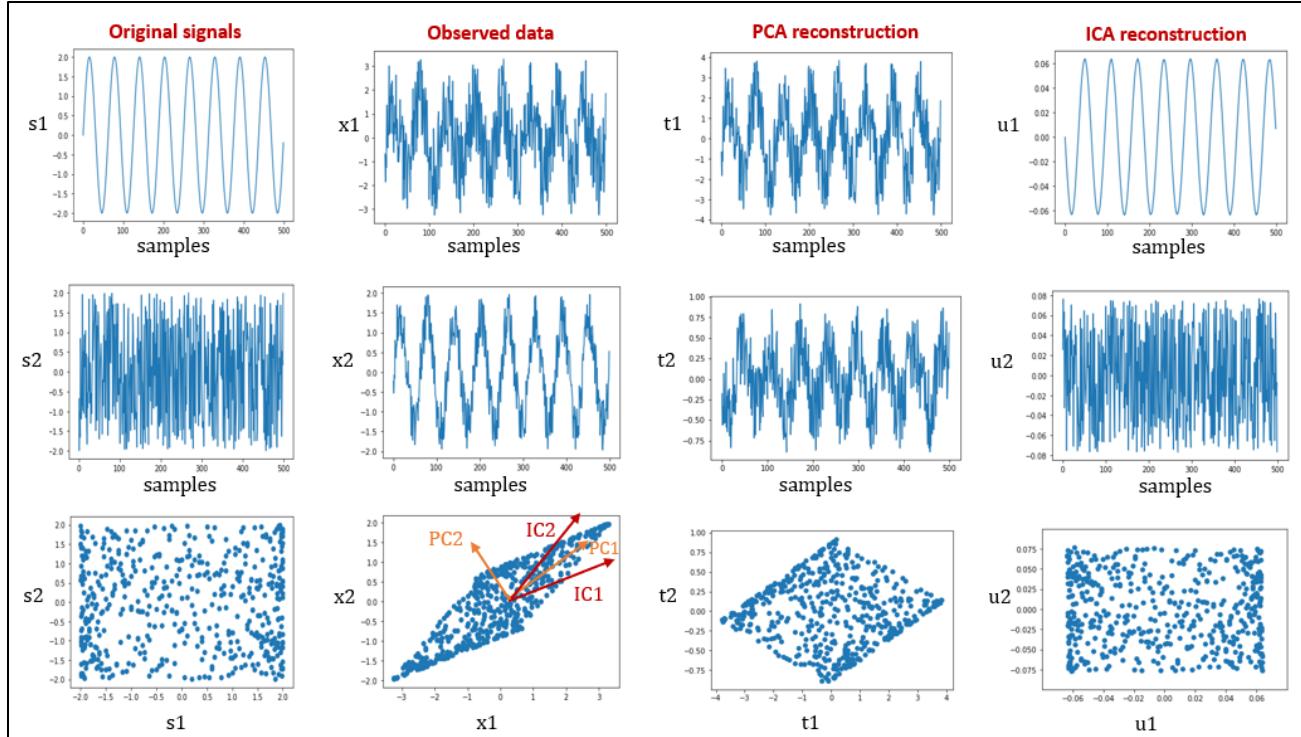


Figure 8.1: Simple illustration of ICA vs PCA. The arrows in the x_1 vs x_2 plot show the direction vectors of corresponding components. Note that the signals t_1 and t_2 are not independent as value of one variable influences the range of values of the other variable.

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

⁴⁶ If you observe closely, you will find that ICA latent signals (u_1 and u_2) do differ from s_1 and s_2 signals in terms of sign and magnitude; we will soon learn why this happens and why this is not a cause of worry.

(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 8.2 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.

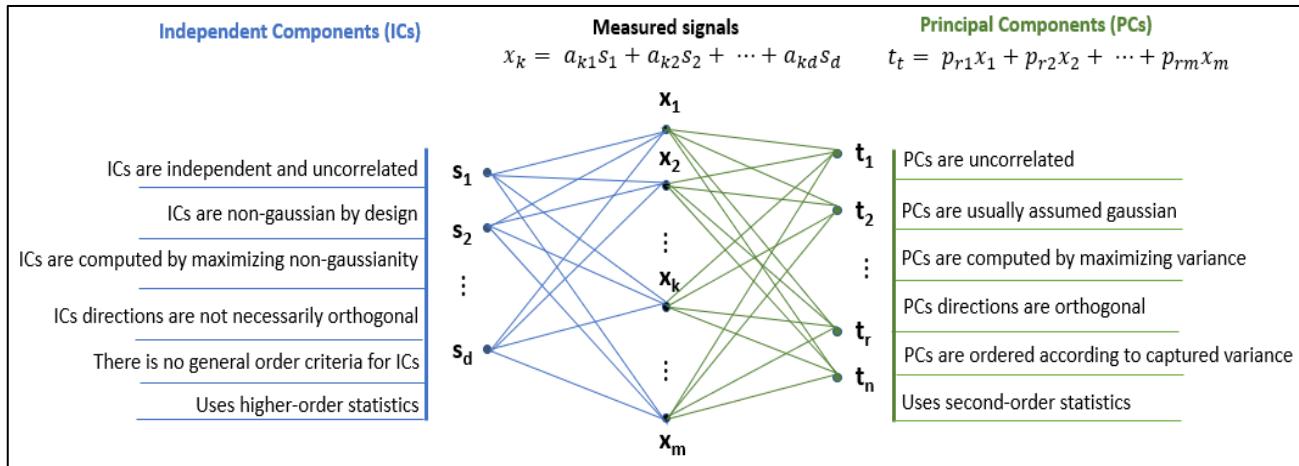


Figure 8.2: Differences between ICA and PCA

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, y_1 and y_2 , 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

$$p(y_1, y_2) = p(y_1)p(y_2)$$

Where $p(y_1)$ is the probability density function of y_1 alone, and $p(y_1, y_2)$ is the joint probability density function. The variables y_1 and y_2 are said to be uncorrelated if their covariance is zero

$$C(y_1, y_2) = E\{(y_1 - E(y_1))(y_2 - E(y_2))\} = E(y_1 * y_2) - E(y_1)E(y_2) = 0$$

Where $E(\cdot)$ 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 $X \in \mathbb{R}^{m \times N}$ 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 ($\leq m$) independent components s_1, s_2, \dots, s_d .

$$X = AS \quad \text{eq. 1}$$

where $S \in \mathbb{R}^{d \times N}$ and $A \in \mathbb{R}^{m \times d}$ 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 (\hat{S}) become as independent as possible.

$$\hat{S} = WX \quad \text{eq. 2}$$

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.

$$Z = QX \quad \text{eq. 3}$$

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

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

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,

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

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 (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 components⁴⁷ 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

⁴⁷ www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_ICA09.pdf is an excellent quick read for more details on this.

sign and magnitude of the original ICs cannot be uniquely estimated. This however does not 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 8.1 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)

To demonstrate the superior performance of ICA and FDA over PCA, we will use the TEP dataset that we previously saw in Chapter 3. We know that this process consists of several unit operations: a reactor, a condenser, a separator, a stripper, and a recycle compressor⁴⁸. There are 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 dataset contains 480 samples collected over 24 operation hours and test dataset contains 960 samples collected over 48 operation hours. For the faulty data, faulty operation starts from sample 160 onwards.

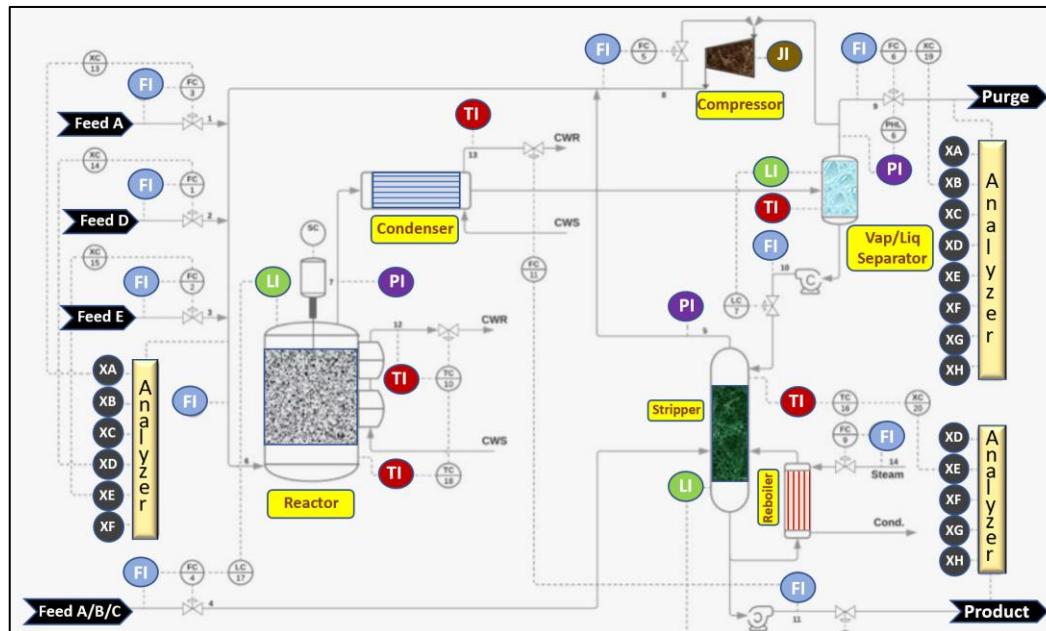


Figure 8.3: Tennessee Eastman process flowsheet⁴⁹

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

⁴⁸ Detailed information about the process and the faults can be obtained from the original paper by Downs and Vogel titled 'A plant-wide industrial process control problem'.

⁴⁹ Adapted from the original flowsheet by Gilberto Xavier (<https://github.com/gmxavier/TEP-meets-LSTM>) provided under Creative-Commons Attribution 4.0 International License (<https://creativecommons.org/licenses/by/4.0/>).

stripper temperature profile (Figure 8.4a). The plot in PC space (Figure 8.4b) 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 and fetch TE data
import numpy as np, matplotlib.pyplot as plt
TEdata_noFault_train, TEdata_Fault_train = np.loadtxt('d00.dat').T, 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')
```

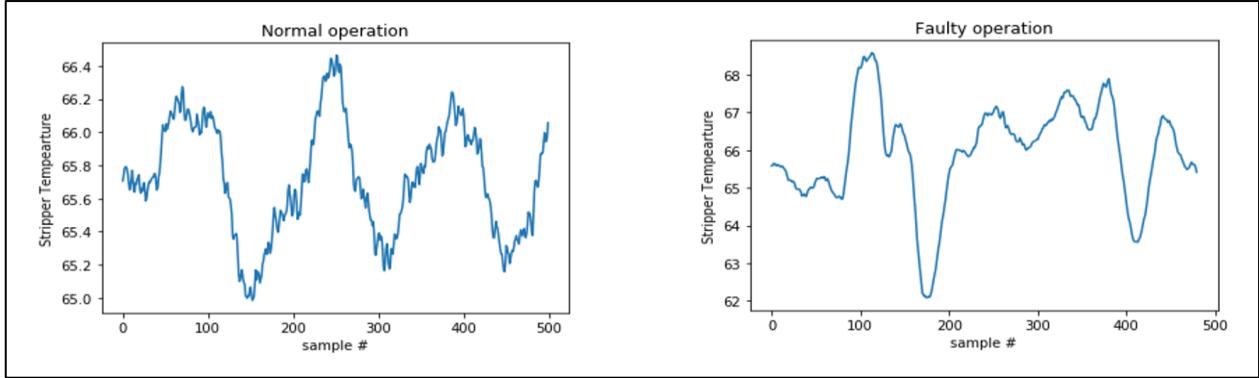


Figure 8.4 (a): Normal vs faulty process profile in Tennessee Eastman dataset

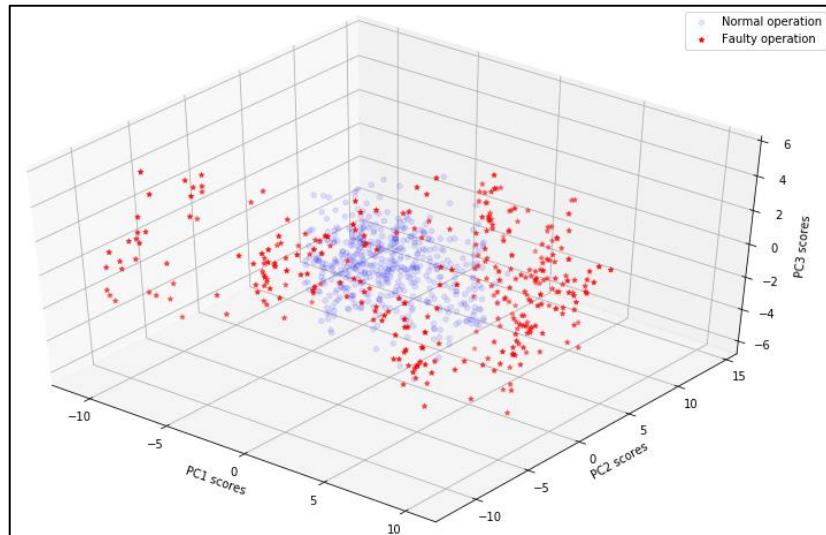


Figure 8.4 (b): Normal vs faulty (Fault 10) TE process data in PC space

Deciding the number of independent components

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. al⁵⁰.

Equation 2 shows that the i^{th} row (w_i) of demixing matrix W corresponds to the i^{th} IC. Therefore Lee et al. suggested using the Euclidean norm (L_2) of w_i to quantify the importance of i^{th} IC and subsequently order the ICs in decreasing order of importance. Using this rationale, Figure 8.5 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
xmvt = TEdata_noFault_train[:,41:52] # 11 manipulated variables
data_noFault_train = np.hstack((xmeas, xmvt))

# 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

⁵⁰ Lee et al., Statistical process monitoring with independent component analysis, Journal of Process Control, 2004

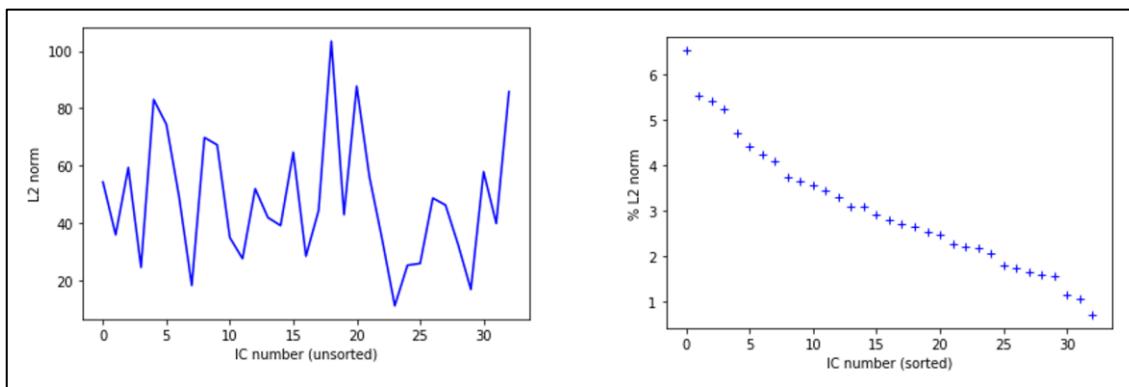


Figure 8.5: (Unsorted) L_2 norm of each row of W and (sorted) percentage of the L_2 norms

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 8.5, 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)

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

# 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
```

Note that the FastICA.fit method expects each row of data matrix (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.

8.2 Fault Detection via ICA: Tennessee Eastman Process Case Study

The illustration example in Figure 8.1 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 NOC 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.

T² statistic

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 s_i denote the i^{th} column of matrix 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

$$I^2 = s_i^T s_i \quad \text{eq. 6}$$

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.

SPE statistic

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 B_d by selecting the columns from matrix B whose indices correspond to the indices of the rows selected from W when we generated matrix W_d . Let x_i and \hat{x}_i denote the i^{th} measured and reconstructed sample. Then,

$$\begin{aligned} \hat{x}_i &= Q^{-1}B_d s_i = Q^{-1}B_d W_d x_i \\ e_i &= x_i - \hat{x}_i \\ SPE &= e_i^T e_i \end{aligned} \quad \text{eq. 7}$$

I_e^2 statistic

The third metric, I_e^2 , is based on the excluded ICs. Let W_e contain the excluded rows of matrix W . Then

$$\begin{aligned}s_i^e &= W_e x_i \\ I_e^2 &= (s_i^e)^T s_i^e\end{aligned}\quad \text{eq. 8}$$

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 8.6 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[number_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, le2, 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, le2 chart, SPE chart
draw_monitoring_chart(ICA_statistics[:,0], CLs[0], 'I2 for ' + trainORtest + ' data')
draw_monitoring_chart(ICA_statistics[:,1], CLs[1], 'le2 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)
le2_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, le2_CL, SPE_CL], 'training')

```

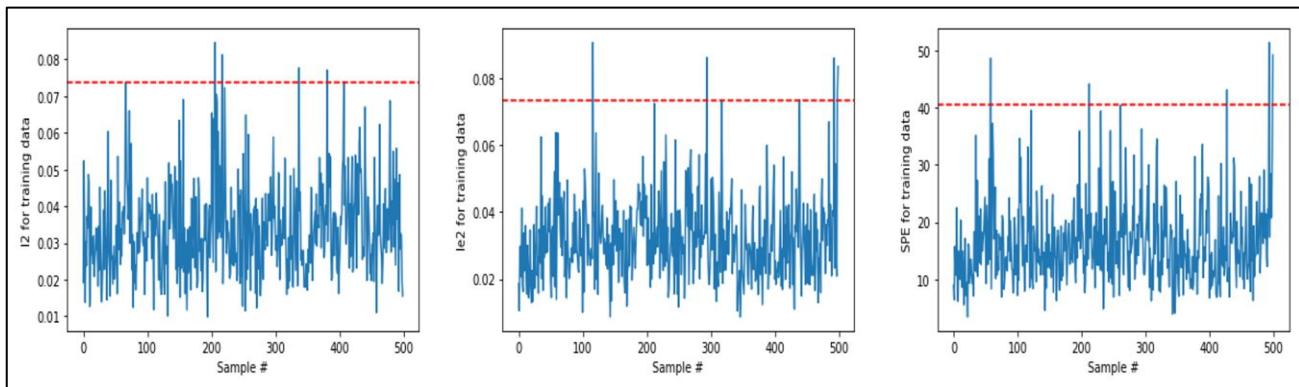


Figure 8.6: ICA monitoring charts for TEP training data

Fault detection on test data

Lee et al.⁵¹ and Yin et al.⁵² have published detailed accounts of fault detection 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.

Figures 8.7 and 8.8 show the ICA/PCA monitoring charts for faults 10 and 5. 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 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
    -----
```

⁵¹ Lee et al., Statistical monitoring of dynamic processes based on dynamic independent component analysis, Chemical Engineering Science, 2004

⁵² 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

```

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

```

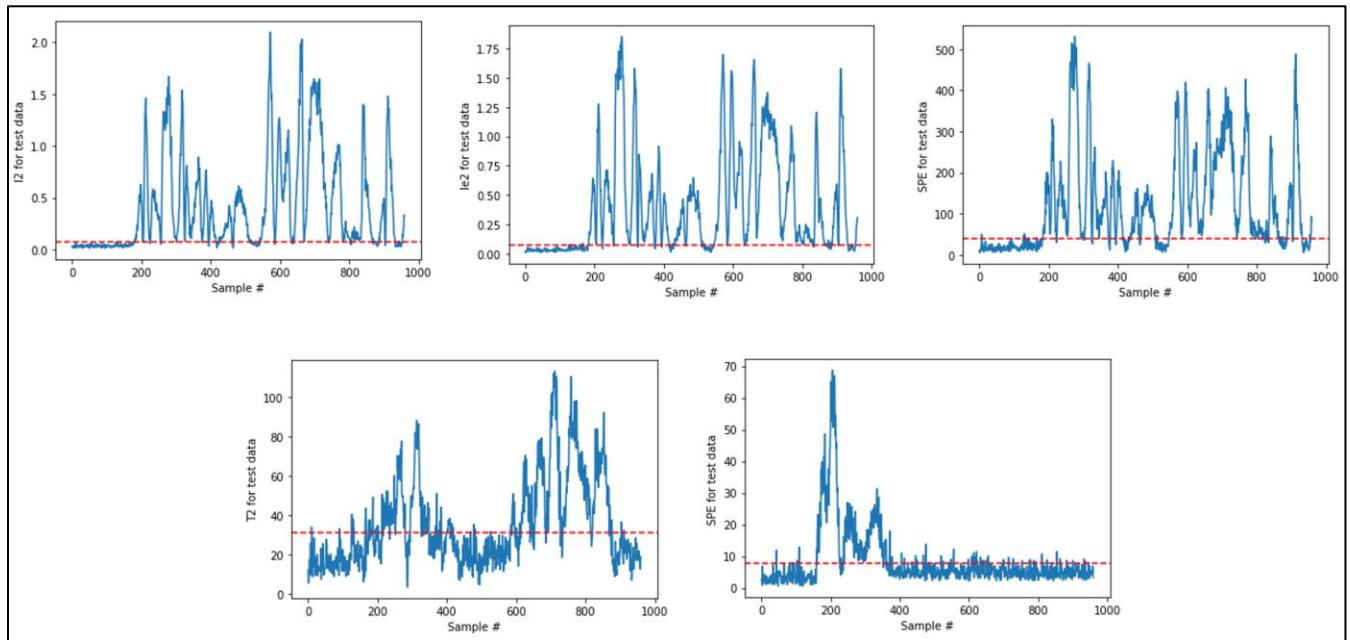


Figure 8.7: Monitoring charts for Fault 10 data with ICA (top, FDR = 90.8%) and PCA (bottom, FDR = 75.6%) metrics

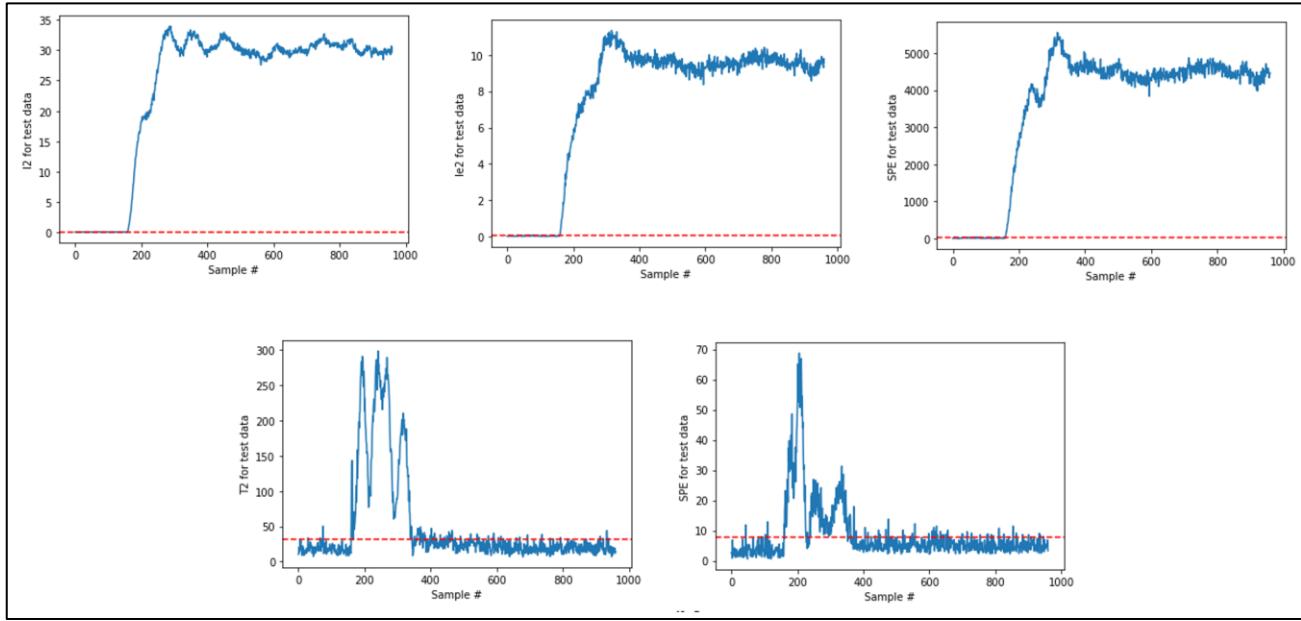


Figure 8.8: Monitoring charts for Fault 5 data with ICA (top, FDR = 100%) and PCA (bottom, FDR = 41.5%) metrics

8.3 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 8.9 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.

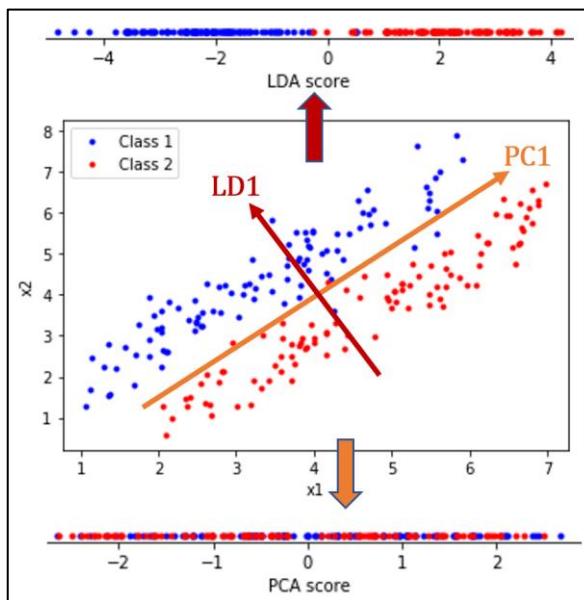


Figure 8.9: Simple illustration of FDA vs PCA. The arrows in the x_1 vs x_2 plot show the direction vectors of 1st components of the corresponding methods

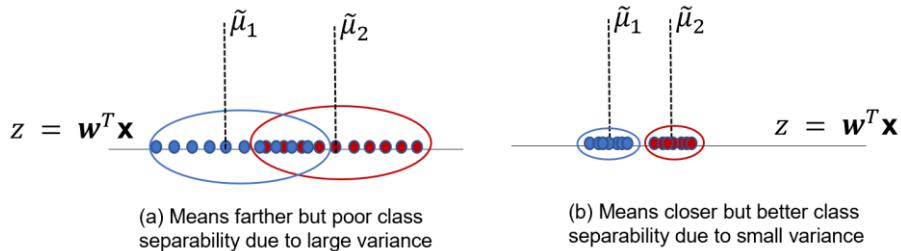
Due to the powerful data discrimination ability of FDA, it is widely used in process industry for operating mode classification and fault diagnosis. Large-scale industrial processes often experience different kinds of process issues/faults and it is imperative to accurately identify the specific issue 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 $X \in \mathbb{R}^{N \times m}$ 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 $w \in \mathbb{R}^m$ such that the projected scalars/samples ($z = w^T x$) are maximally separated. Let $\tilde{\mu}_1$ and $\tilde{\mu}_2$ denote the means of projected values of classes 1 and 2, respectively

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

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.



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.

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

Using the base relation $y = w^T x$ and straightforward algebraic manipulations⁵³, one can equivalently represent the objective $J(w)$ as follows which also holds for any (p) number of data classes

$$J(w) = \frac{w^T S_b w}{w^T S_w w} \quad \text{eq. 9}$$

S_b (between-class-scatter matrix) and S_w (within-class-scatter matrix) are defined as

$$\begin{aligned} S_b &= \sum_{j=1}^p N_j (\mu_j - \mu) (\mu_j - \mu)^T \\ S_w &= \sum_{j=1}^p S_j \\ S_j &= \sum_{x_i \in \omega_1} (x_i - \mu_j)(x_i - \mu_j)^T \end{aligned}$$

where, $\mu \in \mathbb{R}^m$ and $\mu_j \in \mathbb{R}^m$ 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, 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

⁵³ Elhabian & Farag, A tutorial on data reduction Linear Discriminant Analysis, September 2009

most $p-1$ FDA vectors. Alternatively, like PCA, the vectors can also be computed as solutions of a generalized eigenvalue problem

$$S_w^{-1} S_b w = \lambda w \quad \text{eq. 10}$$

where $\lambda = J(w)$. 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

$$Z = XW_p$$

where $W_p \in \mathbb{R}^{m \times (p-1)}$ contains the $p-1$ FDA vectors as columns and $Z \in \mathbb{R}^{N \times (p-1)}$ 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.

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
xmvt = TEdata_Faulty_train[:,41:52] # 11 manipulated variables
data_Faulty_train = np.hstack((xmeas, xmvt))

# 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()

```

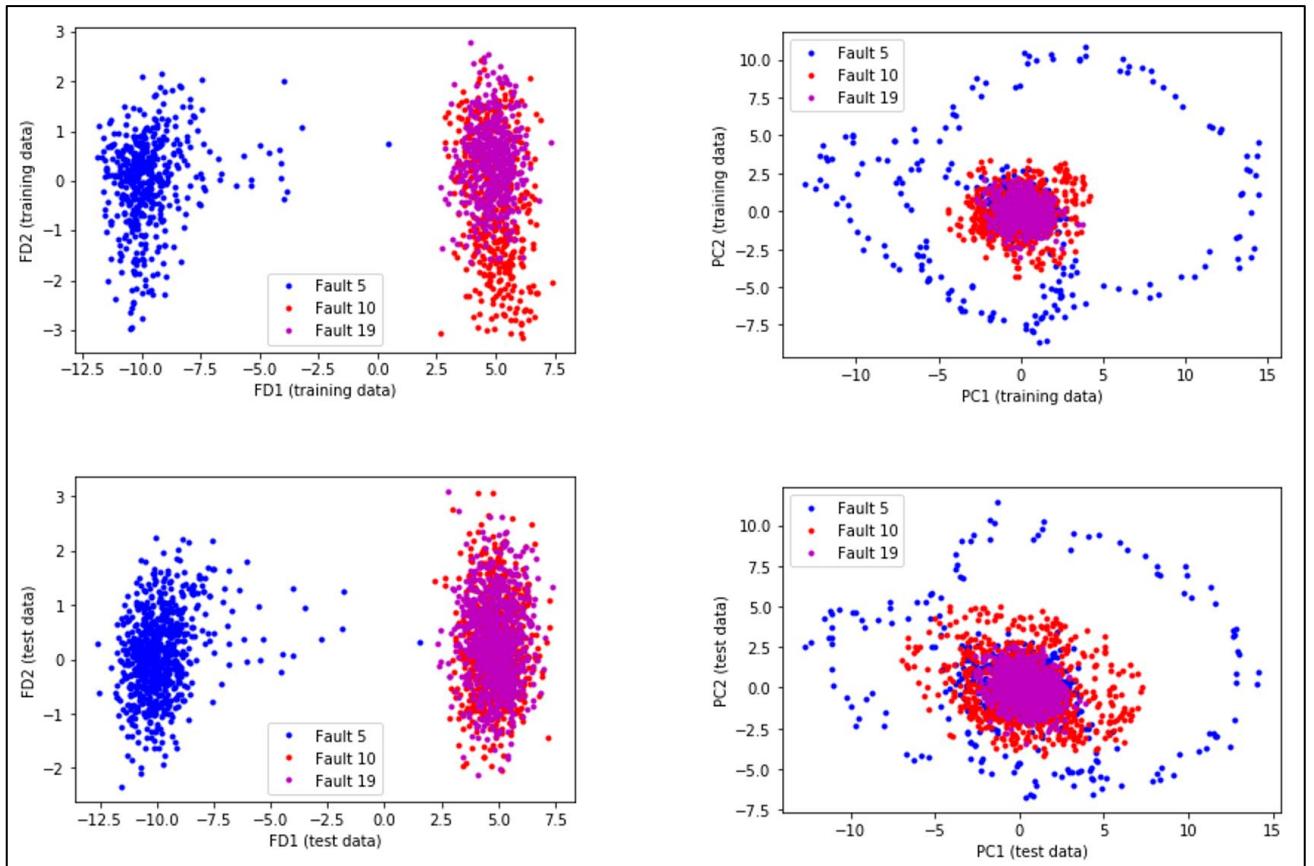


Figure 8.10: FDA and PCA scores in 2 dimensions with 3 fault classes from TEP training (top) and test (bottom) dataset

Figure 8.10 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 explored⁵⁴. Linear PCA, on the other hand, fails to separate any of the classes.



While FDA is a powerful tool for fault diagnosis, it can also be used for fault detection by including NOC data as a separate class.

8.4 Fault Classification via FDA: Tennessee Eastman Process Case Study

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 ($T_{CL,j}^2$) 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

$$T_{sample,j}^2 < T_{CL,j}^2 \Rightarrow \text{sample belongs to the } j^{\text{th}} \text{ fault class}$$

$T_{sample,j}^2$ for a sample for the j^{th} class is given by

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

where, $\tilde{\mu}_j$ and \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 z_{sample} denotes the projected sample in the FDA space. $T_{CL,j}^2$ can be obtained using the same expression we used in PCA, $k(N_j^2 - 1)/N_j(N_j - k) F_{k,N-k}(\alpha)$, 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.

⁵⁴ Hyun-Woo Cho, Nonlinear feature extraction and classification of multivariate process data in kernel feature space, Expert Systems with Applications, 2007

```

# 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]
xmvt = TEdata_Fault5_test[:,41:52]
data_Faulty_test = np.hstack((xmeas, xmvt))

# 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)')

```

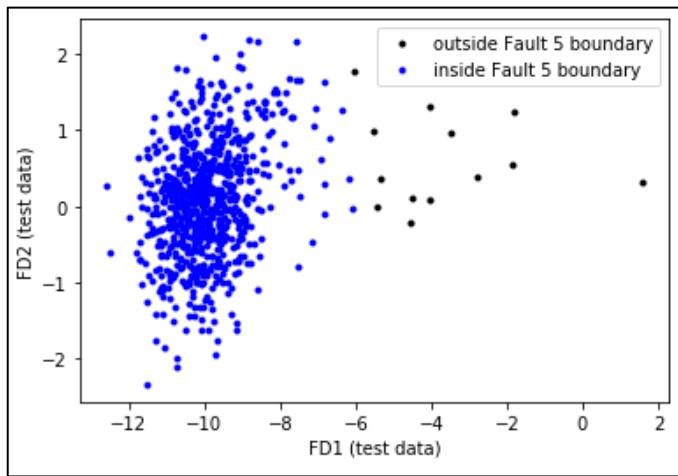


Figure 8.11: Classification result for TEP Fault 5 test samples

About 98% of the samples have been correctly identified as belonging to Fault 5. As shown in Figure 8.11, some of the samples which fall far away from the mean violate the $T_{CL,5}^2$ and therefore are not classified as Fault 5.

Summary

With this chapter, we have now covered the vanilla version of four major MSPM 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 than 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 variants of PCA and PLS suitable for dynamic data in the next chapter.

Chapter 9

Multivariate Statistical Process Monitoring for Linear and Dynamic Processes

In the previous chapters, we saw how beautifully latent variable-based MSPM techniques can extract hidden steady-state relationships from data. However, we imposed a major restriction of absence of dynamics in the dataset. Unfortunately, it is common to have to deal with industrial datasets that exhibit significant dynamics and the standard MSPM techniques fail in extracting dynamic relationships among process variables. Nonetheless, the MSPM research community came up with a simple but ingenious modification to the standard MSPM techniques that made working with dynamic dataset very easy. The trick entails including the past measurements as additional process variables. That's it! The standard techniques can then be employed on the augmented dataset. The dynamic variants of the standard MSPM techniques are dynamic PCA (DPCA), dynamic PLS (DPLS), dynamic ICA (DICA), etc.

Dynamic PCA and dynamic PLS are among the most popular techniques for monitoring linear and dynamic processes; accordingly, these are covered in detail in this chapter. Additionally, this chapter also introduces another very popular and powerful technique that is specially designed to extract dynamic relationships from process data – canonical variate analysis (CVA). Using numerical and industrial-scale case studies, we will see how to use these three techniques to build fault detection tools. Specifically, the following topics are covered

- Introduction to dynamic PCA
- Fault detection using DPCA
- Introduction to dynamic PLS
- Introduction to CVA
- Fault detection using CVA for Tennessee Eastman process

9.1 Dynamic PCA: An Introduction

Dynamic PCA is the dynamic extension of conventional PCA designed to handle process data that exhibit significant dynamics. DPCA simply entails application of conventional PCA to augmented data matrix which, as shown in Figure 9.1, is generated by using past measurements as additional process variables. Note that each ‘variable’ of the augmented matrix is normalized to zero mean and unit variance as is done in conventional PCA. It may seem surprising, but such a simple approach has achieved great success in process industry and has been readily adopted due to ease of implementation.

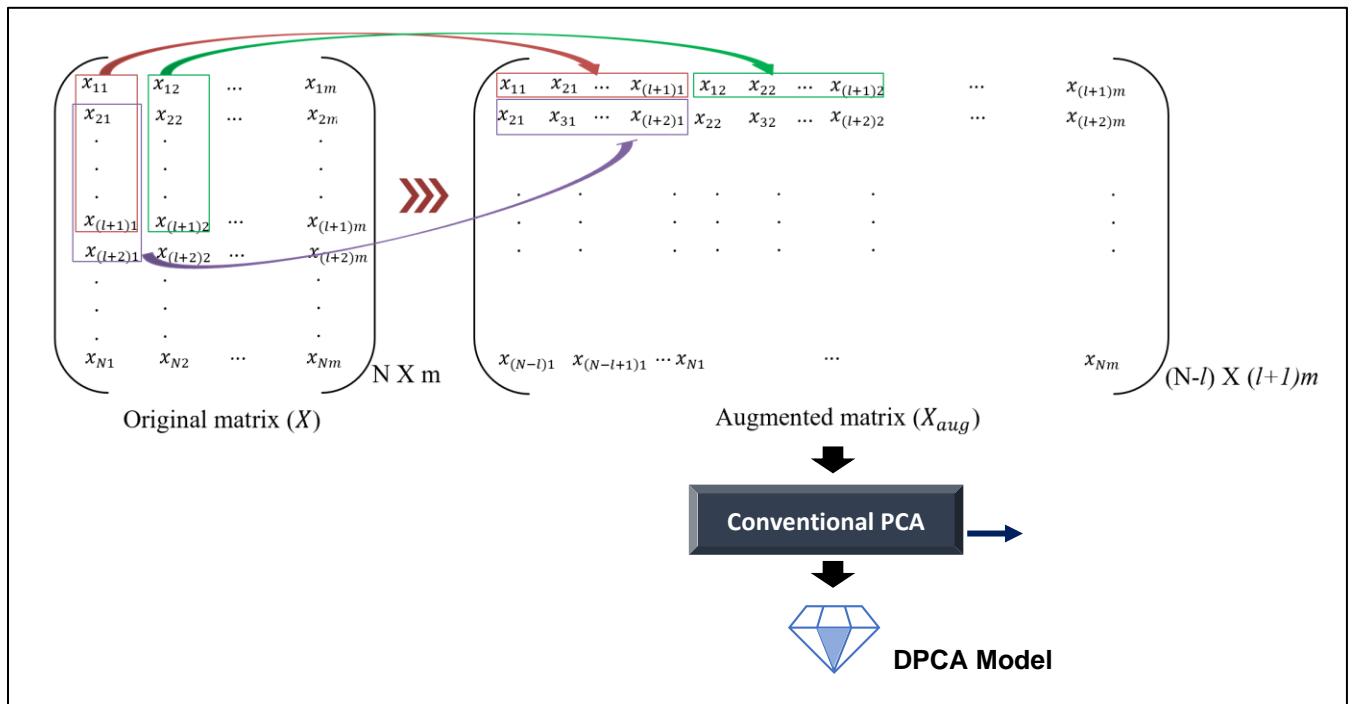


Figure 9.1: Dynamic PCA procedure [l denotes the number of lags used]

All the mathematical expressions for the computations of the score matrix⁵⁵, residual matrix, Hotelling’s T^2 , and SPE remain the same (Eq. 1 to Eq. 8) as that shown in Chapter 7, except that now you will be using scaled X_{aug} instead of X , i.e., $T = X_{aug}P$; $E = X_{aug} - \hat{X}_{aug}$. The procedure for determination of number of retained latent variables also remains the same. You may, amongst other approaches, look for a ‘knee’ in the scree plot of the explained variance or use the cumulative percent variance approach. If you choose ‘ l ’ correctly, then both the static and dynamic relationships among process variables are captured and correspondingly, the residuals and the Q statistic will not exhibit autocorrelations⁵⁶. For test dataset, you would again simply perform augmentation with time-lagged measurements. Before we get into the nitty-gritties, let’s see a quick motivating example on why DPCA is superior to PCA in the presence of dynamics.

⁵⁵ The number of retained principal components in DPCA could be greater than m .

⁵⁶ The DPCA scores can show autocorrelations.

Example 9.1:

To illustrate how DPCA can extract dynamic relationships, let's consider the following noise-free dynamic system.

$$x_1(k) = 0.8x_1(k - 1) + x_2(k - 1); \quad k \text{ is sampling instant}$$

The number of zero singular (eigen) values extracted during PCA indicates the number of linear relationships that exist among the process variables. Let's see if we can extract out the above dynamic relationship using only data (1000 samples of x_1 and x_2).

```
# import required packages
import numpy as np, matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# generate data for the system: x1(k) = 0.8*x1(k-1) + x2(k-1)
x2 = np.random.normal(loc=0, scale=1, size=(1000,1))
x1 = np.zeros((1000,1))
for k in range(1,1000):
    x1[k] = 0.8*x1[k-1] + x2[k-1]
X = np.hstack((x1, x2))

# function to generate augmented matrix
def augment(X, n_lags):
    N, m = X.shape
    X_aug = np.zeros((N-n_lags, (n_lags+1)*m))
    for sample in range(n_lags, N):
        XBlock = X[sample-n_lags:sample+1,:]
        X_aug[sample-n_lags,:] = np.reshape(XBlock, (1,-1), order = 'F')
    return X_aug

# fit DPCA model
X_aug = augment(X, 1) # augment data
X_aug_centered = X_aug - np.mean(X_aug, axis=0) # center data
dPCA = PCA().fit(X_aug_centered) # fit PCA model
print('DPCA singular values:', dPCA.singular_values_) # get singular values

>>> DPCA singular values: [6.664e+01 3.731e+01 3.094e+01 1.1661e-14]
```

As expected, only one singular value is very close to zero. All we now need to do is fetch the singular vector corresponding to this singular value and check if it represents our dynamic system.

```
# get 4th singular vector
print('4th singular vector: ', dPCA.components_[3,:])

>>> 4th singular vector: [ 4.923e-01 -6.154e-01  6.154e-01  1.7348e-17]
```

The 4th singular vector represents the following relation

$$\begin{aligned} 0.4923x_1(k-1) - 0.6154x_1(k) + 0.6154x_2(k) &= 0 \\ \Rightarrow x_1(k) &= 0.8x_1(k-1) + x_2(k-1) \end{aligned}$$

Voila! DPCA has successfully extracted the underlying process dynamics. PCA on the other hand does not reveal any relationship between the variables.

Time lag (l) determination

The time lag (l) is decided by the dynamic order of the process. Unfortunately, the dynamic order is usually unknown. An algorithm for time lag determination was first proposed by Ku et al.⁵⁷ in 1995. Another commonly used strategy was given by Rato and Reis⁵⁸, and is reproduced below.

Algorithm: Time lag determination for DPCA

- 1) Initialize $l = 0$ and specify some maximum value of lag l_{max}
- 2) Generate the augmented data matrix X_{aug}
- 3) Perform singular value decomposition of covariance matrix ($\Sigma_{X_{aug}}$) of X_{aug} . Let s_{ml+1} be the $(ml + 1)^{th}$ singular value of $\Sigma_{X_{aug}}$. Define $KSV(l) = s_{ml+1}$.
- 4) Define $KSVR(l) = KSV(l) / KSV(l-1)$ for $l > 0$
- 5) If $l < l_{max}$, set $l = l + 1$ and go to step 2, else go to step 6
- 6) Normalize KSV and KSVR
$$KSV_{normal}(l) = \frac{KSV(l) - \min(KSV)}{\max(KSV) - \min(KSV)}$$

$$KSVR_{normal}(l) = \frac{KSVR(l) - \min(KSVR)}{\max(KSVR) - \min(KSVR)}$$
- 7) Find l^* as the first l such that $KSVR(l) < KSVR(l-1)$
- 8) Determine the optimal $l_{opt} \in [1, l_{max}]$ as

$$l_{opt} = \operatorname{argmin} \sqrt{KSV_{normal}(l)^2 KSVR_{normal}(l)^2}, \quad \text{s.t. } l \geq l^*$$

⁵⁷ Ku et al., Disturbance detection and isolation by dynamic principal component analysis. *Chemometrics and intelligent laboratory systems*, 1995.

⁵⁸ Rato and Reis, Defining the structure of DPCA models and its impact on process monitoring and prediction activities. *Chemometrics and intelligent laboratory systems*, 2013

9.2 Fault Detection via DPCA: Numerical Illustration

To illustrate each step of DPCA-based fault detection, we will use a simple synthetic 2 input, 2 output process⁵⁹ as shown below

$$\begin{aligned} z(k) &= \begin{bmatrix} 0.118 & -0.191 \\ 0.847 & 0.264 \end{bmatrix} z(k-1) + \begin{bmatrix} 1 & 2 \\ 3 & -4 \end{bmatrix} u(k-1) \\ u(k) &= \begin{bmatrix} 0.811 & -0.226 \\ 0.477 & 0.415 \end{bmatrix} u(k-1) + \begin{bmatrix} 0.193 & 0.689 \\ -0.320 & -0.749 \end{bmatrix} w(k-1) \\ y(k) &= z(k) + v(k) \end{aligned}$$

Here, the outputs ($y(k)$) and inputs ($u(k)$) measurements are available for analysis. $w(k)$ and $v(k)$ are zero-mean random noise signals with variances 1 and 0.1, respectively. Simulated data from the process are available in files *multivariate_NOC_data.txt* and *multivariate_test_data.txt*. In test data, A disturbance with unit step change of w_2 was introduced at sample 50 to simulate a faulty condition.

We will now go through each step of building a DPCA-based fault detection solution. Let's begin by reading the datasets, generating the augmented data matrices, and then training the DPCA model.

```
# fetch data
X_NOC = np.loadtxt('multivariate_NOC_data.txt')
X_test = np.loadtxt('multivariate_test_data.txt')

%%%%%
# DPCA model training
%%%%%
# augment and scale data
X_NOC_aug = augment(X_NOC, 1)
scaler = StandardScaler()
X_NOC_aug_scaled = scaler.fit_transform(X_NOC_aug)

# fit PCA model on augmented data
dPCA = PCA().fit(X_NOC_aug_scaled)

# find n_component
explained_variance = 100*dPCA.explained_variance_ratio_ # in percentage
```

⁵⁹ used by Ku et al. in their seminal paper.

```

cum_explained_variance = np.cumsum(explained_variance) # cumulative % variance explained
n_comp = np.argmax(cum_explained_variance >= 95) + 1

# refit with n_comp
dPCA = PCA(n_components=n_comp)
dPCA.fit(X_NOC_aug_scaled)

# compute scores and error for training data
scores_NOC = dPCA.transform(X_NOC_aug_scaled)
X_NOC_aug_scaled_reconstruct = dPCA.inverse_transform(scores_NOC)
error_NOC = X_NOC_aug_scaled - X_NOC_aug_scaled_reconstruct

# calculate Q for training data
Q_NOC = np.sum(error_NOC*error_NOC, axis = 1)

# calculate T2 for training data
N = X_NOC_aug_scaled.shape[0]
lambda_k = np.diag(dPCA.explained_variance_) # eigenvalue = explained variance
lambda_k_inv = np.linalg.inv(lambda_k)

T2_NOC = np.zeros((N,))
for i in range(N):
    T2_NOC[i] = np.dot(np.dot(scores_NOC[i,:], lambda_k_inv), scores_NOC[i,:].T)

# control limits
Q_CL = np.percentile(Q_NOC, 99)
T2_CL = np.percentile(T2_NOC, 99)

```

Let's now deploy the DPCA model on the test data and see if the monitoring statistics violate the control limits.

```

# augment and scale test data
X_test_aug = augment(X_test, 1)
X_aug_test_scaled = scaler.transform(X_test_aug)

# compute scores and error for test data
scores_test = dPCA.transform(X_aug_test_scaled)
X_aug_test_scaled_reconstruct = dPCA.inverse_transform(scores_test)
error_test = X_aug_test_scaled - X_aug_test_scaled_reconstruct

# calculate Q and T2 for test data
Q_test = np.sum(error_test*error_test, axis = 1)

```

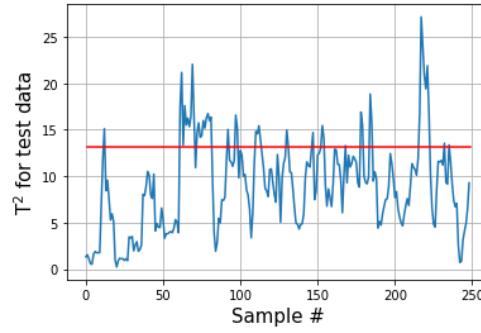
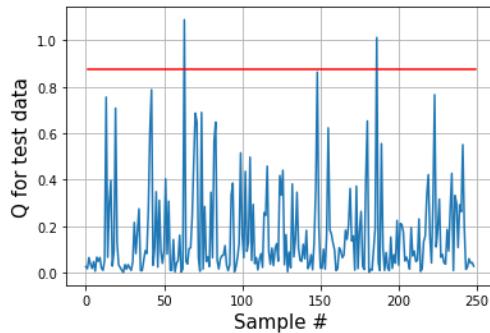
```

N_test = X_aug_test_scaled.shape[0]
T2_test = np.zeros((N_test,))
for i in range(N_test):
    T2_test[i] = np.dot(np.dot(scores_test[i,:], lambda_k_inv), scores_test[i,:].T)

# monitoring charts for test data
plt.figure(), plt.plot(Q_test), plt.plot([1,len(Q_test)],[Q_CL,Q_CL], color='red')
plt.xlabel('Sample #'), plt.ylabel('Q for test data')

plt.figure(), plt.plot(T2_test), plt.plot([1,len(T2_test)],[T2_CL,T2_CL], color='red')
plt.xlabel('Sample #'), plt.ylabel('T$^2$ for test data')

```



You can see that DPC-based fault detection procedure is the same as that for PCA-based fault detection, except for the augmentation of data. In general, DPCA has been found to perform better than PCA when temporal correlation is present in data.

Contribution plots for fault diagnosis

The T^2 and Q contributions for each variable of the augmented matrix are computed in the same way as done for PCA. The overall contribution for an original variable is obtained by summing up the contributions for the variable and each of its l time lagged variables.

9.3 Dynamic PLS: An Introduction

Dynamic PLS is the dynamic extension of conventional PLS. As shown in Figure 9.2, DPLS entails employment of lagged measurements as additional predictor variables. As shown, there are two ways of generating a DPLS model: by including lagged values of only input variables or both input and output variables. While the first approach leads to FIR type of model, the later approach leads to ARX type of model. Like DPCA, once the augmented matrix is ready, scaling is performed and conventional PLS is applied. All the mathematical expressions for the computations of the score matrix, residual matrices, monitoring metrics, etc., remain the same as that seen in Chapter 7 for conventional PLS.

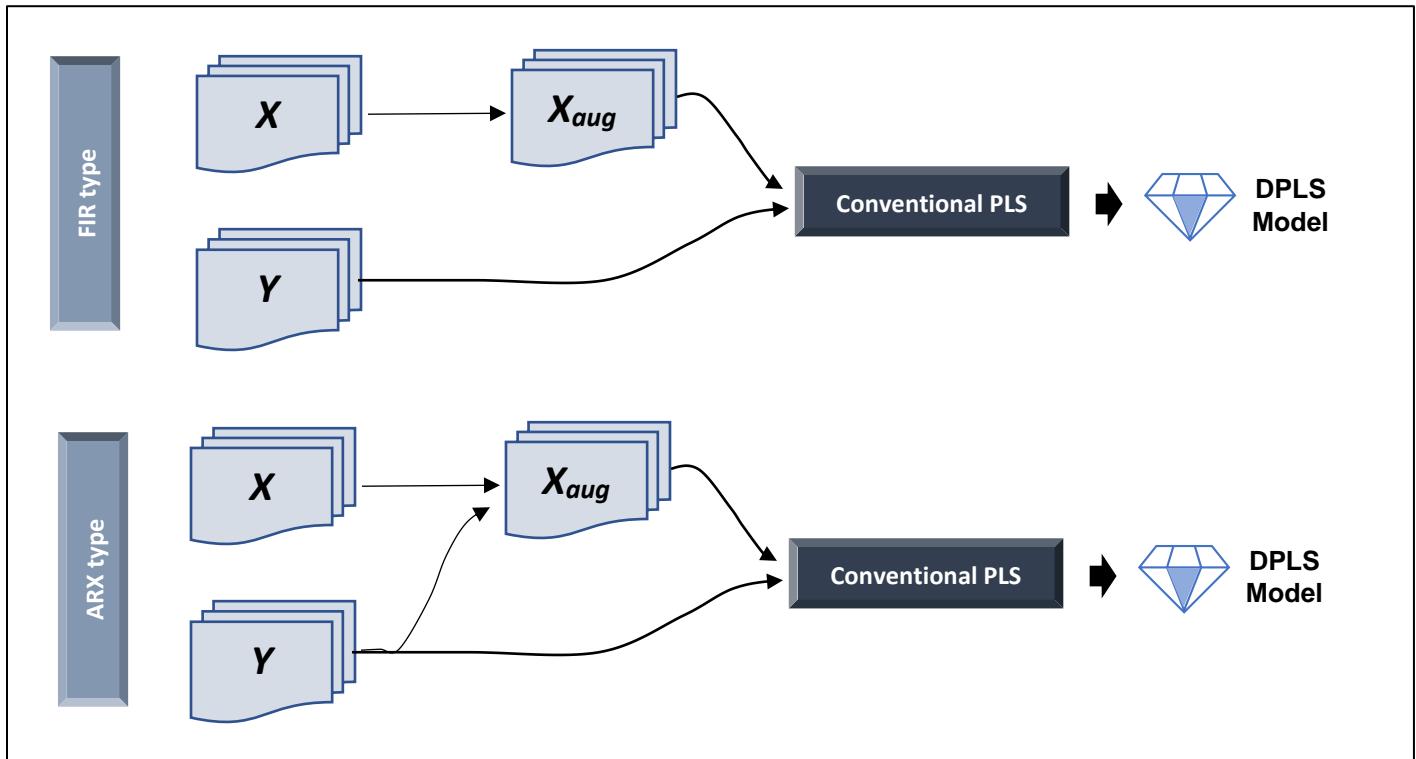


Figure 9.2: Dynamic PLS procedure

There are two hyperparameters that need to be specified: the time lags⁶⁰ and the number of latents retained. Both these hyperparameters are commonly found via cross-validation. We have already seen how to modify the conventional FDI workflow to handle augmented data matrices in DPCA and therefore, we will skip showing the implementation code for DPLS-based FDI. Instead, we will move on to study a slightly different technique (state-space modeling) for building dynamic models.

⁶⁰ Time lags are often chosen to be the same for all the input (and output if building ARX type model) variables

9.4 Canonical Variate Analysis: An Introduction

Canonical variate analysis (CVA) is a classical MSPM technique suited for modeling dynamic process systems. While PCA and PLS lead to input-output models, CVA is designed to provide state-space models as shown in Figure 9.3. States are like latent variables which act as a conduit between process inputs and outputs. For example, in the distillation column system, it is clear that Q_B doesn't affect x_D directly. There are internal dynamics or states (e.g., arising from the liquid holdup at each column tray) that changes in Q_B have to pass through before impacting x_D . Similar argument could be made for Q_B vs D relationship. The outputs share the same internal dynamics and this leads to better parameter efficiency of CVA model (compared to DPCA and DPLS model)

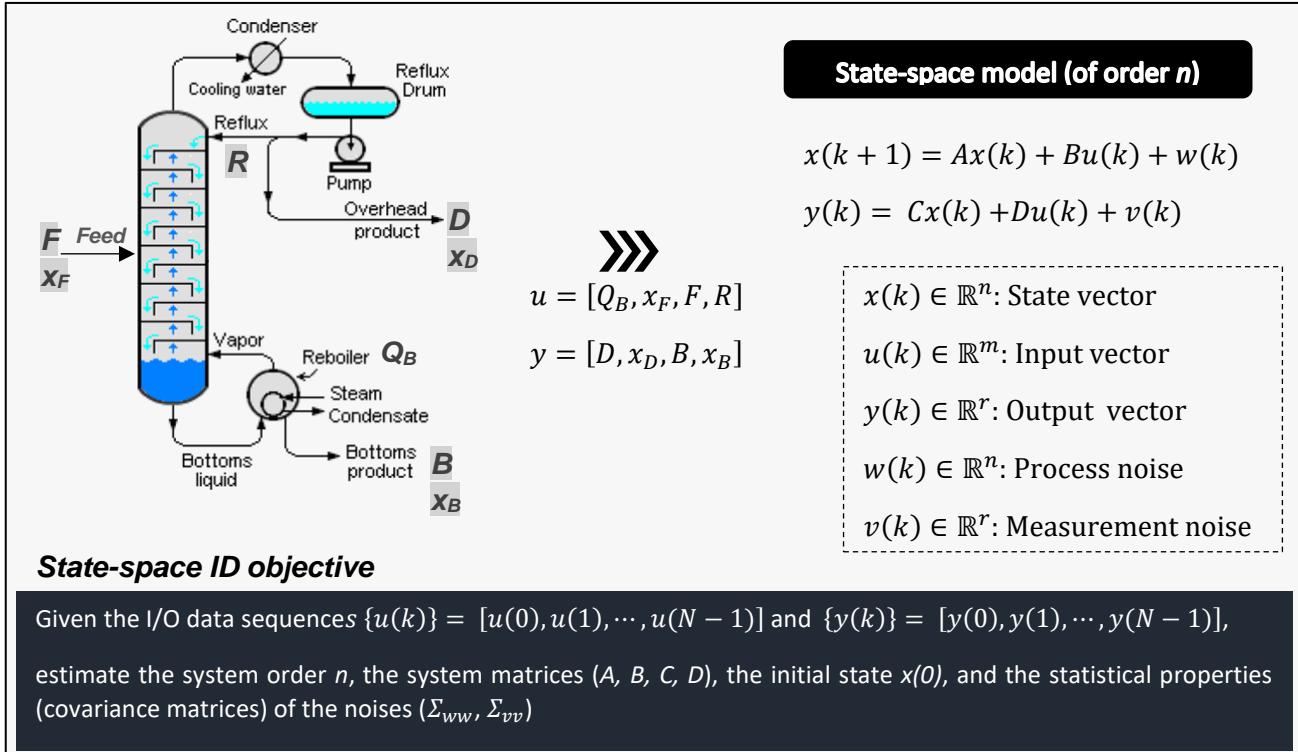


Figure 9.3: State-space representation of a distillation process

For CVA-based process monitoring applications, we do not need the system matrices; only the computation of states for training and test datasets suffices. Tracking the systematic variations of states and state noise allow us to build monitoring metrics. As it turns out, the state vector at any time can be approximated as a linear combination of past inputs and outputs. The specific combination is found as the one that gives the best prediction of future outputs. Let's now familiarize ourselves on how this is achieved.

Mathematical background

Using the state-space model and some smart re-arrangement⁶¹ of input-output data sequence, it can be shown that state vector $x(k)$ at time instant k can be approximated as a linear combination of past inputs and outputs

$$\hat{x}(k) = J p(k) \quad \text{eq. 1}$$

↑ unknown matrix
↓
defined as $[y^T(k-1) \ y^T(k-2) \ \dots \ y^T(k-l_y) \ u^T(k-1) \ u^T(k-2) \ \dots \ u^T(k-l_u)]^T$

In Eq. 1, l_y and l_u denote the number of lagged outputs and inputs used to build the $p(k)$ vector. Usually, $l_y = l_u = l$ and it is a model hyperparameter. The CVA method utilizes CCA (canonical covariate analysis), a multivariate statistical technique, to find J . A quick primer on the CCA technique is provided below.

CCA: A quick primer

CCA is a dimensionality reduction technique which aims to find linear combinations of input variables ($u \in \mathbb{R}^m$) that are maximally correlated with linear combinations of output variables ($y \in \mathbb{R}^r$). You may relate this to PLS wherein covariance between latent variables is maximized. These linear combinations are called canonical variables and are given as

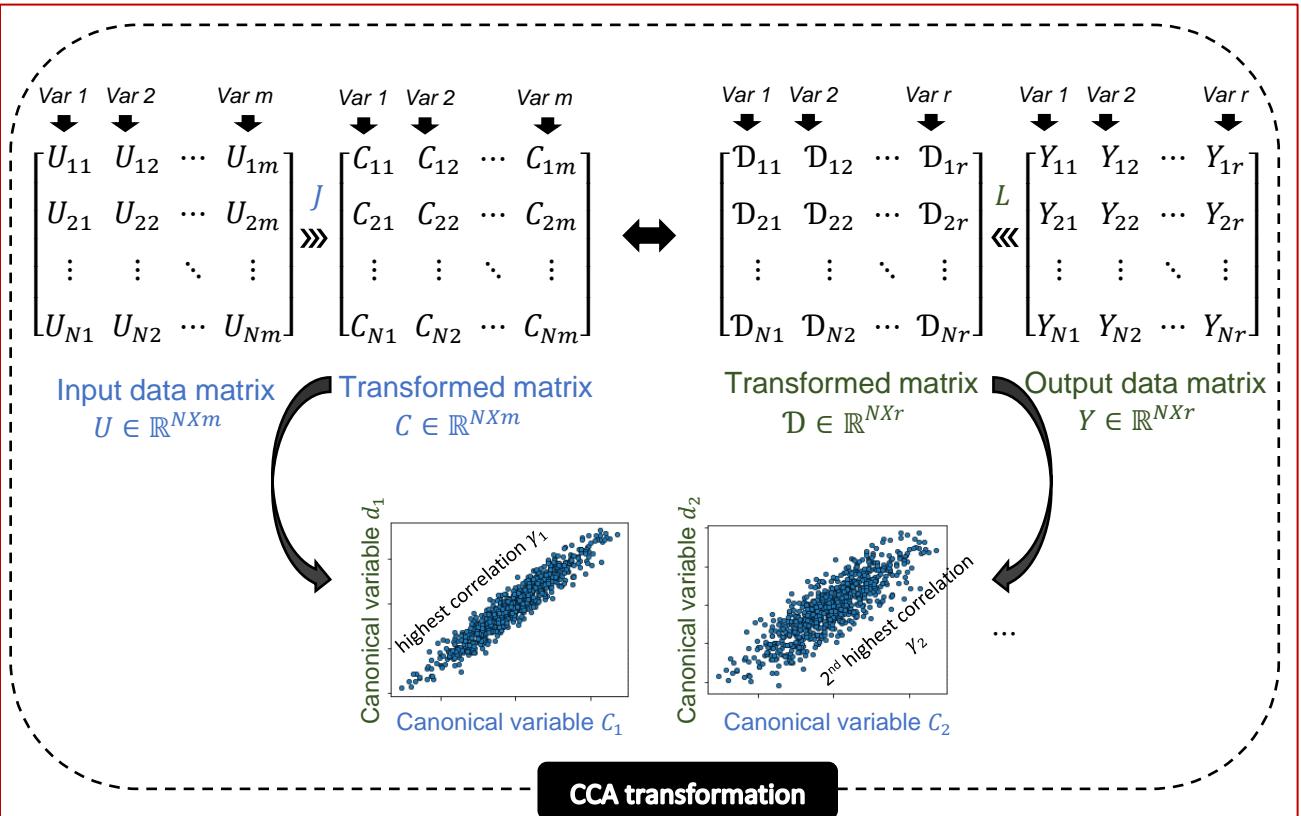
Observations in respective
latent spaces ↗ $c = Ju$ ← c comprises of uncorrelated latent variables
 ↗ $d = Ly$ ← d comprises of uncorrelated latent variables

The covariance matrix between c and d is (rectangular) diagonal

$$\Sigma_{cd} = J \Sigma_{uy} L^T = D = \text{diag}(\gamma_1, \gamma_2, \dots, \gamma_o, 0, \dots, 0) \quad \begin{matrix} \text{latent variables are only} \\ \text{pairwise correlated} \end{matrix}$$

The elements γ_i of matrix D are called canonical correlations with ($\gamma_1 \geq \gamma_2 \geq \dots \geq \gamma_o$) and quantify the degree of correlations between the i^{th} canonical variate pair c_i and d_i .

⁶¹ Chen et. al., A canonical variate analysis based process monitoring scheme and benchmark study. 2014



The matrices $J \in \mathbb{R}^{m \times m}$ and $L \in \mathbb{R}^{r \times r}$ are found via singular value decomposition

$$\Sigma_{uu}^{-1/2} \Sigma_{uy} \Sigma_{yy}^{-1/2} = U \Sigma V^T$$

$$\text{where, } J = U^T \Sigma_{uu}^{-1/2}, L = V^T \Sigma_{yy}^{-1/2}, D = \Sigma$$

One may retain only those canonical variate pairs (say, top n pairs) that show significant correlations and discard the others, and therefore achieve dimensionality reduction. Let \hat{c} and \hat{d} be the n -dimensional ($n < m$ and r) vectors, then,

$$\begin{aligned} \hat{c} &= J_n u && \text{where, } J_n \text{ and } L_n \text{ are first } n \text{ rows of } J \text{ and } L \\ \hat{d} &= L_n y \end{aligned}$$

* It is assumed that u and y are centered. Scaling, however, is not required as CCA is scale-invariant, i.e., the estimated latent variables and the correlations remain the same with or without scaling. Also note that we have not used the 'k' notation as CCA is designed for static data.

Computing CVA states

In CVA, the two sets of vectors between which CCA is performed are $p(k)$ (defined in Eq. 1) and $f(k)$ where⁶²

$$f(k) = [y^T(k) \ y^T(k+1) \ \dots \ y^T(k+l)]^T$$

 current and future outputs

CVA therefore solves the following singular value decomposition problem and finds linear combinations of past inputs and outputs that are maximally correlated with linear combinations of the current and future outputs.

$$\Sigma_{pp}^{-1/2} \Sigma_{pf} \Sigma_{ff}^{-1/2} = U \Sigma V^T \quad \text{eq. 2}$$

We know that Σ is a diagonal matrix. Its elements are also called (Hankel) singular values. If n is assumed to be the model order, then the optimal state at time k that is most predictive of the future outputs is given by⁶³

$$\hat{x}(k) = J_n p(k) \quad \text{eq. 3}$$

 Contains first n rows of matrix $J = U^T \Sigma_{pp}^{-1/2}$

Eq. 3 can be used to find the states for the training and test datasets.

Hyperparameter selection

The mathematical background of the CVA algorithm suggests specification of two hyperparameters: the lag order (ℓ) and the model order (n). While model order specification is not required prior to state estimation, the lag order needs to be known beforehand. Let's see the common approaches for estimation of these hyperparameters.

Lag order (l)

The guidance for assigning a value to the lag l is that it should be large enough to capture data autocorrelation. A commonly recommended strategy is to fit several multivariate autoregressive models with different values of l and choose the l that minimizes the AIC

⁶² The number of lead values used to define $f(k)$ is commonly taken equal to the number of lags used in $p(k)$

⁶³ Alternatively, $\hat{x}(k) = U_n^T \Sigma_{pp}^{-1/2} p(k)$ where U_n contains the first n columns of U .

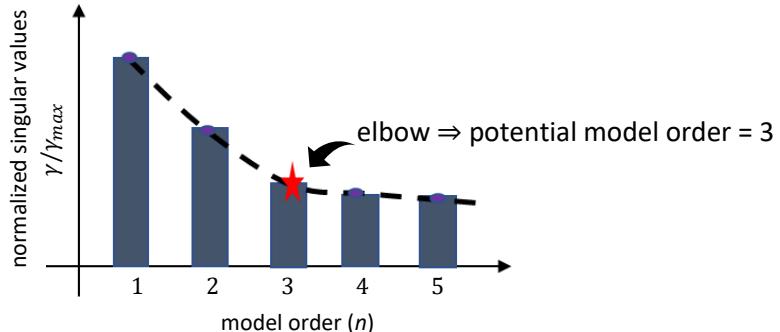
criterion. An alternative approach was suggested by Odiowei and Cao⁶⁴ who suggest checking the autocorrelation function of the summed squares of the process variables and choosing the maximum value of lag for which the autocorrelation is significant as the lag order. Another convenient approach was given by Zhang et al.⁶⁵ who suggest choosing lag order as the minimum s that satisfies the following equation

$$A_{s+1} = \frac{\sqrt{\sum_{j=1}^M (\text{autocorr}(x_j, s+1))^2}}{\sqrt{\sum_{i=1}^s \sum_{j=1}^M (\text{autocorr}(x_j, i))^2} / s} \leq \vartheta$$

Where $\text{autocorr}(x_j, i)$ is the autocorrelation coefficient of the j^{th} variable at time lag i . The threshold is chosen between 0 and 0.5.

Model order (n)

Correct selection of the value for n is important: while too large a value leads to model overfitting, too small value for n leads to underfitting. One approach for model order selection is based on Hankel singular values. If the plot of the singular values vs model order has a prominent 'elbow', then the model order corresponding to the elbow can be chosen as shown below



Another practice is to choose n such that the normalized values of the $(n+1)^{\text{th}}$ onwards singular values are below some threshold. However, very often there is no elbow or the singular values decrease slowly. Therefore, the more extensively used criteria for model order selection is AIC⁶⁶.

⁶⁴ Odiowei & Cao, Nonlinear Dynamic Process Monitoring using Canonical Variate Analysis and Kernel Density Estimations. *IEEE Trans*, 2009.

⁶⁵ Zhang et al., Simultaneous static and dynamic analysis for fine-scale identification of process operation statuses. 2019.

⁶⁶ Let \hat{n} be a trial model order in some range 0 to n_{\max} . The outputs $\hat{y}(k)$ are predicted using the fitted state space model and the AIC is computed. The value of \hat{n} that gives minimum AIC becomes the optimal model order.

Process monitoring/fault detection indices

In CVA-based fault detection strategy, 3 indices - T_s^2, T_e^2, Q - are computed. The computation of these metrics and their corresponding threshold are shown below. Note that the threshold expressions are based on the assumption that process variables are Gaussian distributed; if this assumption is invalid then empirical techniques such as KDE or percentile may be employed.

Metric 1: T_s^2	Metric 2: T_e^2	Metric 3: Q
$T_s^2(k) = x^T(k)x(k)$ $= p^T(k) J_n^T J_n p(k)$ Threshold @ significance level α $T_{s,CL}^2 = \frac{n(N^2 - 1)}{N(N - n)} F_\alpha(n, N - n)$	$T_e^2(k) = p^T(k) J_e^T J_e p(k)$ Threshold @ significance level α $T_{e,CL}^2 = \frac{z(N^2 - 1)}{N(N - z)} F_\alpha(z, N - z)$	$Q(k) = r^T(k)r(k)$ $r(k) = p(k) - J_n^T J_n p(k)$ Threshold @ significance level α $Q_{CL} = \frac{\sigma}{2\mu} \chi_\alpha^2 \left(\frac{2\mu^2}{\sigma} \right)$

- J_e contains all except the first n rows of J i.e., the last $\ell(m + r) - n$ rows of J
- $z = \ell(m + r) - n$
- $F_\alpha(n, N-n)$ is the $1-\alpha$ percentile of a F distribution with n and $N-n$ degrees of freedom
- $\chi_\alpha^2(h)$ is the $(1-\alpha)$ percentile of a chi-squared distribution with h degrees of freedom; μ denotes the mean value and σ denotes the variance of the Q metric
- Significance level of $\alpha = 0.05$ would mean that there is a 5% chance that an alert is false

(Unsupervised) CVA modeling with only output variables



CVA can also be used to capture dynamic correlations in datasets where the variables are not divided into input and outputs. The state-space model that results is the following

$$x(k + 1) = Ax(k) + w(k)$$

$$y(k) = Cx(k) + v(k)$$

Such an approach can be used to monitor a process operating around a stable point. The overall approach remains the same (as shown in Figure 9.4) except for the definition of past and future vectors which are now defined as follows

$$p(k) = [y^T(k - 1) \ y^T(k - 2) \ \dots \ y^T(k - l)]^T$$

$$f(k) = [y^T(k) \ y^T(k + 1) \ \dots \ y^T(k + l)]^T$$

The flowchart (Figure 9.4) below summarizes the aforementioned steps for development of CVA-based process fault detection solution.

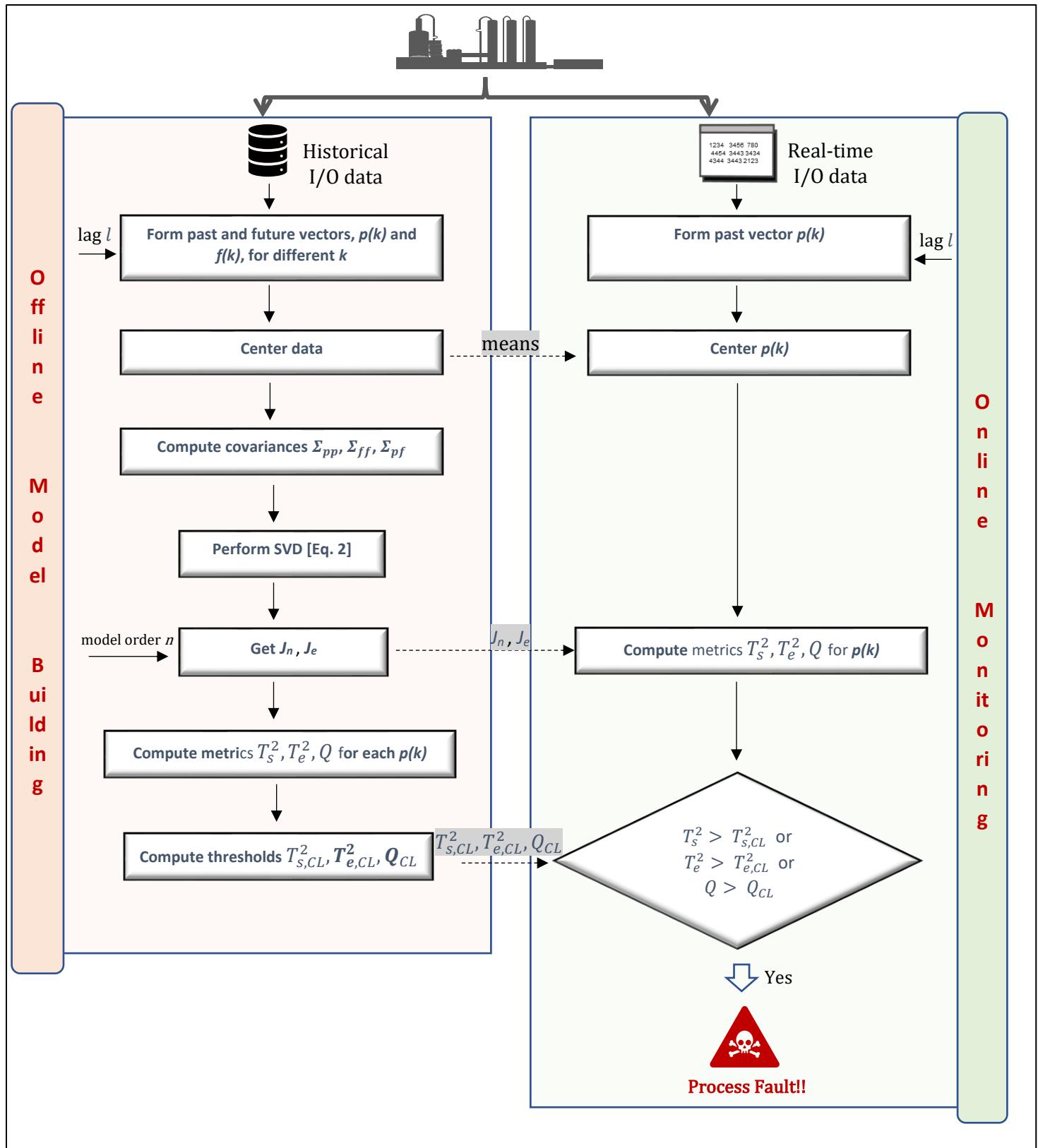


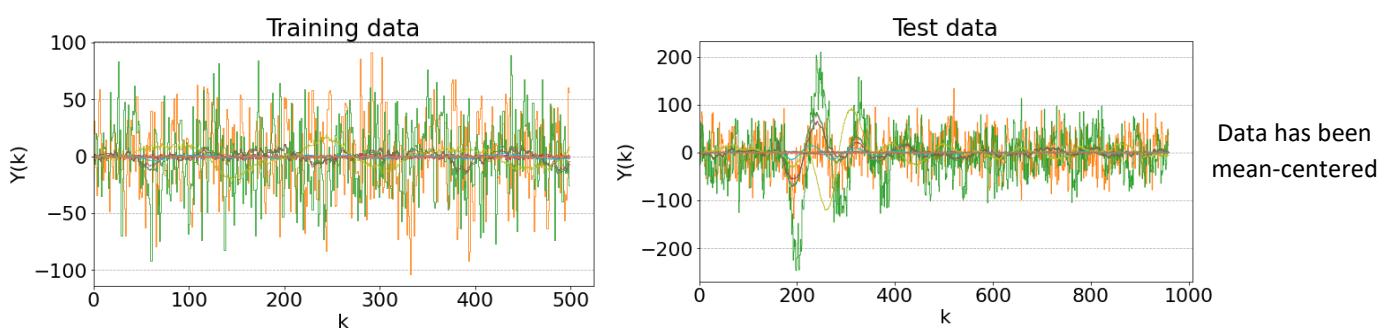
Figure 9.4: Complete workflow for process monitoring via CVA

9.5 Process Monitoring of Tennessee Eastman Process via CVA

In this section, we will learn the details of CVA-based monitoring tool development through step-by-step application for our now familiar TEP process. The CVA monitoring procedures provided in this book mirror the description in the work⁶⁷**Error! Bookmark not defined.** of Russell et al., and we will therefore attempt to replicate the CVA-based fault detection results from their journal paper titled '*Fault detection in industrial processes using canonical variate analysis and dynamic principal component analysis*'. Specifically, we will use the *Fault 5* dataset (file *d05_te.dat*) as our test data and normal operation dataset (file *d00.dat*) as our training data. We will use the manipulated variables as our inputs and the process measurements as our outputs. Our objective is to build a monitoring tool that can accurately flag the presence of a process fault with low frequency of false alerts. Let's begin with a quick exploration of the dataset.

```
# import packages
import matplotlib.pyplot as plt, numpy as np
from sklearn.preprocessing import StandardScaler

# read data and separate input-output variables
trainingData = np.loadtxt('d00.dat').T
FaultyData = np.loadtxt('d05_te.dat')
uData_training, yData_training = trainingData[:,41:52], trainingData[:,0:22]
uData_test, yData_test = FaultyData[:,41:52], FaultyData[:,0:22]
```



It is obvious that continuously monitoring all the 22 output (and 11 input) variables is not practical. The single combined output plots above do seem to clearly indicate a process upset around sample number 200 in faulty dataset; however, they also seem to give a wrong

⁶⁷ Russell et al., Data-Driven Methods for Fault Detection and Diagnosis in Chemical Processes. Springer, 2001.

Russell et al., Fault detection in industrial processes using canonical variate analysis and dynamic principal component analysis. *Chemometrics and intelligent laboratory systems*, 2000

impression of normal process conditions sample number 500 onwards. It is known that Fault 5 continues until the end of the dataset. Let's compute the monitoring indices for the training dataset.

```

#####
## generate past (p) and future (f) vectors for training dataset and center them
#####
N = trainingData.shape[0]
l = 3 # as used by Russell et. al
m = uData_training.shape[1]
r = yData_training.shape[1]

pMatrix_train = np.zeros((N-2*l, l*(m+r)))
fMatrix_train = np.zeros((N-2*l, (l+1)*r))
for i in range(l, N-l):
    pMatrix_train[i-l,:] = np.hstack((yData_training[i-l:i,:].flatten(), uData_training[i-l:i,:].flatten()))
    fMatrix_train[i-l,:] = yData_training[i:i+l+1,:].flatten()

# center data
p_scaler = StandardScaler(with_std=False); pMatrix_train_centered = p_scaler.fit_transform(pMatrix_train)
f_scaler = StandardScaler(with_std=False); fMatrix_train_centered = f_scaler.fit_transform(fMatrix_train)

#####
## computes covariances and perform SVD
#####
import scipy

sigma_pp = np.cov(pMatrix_train_centered, rowvar=False)
sigma_ff = np.cov(fMatrix_train_centered, rowvar=False)
sigma_pf = np.cov(pMatrix_train_centered, fMatrix_train_centered,
                  rowvar=False)[:len(sigma_pp),:len(sigma_pp)]
matrixProduct = np.dot(np.dot(np.linalg.inv(scipy.linalg.sqrtm(sigma_pp).real), sigma_pf),
                      np.linalg.inv(scipy.linalg.sqrtm(sigma_ff).real))
U, S, V = np.linalg.svd(matrixProduct)
J = np.dot(np.transpose(U), np.linalg.inv(scipy.linalg.sqrtm(sigma_pp).real))

# get the reduced order matrices
n = 29 # as used by Russell et. al
Jn, Je = J[:n,:], J[n:,:]

#####
## get fault detection metrics for training data
#####

```

```

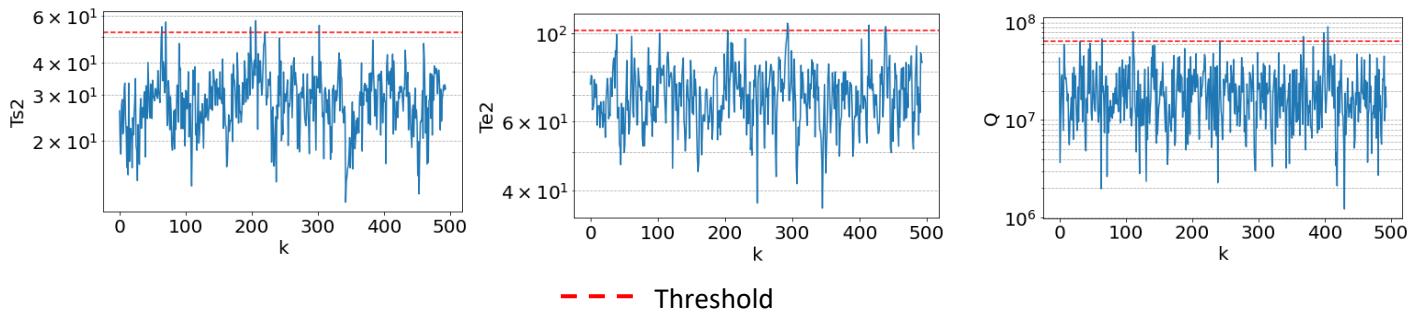
#####
Xn_train = np.dot(Jn, pMatrix_train_centered.T)
Ts2_train = [np.dot(Xn_train[:,i], Xn_train[:,i]) for i in range(pMatrix_train_centered.shape[0])]

Xe_train = np.dot(Je, pMatrix_train_centered.T)
Te2_train = [np.dot(Xe_train[:,i], Xe_train[:,i]) for i in range(pMatrix_train_centered.shape[0])]

r_train = pMatrix_train_centered.T - np.dot(Jn.T, Xn_train)
Q_train = [np.dot(r_train[:,i], r_train[:,i]) for i in range(pMatrix_train_centered.shape[0])]

# determine thresholds as 99th percentile of respective metrics
Ts2_CL = np.percentile(Ts2_train, 99)
Te2_CL = np.percentile(Te2_train, 99)
Q_CL = np.percentile(Q_train, 99)

```



The T_s^2 metric measures the systematic variations ‘inside’ the state space defined by the state variables. Correspondingly, T_r^2 quantifies the variations ‘outside’ the state space. When only T_s^2 violates the threshold, it indicates that the state variables are ‘out-of-control’ but the process can still be well explained by the estimated state-space model. When T_s^2 or Q violates the threshold, it indicates that the characteristics of noise affecting the system has changed or the estimated model cannot explain the new faulty observations.

Fault detection on test data

It’s time now to check whether our monitoring charts can help us detect the presence of process abnormalities. For this, we will compute the monitoring statistics for the test data.

```

#####
## generate past vectors (p(k)) for test dataset and center
#####
Ntest = FaultyData.shape[0]

pMatrix_test = np.zeros((Ntest-l+1, l*(m+r)))

```

```

for i in range(l,Ntest+1):
    pMatrix_test[i-l,:] = np.hstack((yData_test[i-l:i,:].flatten(), uData_test[i-l:i,:].flatten()))

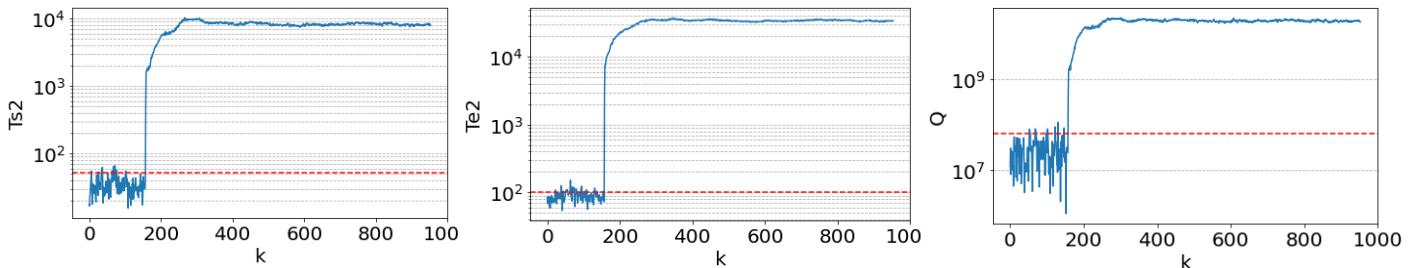
pMatrix_test_centered = p_scaler.transform(pMatrix_test)

#####
## get fault detection metrics for training data
#####
Xn_test = np.dot(Jn, pMatrix_test_centered.T)
Ts2_test = [np.dot(Xn_test[:,i], Xn_test[:,i]) for i in range(pMatrix_test_centered.shape[0])]

Xe_test = np.dot(Je, pMatrix_test_centered.T)
Te2_test = [np.dot(Xe_test[:,i], Xe_test[:,i]) for i in range(pMatrix_test_centered.shape[0])]

r_test = pMatrix_test_centered.T - np.dot(Jn.T, Xn_test)
Q_test = [np.dot(r_test[:,i], r_test[:,i]) for i in range(pMatrix_test_centered.shape[0])]

```



The control charts for the test data shows successful detection of process fault. The T_e^2 metric is known to be overly sensitive with high incidence of false alerts which can explain why it seems to often breach the threshold before the onset of the actual fault.

Summary

In this chapter, we added more ML tools to our arsenal to handle process systems that show significant dynamics. We looked at two different popular approaches: the first approach entailed a simple extension of conventional MSPM tools through inclusion of lagged measurements and the second approach entailed usage of state-space (CVA) models. While dynamic MSPM methods are easier to understand and implement, the CVA model-based fault detection have been found to provide superior results. Next, we will move on to learn how to deal with nonlinear processes.

Chapter 10

Multivariate Statistical Process Monitoring for Nonlinear Processes

In the previous chapters, we saw how a simple trick of using time-lagged variables enabled application of conventional MSPM techniques to dynamic processes. You may wonder if anything similar exists for nonlinear processes. Fortunately, it does! The underlying principle is to project the original variables onto a high-dimensional feature space where features are linearly related. The challenging part is the determination of the nonlinear mapping from the original measurement space to the feature space. This is where a ‘kernel’ trick comes into picture wherein data gets projected without the need to explicitly define the nonlinear mapping. Conventional MSPM is then employed in the feature space. Sounds complicated? Once you are done with this chapter, you will realize that it’s much easier than it may seem to you right now.

The main advantage of kernel-based MSPM techniques (KPCA, KPLS, KICA, KFDA, etc.) is that they do not require nonlinear optimization and only linear algebra is involved. Unsurprisingly, kernelized methods have become very attractive for dealing with nonlinear datasets while retaining the simplicity of their linear counterparts. Among the kernel MSPM techniques, kernel PCA and kernel PLS are the most widely adopted, have found considerable successes in process monitoring applications, and therefore will be the focus of our study in this chapter. Specifically, the following topics are covered

- Introduction to kernel PCA
- Fault detection using kernel PCA
- Introduction to kernel PLS
- Fault detection using kernel PLS

10.1 Kernel PCA: An Introduction

Kernel PCA is the nonlinear extension of conventional PCA suitable for handling processes that exhibit significant nonlinearity. To understand the motivation behind KPCA, consider the simple scenarios illustrated in Figure 10.1. In Figure 10.1a, we can see that the data lie along a line which can be obtained from the first eigenvector of linear PCA. In Figure 10.1b, data lie along a curve; conventional PCA cannot help to find this nonlinear curve. Correspondingly, PCA fails to detect the obvious outlier. However, all is not lost for the latter scenario. Instead of working in the (x_1, x_2) measurement space, if we work in the $(z_1, z_2) = (x_1^4, x_2)$ feature space, then we end up with linearly related features and the abnormal data point can be flagged as such. Unfortunately, the task of finding such (nonlinear) mapping that maps raw data to feature variables is not trivial. Thankfully, there is something called ‘kernel trick’ that allows you to work in the feature space without having to define the nonlinear mapping. We will learn how this is accomplished in the next section.

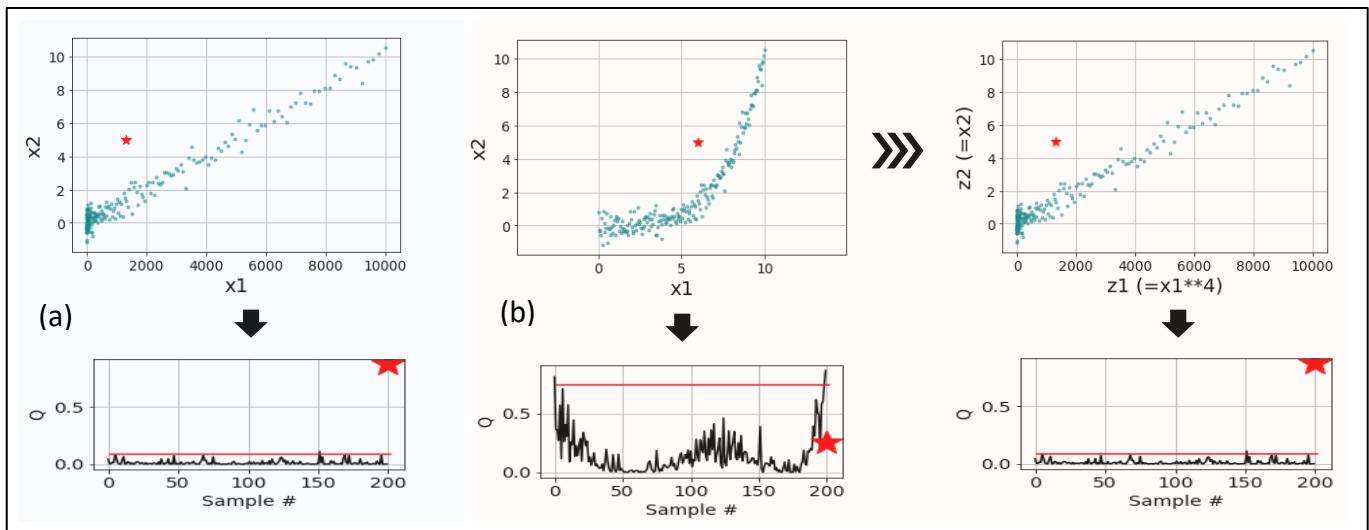
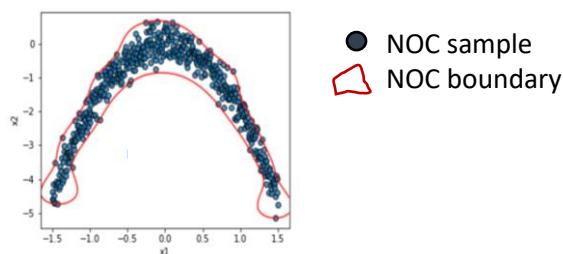


Figure 10.1: Nonlinearity impact on PCA-fault detection. Faulty sample shown in red. [One principal component chosen in all simulations]

KPCA can work with arbitrary data distributions. As far as process monitoring applications are concerned, KPCA can help you create an abnormality boundary around your NOC data as shown below.



To understand how KPCA works, let's revisit the mathematical underpinnings of PCA.

Kernel functions and kernel trick

Usage of kernel trick is not limited to KPCA and KPLS. Other ML techniques such as SVM, CVA, etc., also use kernel functions to model nonlinear processes. So, what are these kernel functions? Let's try to understand them.

We alluded to before that a popular approach to handling nonlinearity is to map observation sample x to x_F in feature space where conventional linear ML technique can be applied.

$$\varphi(x): x \rightarrow x_F$$

However, the map $\varphi(\cdot)$ is unknown. Thankfully, in the mathematical formulation of many ML algorithms, the inner (or dot) product of feature vectors, $\varphi(x)^T \varphi(x)$, is frequently encountered. This inner product is denoted as

$$k(x_i, x_j) = \langle \varphi(x_i), \varphi(x_j) \rangle = \varphi(x_i)^T \varphi(x_j)$$

where $k(\cdot, \cdot)$ is called the kernel function. Several forms of $k(\cdot, \cdot)$ are available and the most common form is Gaussian or radial basis function defined as

$$k(x_i, x_j) = \exp\left[-\frac{(x_i - x_j)^T (x_i - x_j)}{\sigma^2}\right]$$

where, δ (a hyperparameter), is called kernel width. Usage of kernel functions allow application of linear ML techniques in feature space without explicitly knowing the feature vectors and this trick is called the '*kernel trick*'. Another term you will encounter in kernelized algorithms is kernel matrix (often denoted as K). The $(i, j)^{\text{th}}$ element of K is simply $k(x_i, x_j)$. The table below lists the commonly used kernel functions

Function	Equation	Hyperparameter
Linear	$k(x, z) = x^T z$	
Gaussian	$k(x, z) = \exp(-\frac{\ x - z\ ^2}{\sigma^2})$	σ
Polynomial	$k(x, z) = (\gamma x^T z + r)^d$	γ, r, d
Sigmoid	$k(x, z) = \tanh(\gamma x^T z + r)$	γ, r

Let's use the polynomial kernel to illustrate how using kernel functions amounts to higher dimensional mapping. Assume that we use the following kernel

$$k(x, z) = (x^T z + 1)^2$$

where $x = [x_1, x_2]^T$ and $z = [z_1, z_2]^T$ are two vectors in the original 2D space. We claim that the above kernel is equivalent to the following mapping

$$\varphi(x) = [x_1, x_2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, 1]$$

To see how, just compute $\varphi(x)^T \varphi(z)$

$$\begin{aligned}\varphi(x)^T \varphi(z) &= x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 z_1 + 2x_2 z_2 + 2x_1 x_2 z_1 z_2 + 1 \\ &= (x_1 z_1 + x_2 z_2 + 1)^2 \\ &= (x^T z + 1)^2 \\ &= k(x, z)\end{aligned}$$

Therefore, if you use the above polynomial kernel, you are implicitly projecting your data onto a 6th dimensional feature space! If you were amazed by this illustration, you will find it more interesting to know that Gaussian kernels map original space into an infinite dimensional feature space! Luckily, we don't need to know the form of this feature space.

Mathematical background

The main objective in KPCA is to obtain the score vectors in the feature space. To see how it can be obtained, let $x \in \mathbb{R}^m$ and $\varphi(x)$ denote a sample in the original measurement space and the projected feature vector in the high-dimensional feature space, respectively. We know that the conventional PCA involves the solution of the following eigenvalue problem

$$\frac{1}{N-1} X^T X p = \lambda p$$

Covariance matrix

Let $X_F = [\varphi(x_1), \varphi(x_2), \dots, \varphi(x_N)]^T$ denote the feature data matrix. We assume for now that X_F is mean centered. PCA in the feature space, therefore, involves the following eigen-decomposition problem

$$\frac{1}{N-1} X_F^T X_F v = \lambda_F v \quad \text{eq. 1}$$

↑
Eigenvector in feature space

X_F is not known and, therefore, the above equation cannot be solved directly. Let's premultiply both sides of Eq. 1 by X_F which results in

$$\frac{1}{N-1} X_F X_F^T X_F v = \lambda_F X_F v \quad \text{eq. 2}$$

Linear algebra shows that eigenvector⁶⁸ can be written as a linear combination of the samples, i.e.,

$$v = \sum_{i=1}^N \alpha_i \varphi(x_i) = X_F^T \alpha \quad \text{eq. 3}$$

Let us also define the kernel matrix, $K = X_F X_F^T$. Let's now substitute the kernel matrix and v from Eq. 3 in Eq. 2 which leads to

$$K\alpha = (N-1)\lambda_F \alpha \quad \text{eq. 4}$$

Eq. 4 is an eigen-decomposition problem which can be solved for the kernel eigenvectors $\alpha^{(1)}, \alpha^{(2)}, \dots, \alpha^{(N)}$ and kernel eigenvalues⁶⁹ $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_N$. The l^{th} eigenvector $\alpha^{(l)} \equiv [\alpha_1^{(l)}, \alpha_2^{(l)}, \dots, \alpha_N^{(l)}] \in \mathbb{R}^N$. One may retain only the first a eigenvectors for dimensionality reduction in feature space. The corresponding eigenvector in feature space becomes $v^{(l)} = X_F^T \alpha^{(l)}$. To ensure eigenvector $v^{(l)}$ is of unit length, the following condition is imposed on $\alpha^{(l)}$,

$$\begin{aligned} \|v^{(l)}\|_2 &= v^{(l)T} v^{(l)} = \alpha^{(l)T} X_F X_F^T \alpha^{(l)} = 1 \\ &\Rightarrow (N-1)\lambda_{Fl} \|\alpha^{(l)}\|_2 = 1 \\ &\Rightarrow \|\alpha^{(l)}\|_2 = 1/(N-1)\lambda_{Fl} \end{aligned} \quad \text{eq. 5}$$

⁶⁸ Note that with Gaussian kernels, $\varphi(x)$ and v have infinite dimensions

⁶⁹ Kernel eigenvalue λ_l can be divided by $N-1$ to obtain eigenvalue λ_{Fl} .

Above expression implies that the calculated kernel eigenvectors $\alpha^{(l)}$ obtained from Eq. 4 corresponding to non-zero eigenvalues are scaled to satisfy Eq. 5. Note that the eigenvectors $v^{(l)}$ are still unobtainable as X_F is unknown. However, our main variable of interest is score vector $t_j \in \mathbb{R}^a$ corresponding to sample x_j . Before we see how t_j can be obtained, we need to come straight on a big assumption we made regarding data in feature space $\varphi(x_1), \varphi(x_2), \dots, \varphi(x_N)$ being mean centered. To ensure mean-centering, kernel matrix K is modified as follows

$$\bar{K} = K - \mathbf{1}_N K - K \mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N \quad \text{eq. 6}$$

$$\text{Where } \mathbf{1}_N = \frac{1}{N} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{N \times N}$$

$$\rightarrow \bar{K} \text{ can be shown to be equal to } \bar{X}_F^T \bar{X}_F$$

$\bar{X}_F = [\bar{\varphi}(x_1), \bar{\varphi}(x_2), \dots, \bar{\varphi}(x_N)]^T$

Centered feature vector

$$\rightarrow \text{equivalently, } \bar{K}_{ij} = \bar{k}(x_i, x_j) = \bar{\varphi}(x_i)^T \bar{\varphi}(x_j)$$

Score vectors for training and test samples

To find score vector ($t_j \in \mathbb{R}^{ax1} = [t_{j1}, t_{j2}, \dots, t_{ja}]^T$) for a training sample x_j , we need to compute the score or projection of $\varphi(x_i)$ along each eigenvector $v^{(l)}$, $l = 1, 2, \dots, a$, which can be obtained as follows

$$t_{jl} = \langle v_l, \bar{\varphi}(x_j) \rangle = \sum_{i=1}^N \alpha_i^{(l)} \bar{k}(x_i, x_j) = \sum_{i=1}^N \alpha_i^{(l)} \bar{K}_{ji} = \alpha^{(l)T} \bar{k}_j \quad \text{eq. 7}$$

Scaled eigenvectors [Eq. 5]

Centered kernel vector $\in \mathbb{R}^{N \times 1}$ with $\bar{k}_{ji} = \bar{k}(x_j, x_i) = \bar{K}_{ij}$
where x_i belongs to training dataset and $i = 1, 2, \dots, N$.

For a test sample x_t , the above expression becomes

$$t_{tl} = \alpha^{(l)}^T \bar{k}_t \quad \text{eq. 8}$$

Where $\bar{k}_t = k_t - \mathbf{1}_N k_t - K\mathbf{1}_t + \mathbf{1}_N K\mathbf{1}_t$

kernel vector with $k_{ti} = k(x_t, x_i)$
where x_i belongs to training dataset and $i = 1, 2, \dots, N$.

$$\mathbf{1}_t = \frac{1}{N} \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{N \times 1}$$

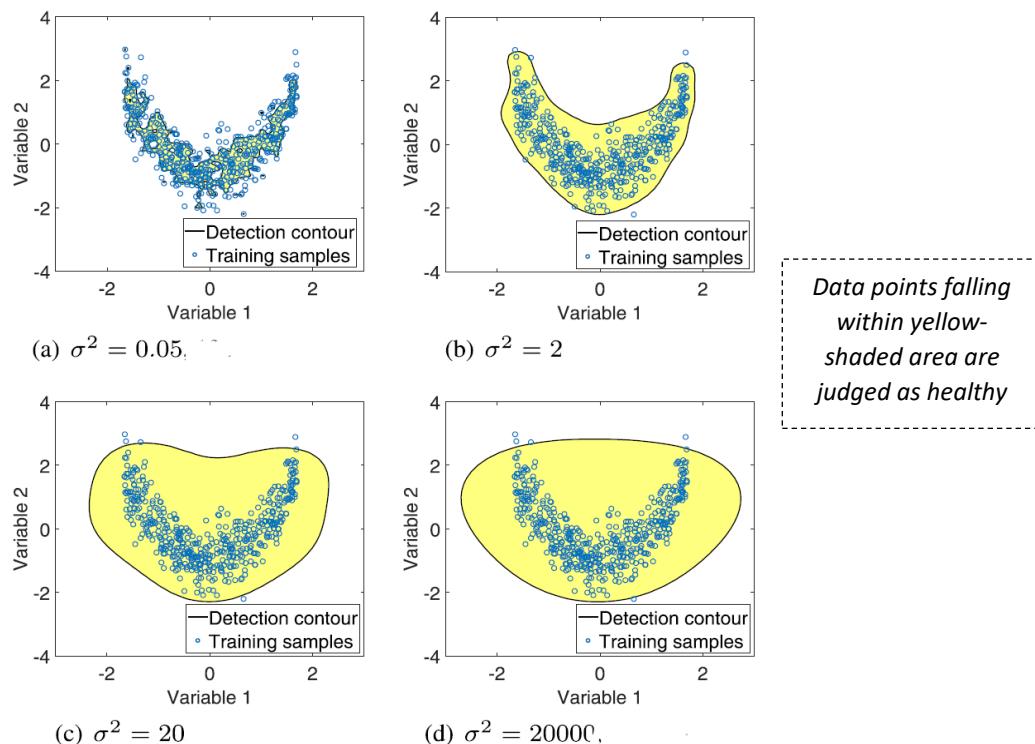
from [Eq. 6]

Hyperparameter specification

Assuming that Gaussian kernel is chosen, you will need to specify the kernel width σ and the number of retained latents (a).

Selection of kernel width (σ)

The selection of σ strongly impacts the accuracy of the KPCA model. The illustration below shows the impact of kernel width on the abnormality detection contour.



[Diagram shared under Creative Commons Attribution 4.0 license
(creativecommons.org/licenses/by/4.0/) by Tan et al. in paper titled 'Monitoring Statistics and Tuning of Kernel Principal Component Analysis With Radial Basis Function Kernels']

It is apparent that very small σ yields overfitted models leading to high rate of false alarms, while large σ results in underfitted models because the abnormality contours are not able to accurately capture the data distribution. The kernel width may be selected empirically or via cross-validation. Empirically, σ is often set to be $5m$ where m is the number of process variables⁷⁰. In the cross-validation approach, σ is set to be the smallest value that results in acceptable level of false alarm on a validation dataset. Tan et al., suggested the range $[0, d_{train,max}]^{71}$ to search for the optimal σ where $d_{train,max} = \max \sqrt{\|x_i - x_j\|}$ for $x_i, x_j \in$ training dataset.

Selection of number of retained latents (a)

A common approach is to choose a such that 95% of the variability in feature space is captured, i.e.,

$$a = \underset{j \in [1,N]}{\operatorname{argmin}} \frac{\sum_{i=1}^j \lambda_i}{\sum_{i=1}^N \lambda_i} \geq 0.95$$

Another popular approach is the average eigenvalue method where only eigenvectors with eigenvalues above the average of all eigenvalues are included.

10.2 Fault Detection using Kernel PCA

Phew! It was a lot of mathematics in the previous section. Our aim was not to scare you away by going into this level of details, but to provide you with enough background so that you can carry out each step of KPCA-based fault detection confidently. In fact, Sklearn provides KernelPCA class that does the job of computing the scores for you; all you need to do it compute the monitoring statistic. To further clarify all the steps, we will use the following synthetic process for demonstration. It is a system with 3 measurements x_1 , x_2 and x_3 .

$$\begin{aligned} x_1 &= u^2 + 0.3 \sin(2\pi u) + \varepsilon_1 \\ x_2 &= u + \varepsilon_2 \\ x_3 &= u^3 + u + 1 + \varepsilon_3 \end{aligned}$$

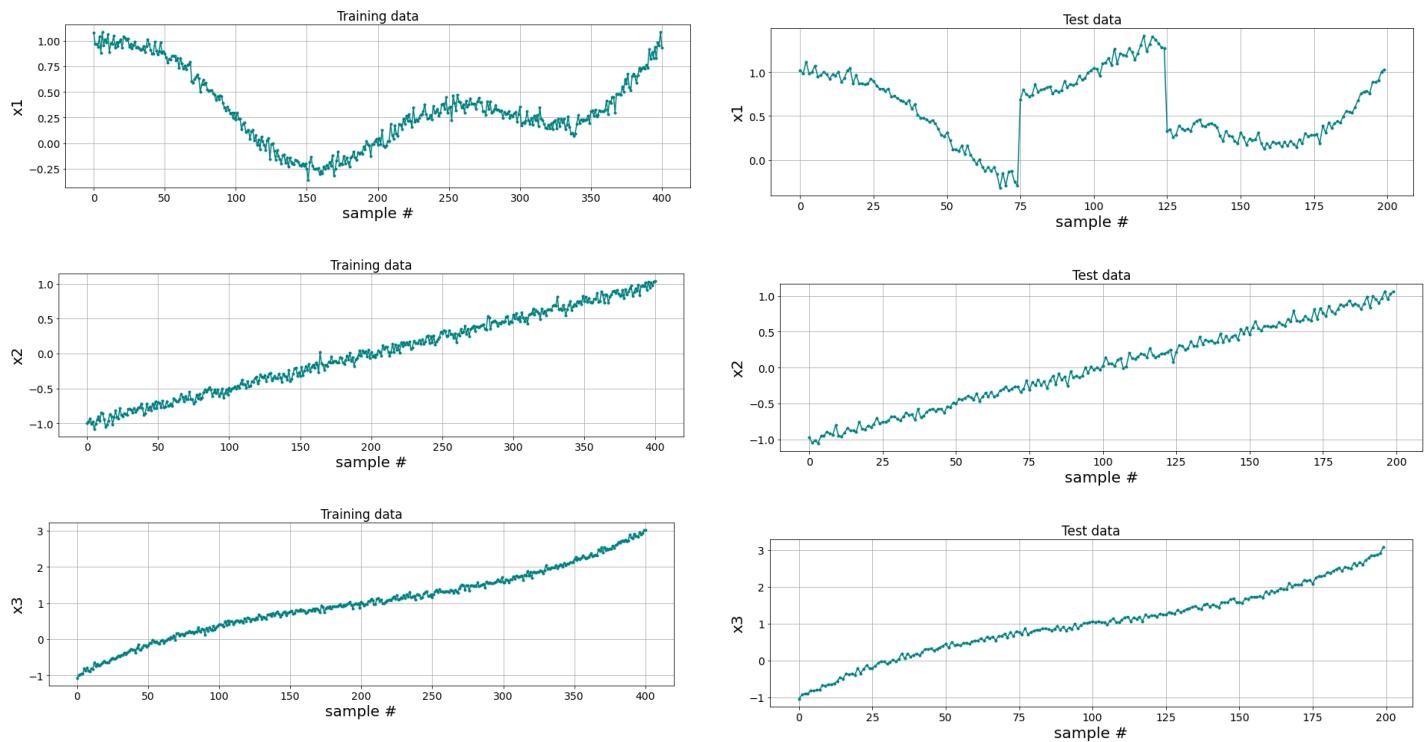
⁷⁰ Cho et al., Fault identification for process monitoring using kernel principal component analysis. *Chemical Engineering Science*, 2005.

⁷¹ Tan et al., . Monitoring Statistics and Tuning of Kernel Principal Component Analysis With Radial Basis Function Kernels. IEEE Access, 2020.

Here, u is a variable uniformly sampled between -1 and 1, and ε_i is independent white noise with variance 0.05. This system has been used by Nounou et al. to demonstrate superiority of kernelized MSPM models in an open-access paper titled '*Process monitoring using data-based fault detection techniques: comparative studies (2017)*'. The system was simulated to generate 401 and 200 samples of NOC and test data (provided in files *multivariateKPLS_NOC_data.txt* and *multivariateKPLS_test_data.txt*, respectively). In test data, a single fault of unit magnitude is introduced between samples 75 and 125 in x_1 . Let's begin by reading the data files.

```
# determine thresholds as 99th percentile of respective metrics
import numpy as np, matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import KernelPCA

# read data
X_train = np.loadtxt('KPCA_NOC_data.txt')
X_test = np.loadtxt('KPCA_test_data.txt')
```



We will now go through each step of generating monitoring chart for the test data. Therefore, let's first understand how the monitoring statistic is computed.

Monitoring statistic

The primary statistic to monitor is Q , which for a sample x_t is defined as

$$Q = \|\bar{\varphi}(x_t) - \hat{\varphi}_a(x_t)\|^2$$

reconstructed feature vector using a latents

The above expression can be expressed in terms of the score vector t_t as follows⁷²

$$Q = \sum_{i=1}^n t_{ti}^2 - \sum_{i=1}^a t_{ti}^2$$

- n is the number of non-zero eigenvalues in Eq. 4

Note that Tan et al. have shown that the KPCA Q metric can help detect fault cases where variables go out of healthy operating range as well as cases where test samples do not follow model of the training data.



In KPCA literature, you may encounter usage of Hotelling's T^2 as a monitoring statistic. However, Tan et al. have shown that Hotelling's T^2 is non-monotonic w.r.t. the severity of fault, i.e., Hotelling's T^2 may decrease when faults become more severe! For example, if a sample x_{test} lies far away from the NOC samples, then score value along every principal component will be small because the Gaussian kernel function values will be small. Therefore, T^2 will be small and will only become smaller as x_{test} goes farther away from the healthy data.

Let us now implement the KPCA model and generate monitoring charts.

```
# scale data
X_scaler = StandardScaler()
X_train_scaled = X_scaler.fit_transform(X_train)

# fit kPCA model
gamma = 1/((5*3)**2) # gamma = 1/σ²
kPCA = KernelPCA(kernel='rbf', gamma=gamma)
```

⁷² Zhou et al., Randomized Kernel Principal Component Analysis for Modeling and Monitoring of Nonlinear Industrial Processes with Massive Data. *Industrial & Engineering Chemistry Research*, 2019.

```

kPCA.fit(X_train_scaled)

# find number of components to retain
eigVals = kPCA.eigenvalues_ # 38 non-zero eigen values have been found
eigVals_normalized = eigVals / np.sum(eigVals)
cum_eigVals = 100*np.cumsum(eigVals_normalized)
n_comp = np.argmax(cum_eigVals >= 95) + 1
print('Number of components cumulatively explaining atleast 95% variance: ', n_comp)

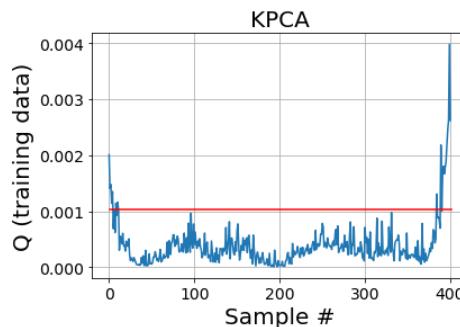
>>> Number of components cumulatively explaining atleast 95% variance: 2

# compute scores for training data
scores_train = kPCA.transform(X_train_scaled) # one column for each of the 38 columns

# compute Q statistics for training data
N = X_train.shape[0]
Q_train = np.zeros((N,1))
for i in range(N):
    Q_train[i] = np.dot(scores_train[i,:], scores_train[i,:]) - np.dot(scores_train[i,:n_comp],
                                                                     scores_train[i,:n_comp])
Q_CL = np.percentile(Q_train, 95)

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

```



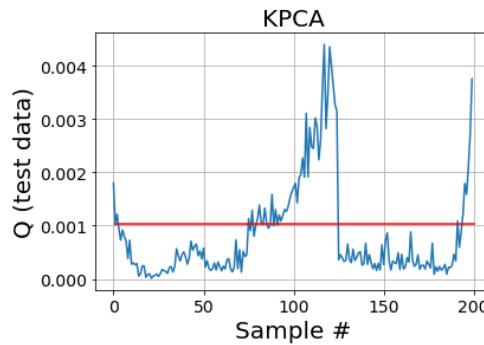
```

#####
##  kPCA model application on test data
#####
X_test_scaled = X_scaler.transform(X_test)
scores_test = kPCA.transform(X_test_scaled)

```

```
# compute Q statistics for test data
N_test = X_test.shape[0]
Q_test = np.zeros((N_test,1))
for i in range(N_test):
    Q_test[i] = np.dot(scores_test[i,:], scores_test[i,:]) - np.dot(scores_test[i,:n_comp],
                                                               scores_test[i,:n_comp])

# monitoring chart for test data
plt.figure(), plt.plot(Q_test), plt.plot([1,len(Q_test)],[Q_CL,Q_CL], color='red')
plt.xlabel('Sample #'), plt.ylabel('Q (test data)')
```



Hopefully, you now feel confident about being able to deploy a KPCA-based fault detection system for nonlinear systems.

10.3 Kernel PLS: An Introduction

Kernel PLS is the nonlinear extension of conventional PLS to handle nonlinear process data that exhibit significant nonlinearity. As shown in Figure 10.2 below, the input vectors are mapped to an ‘unknown’ feature space such that the variables become linearly related. We know from Chapter 7 that the NIPALS algorithm used to fit PLS model involves the term XX^T and therefore, like PCA, kernel functions are used to implicitly define $\varphi\varphi^T$ in the feature space, obtain the score vectors, and thereafter, generate the predicted outputs. The overall procedure relies on classic linear algebra and retains most of the favorable properties of standard PLS.

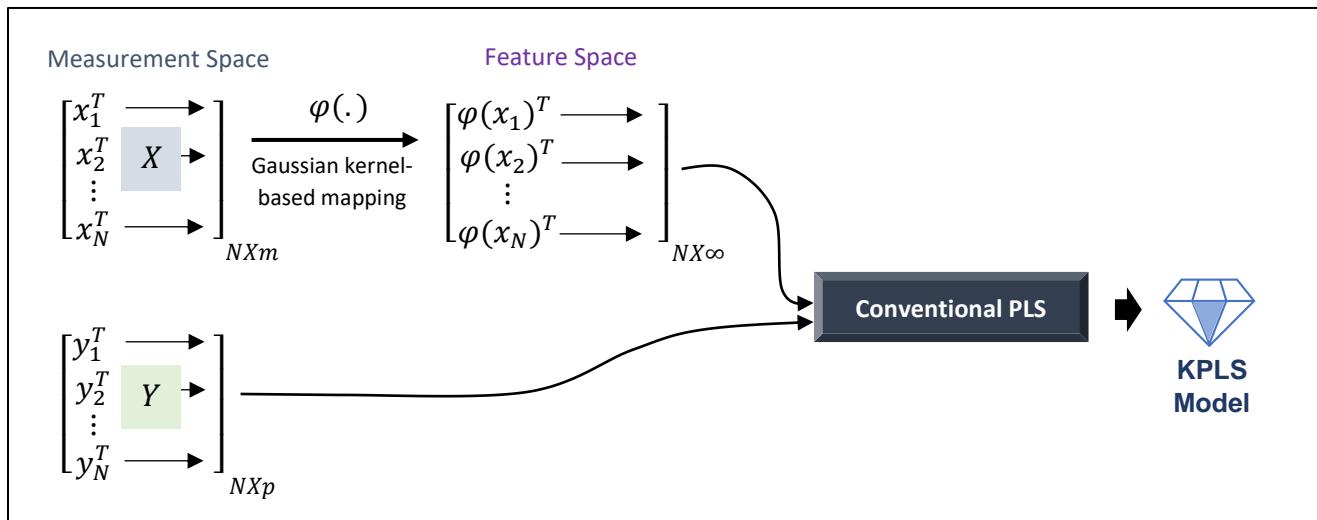


Figure 10.2: Kernel PLS methodology

Sklearn, unfortunately, does not provide any off-the-shelf module for KPLS. Nonetheless, it is not tough to code the steps of KPLS. The mathematical exposition in the following subsection will provide all the information you need to work on your own KPCA module.

Alternative nonlinear PLS models



Kernelized PLS is not the only methodology commonly employed for nonlinear regression modeling. An alternative methodology involves working in the original measurement space, projecting input and output data linearly to obtain the scores, and building a nonlinear inner relation between the t and u scores.

$$u \approx f(t)$$

↳ if $f(\cdot)$ is polynomial \Rightarrow polynomial PLS
↳ if $f(\cdot)$ is neural network \Rightarrow neural net PLS

Mathematical background

Let $X \in \mathbb{R}^{Nxm}$ and $Y \in \mathbb{R}^{Nxp}$ be the centered input and output data matrices of the training samples. Let $\bar{\varphi}(x_t)$ denote the mapped and centered feature vector of the i^{th} sample or row of X , $X_F \in \mathbb{R}^{N \times \infty}$ denote the input data matrix in feature space. The kernel matrix is computed and centered as done in KPCA algorithm. The algorithm below summarizes the KPLS model training via NIPALS that can provide us the input and output score matrices T and U .

Algorithm: KPLS training via NIPALS

- 1) Initialize $i = 1$, $\bar{K}_1 = \bar{K}, Y_1 = Y$
- 2) Set score vector $u_i (\in \mathbb{R}^{N \times 1})$ to any column of Y_i
- 3) Compute score vector $t_i (\in \mathbb{R}^{N \times 1}) = \bar{X}_{Fi} w_i = \bar{X}_{Fi} \bar{X}_{Fi}^T w_i = \bar{K}_i u_i$
- 4) Scale t_i to unit length: $t_i = t_i / \|t_i\|$
- 5) Calculate weight vector $c_i = Y_i^T t_i$
- 6) Update score vector $u_i = Y_i c_i$
- 7) Scale u_i to unit length: $u_i = u_i / \|u_i\|$
- 8) Repeat steps 3 to 7 until t_i converges
- 9) Deflate matrices:

$$\bar{K}_{i+1} = \bar{K}_i - t_i t_i^T \bar{K}_i - \bar{K}_i t_i t_i^T + t_i t_i^T \bar{K}_i t_i t_i^T$$

$$Y_{i+1} = Y_i - t_i t_i^T Y_i$$
- 10) Set $i = i + 1$ and return to step 2. Stop when $i > a$ [number of latents to be retained]
- 11) Form the matrices $T = [t_1, t_2, \dots, t_a]$

$$U = [u_1, u_2, \dots, u_a]$$

The training score matrix can also be represented as

$$T = \bar{X}_F R \quad \text{where } R = \bar{X}_F^T U (T^T \bar{K} U)^{-1} \in \mathbb{R}^{\infty \times a}$$

$$\Rightarrow T = \bar{K} U (T^T \bar{K} U)^{-1}$$

The R matrix will help us compute the score values for a test sample as we will see shortly. The last missing piece of information is the regression coefficient matrix which can be used to generate the output predictions

$$\begin{aligned}\hat{Y} &= \bar{X}_F B \quad \text{where } B = \bar{X}_F^T U (T^T \bar{K} U)^{-1} T^T Y \\ \Rightarrow \hat{Y} &= \bar{K} U (T^T \bar{K} U)^{-1} T^T Y\end{aligned}$$

Score vector and predictions for a test sample

The score vector $t_t (\in \mathbb{R}^{ax1})$ for a test sample x_t can be obtained as

$$\begin{aligned}t_t &= R^T \bar{\varphi}(x_t) \\ &= (U^T \bar{K} T)^{-1} U^T \bar{X}_F \bar{\varphi}(x_t) \\ &= (U^T \bar{K} T)^{-1} U^T \bar{k}_t \quad \begin{matrix} \text{Centered kernel vector } \in \mathbb{R}^{N \times 1} \text{ with } k_{ti} = k(x_t, x_i) \\ \text{where } x_i \text{ belongs to training dataset and } i = 1, 2, \dots, N. \\ \text{Given as in Eq. 8.} \end{matrix}\end{aligned}$$

The predictions $\hat{y} (\in \mathbb{R}^{px1})$ can be obtained using the matrix B

$$\begin{aligned}\hat{y} &= B^T \bar{\varphi}(x_t) \\ &= Y^T T (U^T \bar{K} T)^{-1} U^T \bar{X}_F \bar{\varphi}(x_t) \\ &= Y^T T (U^T \bar{K} T)^{-1} U^T \bar{X}_F \bar{k}_t\end{aligned}$$

Hyperparameter specification

Like KPCA, KPLS involve selection of two critical hyperparameters: the Gaussian kernel (σ) and the number of latents retained (a). The number of latents retained is commonly determined via cross-validation, where we choose a that minimizes the error in predicted outputs for validation dataset. For σ , Jose et al. showed⁷³ that setting $\sigma = 2m$ provides robust fault detection performance. Alternatively, σ can be chosen such that we obtain an acceptable level of false alerts on a validation dataset of NOC samples.

⁷³ Jose et al., New contributions to non-linear process monitoring through kernel partial least squares. *Chemometrics and Intelligent Laboratory Systems*, 2014.

10.4 Fault Detection using Kernel PLS

Monitoring statistics

Like KPCA, we will use the Q (or SPE) statistics for fault detection. SPE_φ and SPE_y monitor the residuals in the feature space and the output space, respectively. SPE_φ is computed as follows for a sample x_t (Jose et al.)

$$\begin{aligned} SPE_\varphi &= \|\bar{\varphi}(x_t) - \hat{\varphi}_a(x_t)\|^2 \\ &= \bar{k}(x_t, x_t) - 2\bar{k}_t^T \bar{K}V t_t + t_t^T T^T \bar{K} T t_t \\ &\quad \downarrow \\ & V = U(T^T \bar{K} U)^{-1} \end{aligned}$$

The term $\bar{k}(x_t, x_t)$ can simply be picked from the \bar{K} matrix if x_t is a training sample. For a test sample, it is given as⁷⁴

$$\bar{k}(x_t, x_t) = 1 - \frac{2}{N} \sum_{i=1}^N k(x_i, x_t) + \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N K_{ij}$$

where x_i belongs to training dataset and $i = 1, 2, \dots, N$.

SPE_y is simple to compute as follows

$$SPE_y = \|y - \hat{y}\|^2$$

The flowchart below summarizes all the steps of the KPLS-based fault detection solution development.

⁷⁴ Peng at al., Quality-related process monitoring based on total kernel PLS model and its industrial application. *Mathematical Problems in Engineering*, 2013.

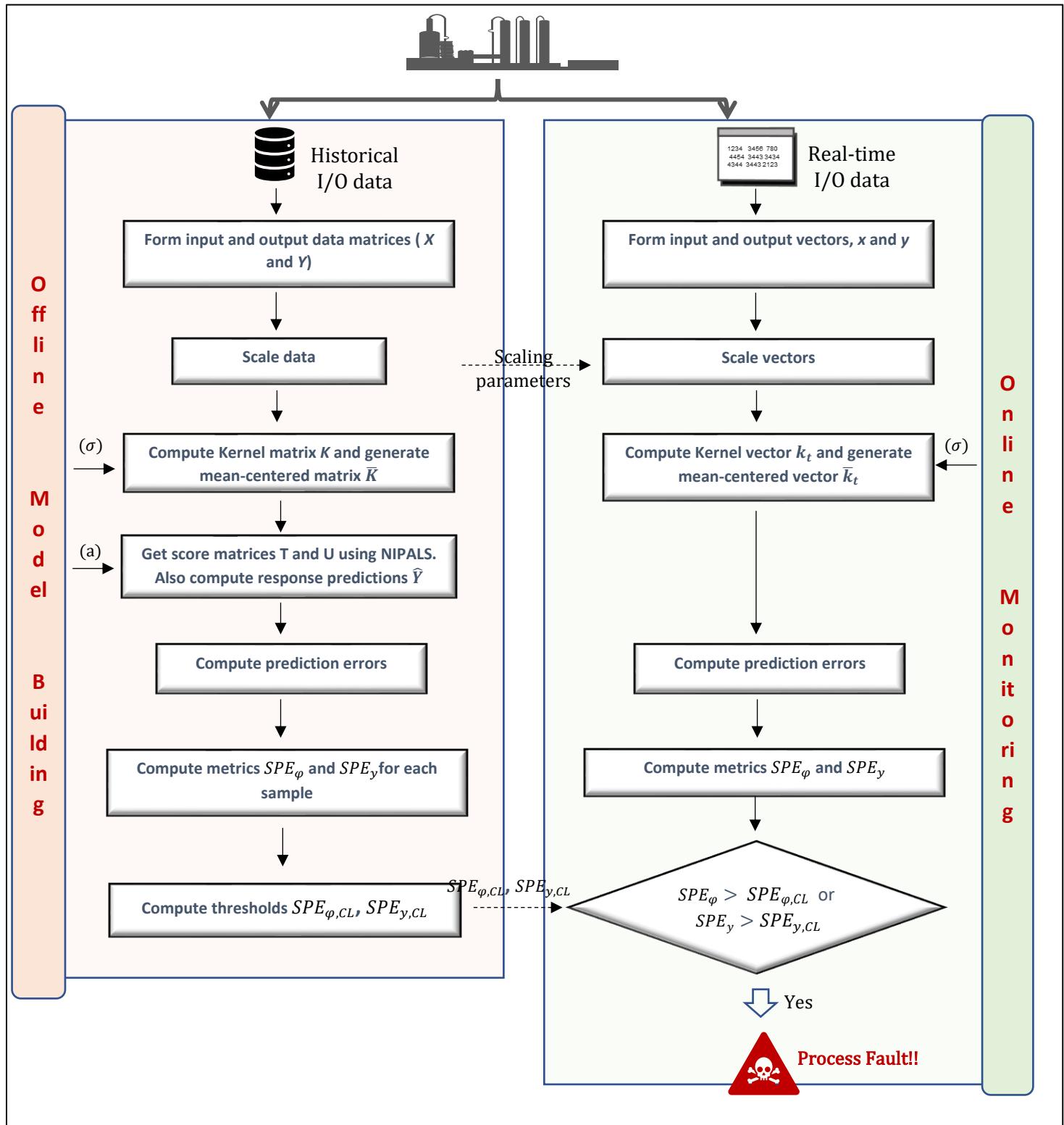


Figure 10.3: Complete workflow for fault detection via KPLS

The kernel trick acts as a beautiful solution to allow the KPCA/KPLS scores to relate nonlinearly to the original input data while retaining the simplicity of linear algebra-based PCA/PLS. However, before we fall in love with kernelized MSPM techniques, remember that there are a few drawbacks as well! If the number of samples, N , is very large then eigenvector decomposition of $N \times N$ dimensional kernel matrix can become challenging. Also, judicious selection of the hyperparameter σ , is critical, else a poor model is obtained.

Summary

In this chapter, we acquainted ourselves with the procedures to obtain nonlinear versions of classical MSPM techniques using kernels. Specifically, we looked at kernel PCA and kernel PLS algorithms. These techniques can help you build robust monitoring solutions for process systems that show significant nonlinearities. In the next chapter, we will learn how to deal with multi-clustered.

Chapter 11

Process Monitoring of Multi-Mode Processes

In the previous chapters, we witnessed the benefits of customizing the conventional MSPM techniques for non-Gaussian, dynamic, and nonlinear processes. In this chapter, we will remove the last remaining restriction of unimodal operation. In your career, you will frequently encounter industrial datasets that exhibit multiple operating modes due to variations in production levels, feedstock compositions, ambient temperature, product grades, etc., and data-points from different modes tend to group into different clusters. The mean and covariance of process variables may be different under different operation models and therefore, when you are building a monitoring tool, judicious incorporation of the knowledge of these data clusters into process models will lead to better performance and, alternatively, failure to do so will often lead to unsatisfactory monitoring performance.

In absence of specific process knowledge or when the number of variables is large, it is not trivial to find the number of clusters or to characterize the clusters. Fortunately, several methodologies are available which you can choose from for your specific solution. In this chapter, we will learn different ways of working with multimodal data, some of the popular clustering algorithms, and understand their strengths and weaknesses. We will conclude by building a monitoring tool for a multimode semiconductor process. Specifically, the following topics are covered

- Different methodologies for modeling multimodal process data
- Introduction to clustering
- Finding groups using classical k-means clustering
- Probabilistic clustering via Gaussian mixture modeling
- Process monitoring of multimode semiconductor manufacturing operation

11.1 Need and Methods for Specialized Handling of Multimode Processes

In process systems, multimode operations occur naturally due to varied reasons. For example, in a power generation plant, production level changes according to the demand leading to significantly different values of plant variables with potentially different inter-variable correlations at different production levels. The multimode nature of data distribution causes problems with traditional ML techniques. To understand this, consider the illustrations in Figure 11.1. In subfigure (a), data indicates 2 distinct modes of operation. From process monitoring perspective, it would make sense to draw separate monitoring boundaries around the two clusters; doing so would clearly identify the red-colored data-point as an outlier or a fault. The Conventional PCA-based monitoring, on the other hand, would fail to identify the outlier. In subfigure (b), the correlation between the variables is different in the two clusters. It would make sense to build separate models for the two clusters to capture the different correlation structure. The Conventional PLS model would give inaccurate results.

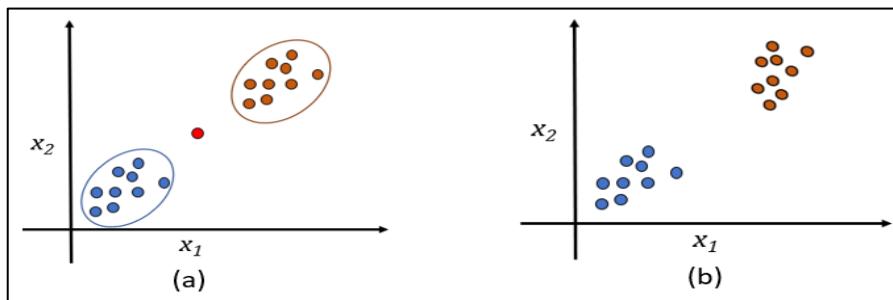


Figure 11.1: Illustrative scenarios for which conventional ML techniques are ill-suited

A few different methodologies have been adopted by the PSE community to monitor multimode process operations. Let's familiarize ourselves quickly with these.

Multiple model approach

Here, separate models are built for each of the clusters corresponding to the different operation modes as shown in the figure below. Once the clusters have been characterized in the training data and cluster-wise models have been built, prediction for a new sample can be obtained by either only considering the cluster-model most suitable for the new sample or combining the predictions from all the models as shown in Figure 11.2. The decision fusion module can also take various forms. For example, for a process monitoring application, a simple fusion strategy could be to consider a new sample as a normal sample if atleast one of the cluster-models predict so. A different strategy could be to combine the abnormality metrics from all the models and make prediction based on this fused metric. Similarly, for soft sensing application, response variable prediction from individual models can be weighted and combined to provide final prediction.

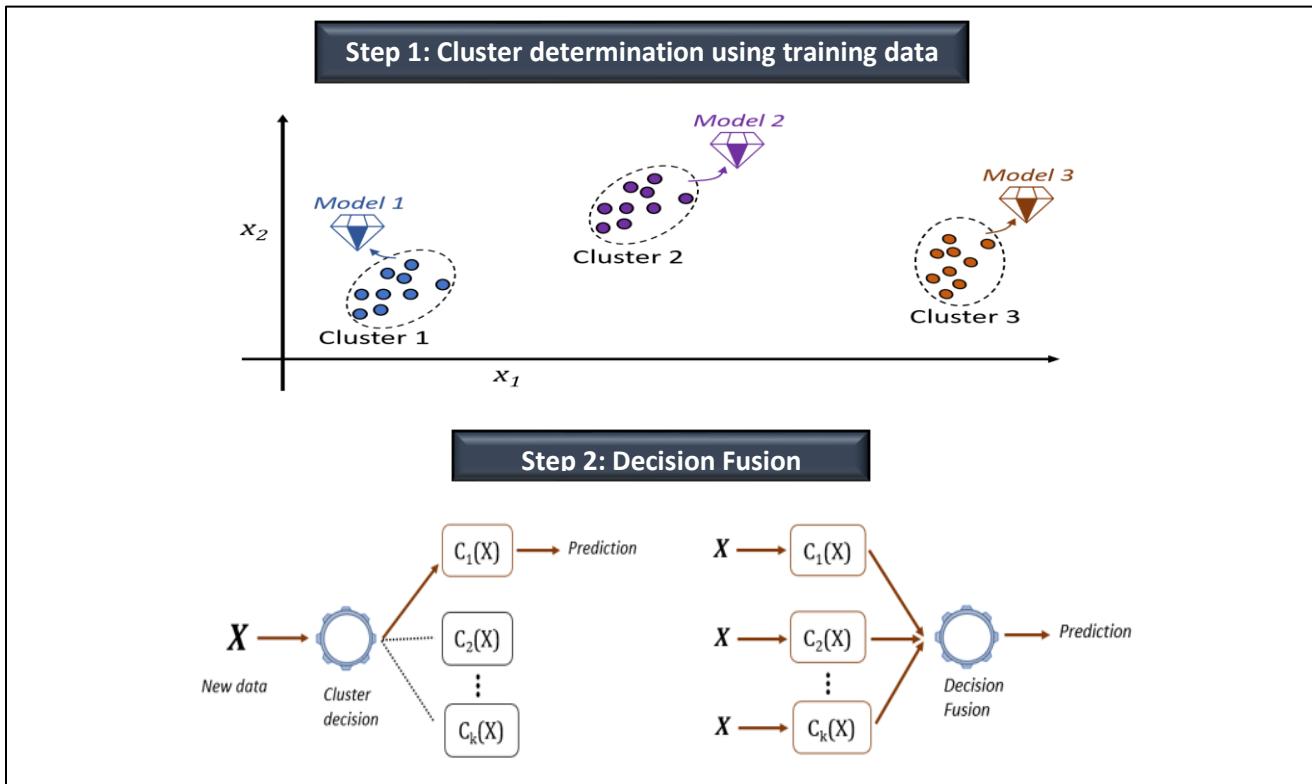


Figure 11.2: Multiple model approach for multimode processes

Lazy or just-in-time learning

In this approach, the model building exercise is carried out online. When new process data come in, relevant data are fetched from the historical dataset that are similar to the incoming samples based on some nearest neighborhood criterion. A local model is built using the fetched relevant data. The obtained model processes the incoming samples and is then discarded. A new local model is built when the next samples come in.

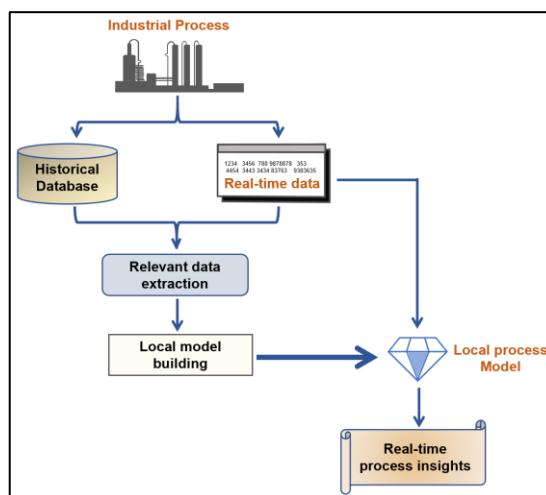


Figure 11.3: Steps involved in a just-in-time learning methodology

External analysis

In this strategy, the influence of process variables (called external variables) such as product grade, feed flow, etc., that lead to multimode operation is removed from the other ‘main’ process variables and then the conventional MSPM techniques are employed on the ensuing residuals as shown in the figure below.

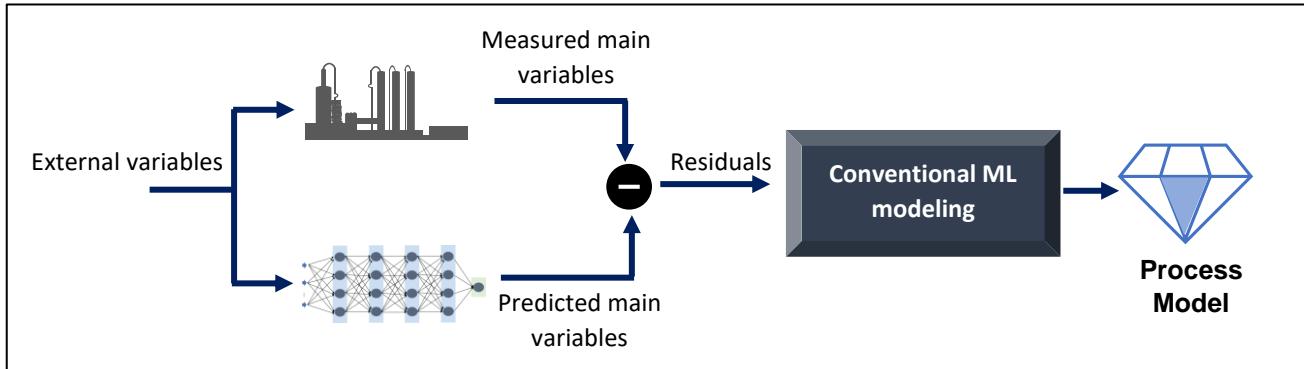


Figure 11.4: External analysis approach for multimode data

Single model approach

In this approach a single model is built with the hope that it captures the presence of different modes of operation. Several ML techniques fall in this category such as neural networks, SVDD, etc. You will learn about them in the next parts of the book.

Proximity-based approach

In this approach the presence of multiple NOC clusters is taken into account by making inference for a test sample based on the sample’s proximity to the NOC samples. Proximity measures such as distance from neighbors or local density is commonly employed. We will cover these techniques in Chapter 14.

In this chapter, we will focus upon on the multi-modal approach which involves finding the different clusters in the dataset. Therefore, let’s now acquaint ourselves with some popular clustering techniques. However, before we do that, let’s take a quick look at the dataset that we will work with.

11.2 Multimode Semiconductor Manufacturing Dataset

In this chapter, we will work with dataset from a semiconductor manufacturing process⁷⁵. The dataset was obtained from multiple batches from an etching process and consists of 19 process variables measured over the course of 108 normal batches and 21 faulty batches. The batch durations range from 95 to 112 seconds. In the rest of the chapter, we will investigate whether the dataset exhibit multimode operations and devise a monitoring strategy to automatically detect the faulty batches. The data is provided in a MATLAB structure array format and so we will use a library to fetch data in Python environment.

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

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

variable_names = Etch_data[0,0].variables

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

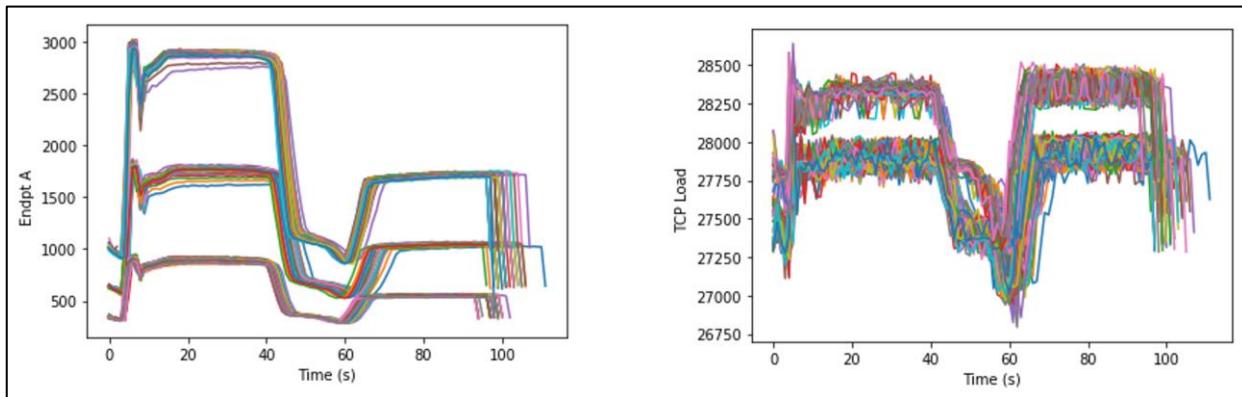


Figure 11.5: Select variable plots for all batches in metal etch dataset. Each colored curve corresponds to a batch.

⁷⁵ Data can be downloaded from <http://www.eigenvector.com/data/Etch>.

Figure 11.5 does indicate multimode operations with mean and covariance changes. It is however difficult to estimate the number of operation modes by examining high-dimensional dataset directly. A popular practice is to reduce process dimensionality via PCA and then apply clustering to facilitate visualization. Performing PCA serves other purposes as well. We will see later that expectation-maximization (EM) algorithm is employed to estimate cluster parameters in K-Means and GMM models. High dimensionality implies high number of parameters to be estimated which increases possibility of EM converging to locally optimum results and correlated variables cause EM convergence issues. PCA helps to overcome these two problems simultaneously.

We will employ multiway PCA for this batch process dataset. We will follow the approach of He et al.⁷⁶ where for each batch 85 sample points are retained to deal with batch length variability, first 5 samples are ignored to eliminate initial fluctuations in sensor measurements, and 3 PCs are retained.

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

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

    if calibration_expt.shape[0] < 85:
        continue

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

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

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

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

⁷⁶ He and Wang, Fault detection using the k-nearest neighbor rule for semiconductor manufacturing processes, *IEEE Transaction on Semiconductor Manufacturing*, 2007

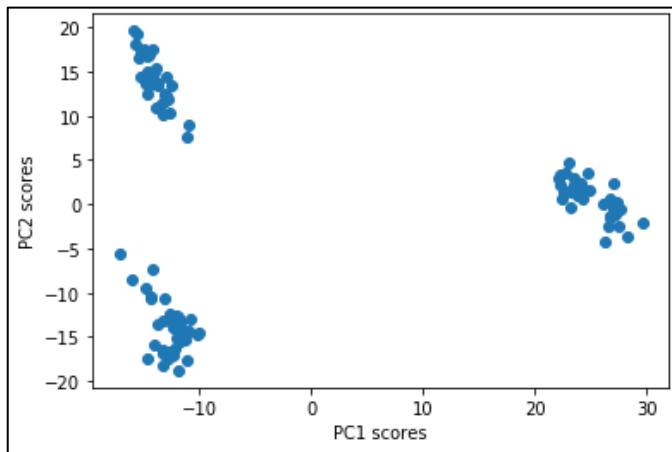


Figure 11.6: Score plot of PC1 and PC2 for calibration batches in metal etch dataset

Figure 11.6 confirms existence of 3 operating modes. While visual inspection of score plots can help to decide the number of clusters, we will, nonetheless, learn ways to estimate this in a more automated way.

11.3 K-Means Clustering: An Introduction

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

$$SSE = \sum_{k=1}^K \sum_{x_i \in k^{th} \text{cluster}} \|x_i - \mu_k\|_2^2 \quad \text{eq. 1}$$

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

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

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

```
# fit k-means model
from sklearn.cluster import KMeans

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

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

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

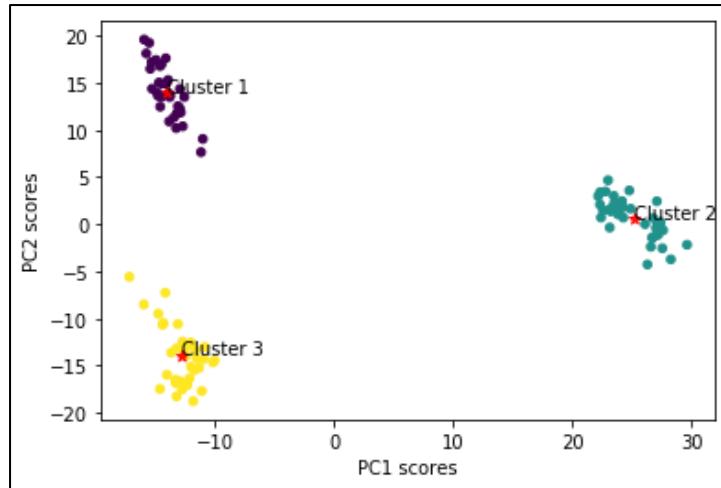


Figure 11.7: Clustering metal-etch training dataset via k-means

As expected, Figure 11.7 shows that k-means does a good job at cluster assignment. K-means clustering results are strongly influenced by initial selection of cluster centers; a bad selection can result in improper clustering. To overcome this, k-means algorithm allows a parameter, `n_init` (default value is 10), which determines the number of times independent k-means clustering is performed with different initial centroids assignment; the clustering with the lowest SSE is selected as the final model. The strategy for selection of initial centroids can also be changed via `init` parameter; the default *k-means++* option adopts a smarter (compared to the *random* option) way to speed up convergence by ensuring that the initial centroids are far away from each-other.

Determining the number of clusters via elbow method

One of the shortcomings in k-means methodology is the need to specify the number of clusters which is not known a priori for large process datasets. However, we can use the elbow method to overcome this issue. For k-means, the method entails computing the cluster inertia or SSE for different number (K) of clusters. Expectedly, when K increases, SSE decreases as data-points get closer to their assigned centroids. The value of K where only minor improvement in SSE occurs upon increasing K can be regarded as an optimal value. Figure 11.8 shows that $K= 3$ is a good choice for metal etch dataset.

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

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

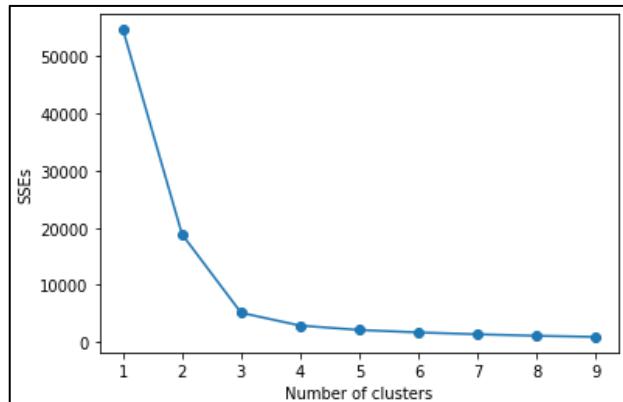


Figure 11.8: Cluster inertias for different number of clusters for metal-etch dataset



Determining the number of clusters or local models should not be taken lightly. We saw in Figure 11.1 that less than optimal number of clusters leads to low soft sensing accuracy and high frequency of missing alarms. On the other hand, using too many clusters increases the number of estimated parameters and consequently, the convergence time. Too many clusters also lead to overfitting, resulting in high frequency of false alarms and low soft sensing accuracy.

Silhouette analysis for quantifying clusters quality

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

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

Figure 11.9 shows the silhouette plot for the cluster shown in Figure 11.7. Each of the colored bands is formed by stacking the silhouette coefficient of all data-points in that cluster and therefore the thickness of the band is an indication of the cluster size. The overall silhouette score is simply the average of silhouette coefficients of all the data-points. As expected, average score is high and cluster 2 shows highest coefficients as it is far away from the other two clusters.

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

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

>>> Average silhouette score is : 0.7444602567351603

# silhouette plot
from matplotlib import cm

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

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

```

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

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

```

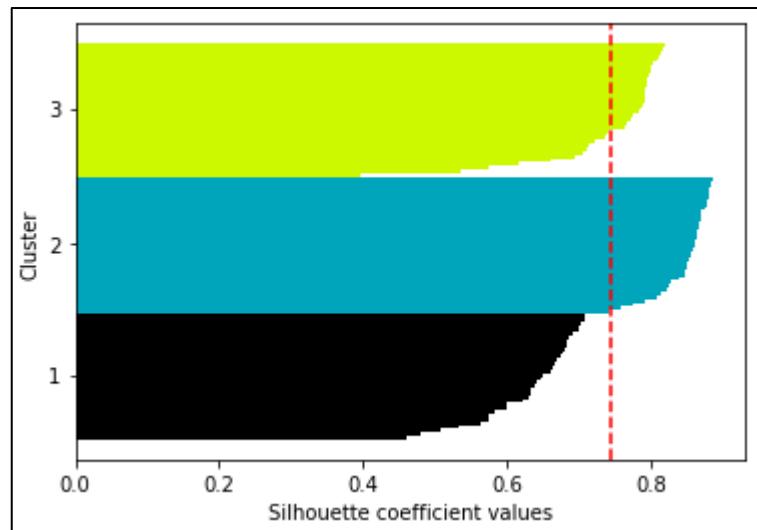


Figure 11.9: Silhouette plot for metal-etch data with 3 clusters determined via k-means. Red dashed line denotes the average silhouette value.

For comparison, let's look at a silhouette plot for a sub-optimal clustering in Figure 11.10. Lower sample-wise coefficients and lower overall score clearly indicate worse clustering.

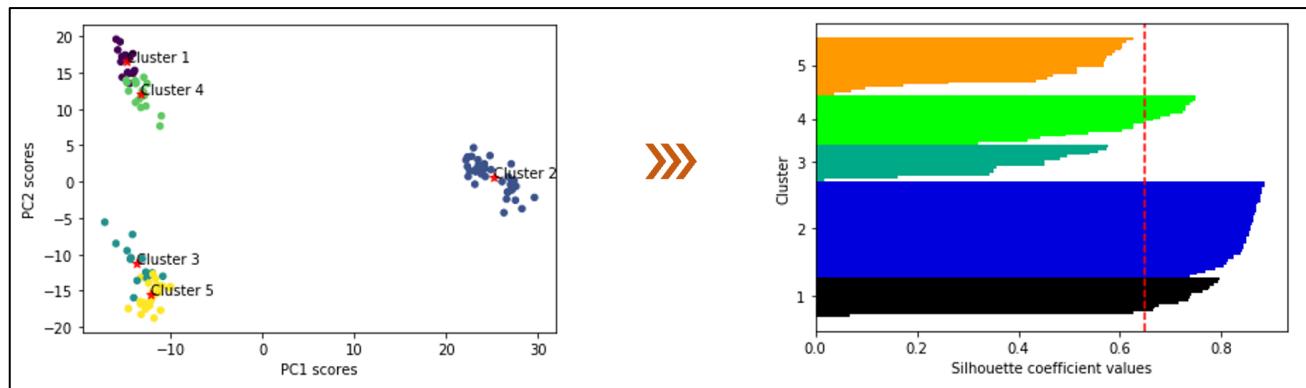


Figure 11.10: Silhouette plot for a sub-optimal clustering example

Pros and cons

As alluded to before, ease of application and simple concept are advantages with k-means algorithm. However, a serious shortcoming in this algorithm is the inability to deal with complicated geometries as shown in Figure 11.11. The next clustering algorithm that we will study is able to handle such clusters.

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

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

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

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

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

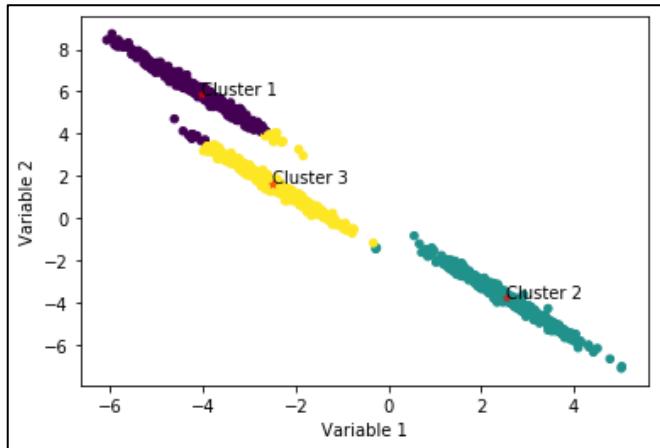


Figure 11.11: k-means clustering failure

With the clusters determined, we can go ahead and build separate models for each cluster and then monitor an incoming test sample using the cluster whose centroid is closest. However, let us move quickly to the next clustering algorithm that naturally lends itself to creation of fused decision metrics for fault detection.

11.4 Gaussian Mixture Modeling: An Introduction

Gaussian mixture model (GMM) is a clustering technique that works under the assumption that even when the overall process data does not follow a (unimodal) Gaussian distribution, it may still be appropriate to characterize data from each individual operating mode/cluster through local Gaussian distributions. GMM find the centroids and covariances of the local clusters automatically once the number of clusters have been specified. As shown in Figure 11.12, it works very well for multimodal data distributions.

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

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

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

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

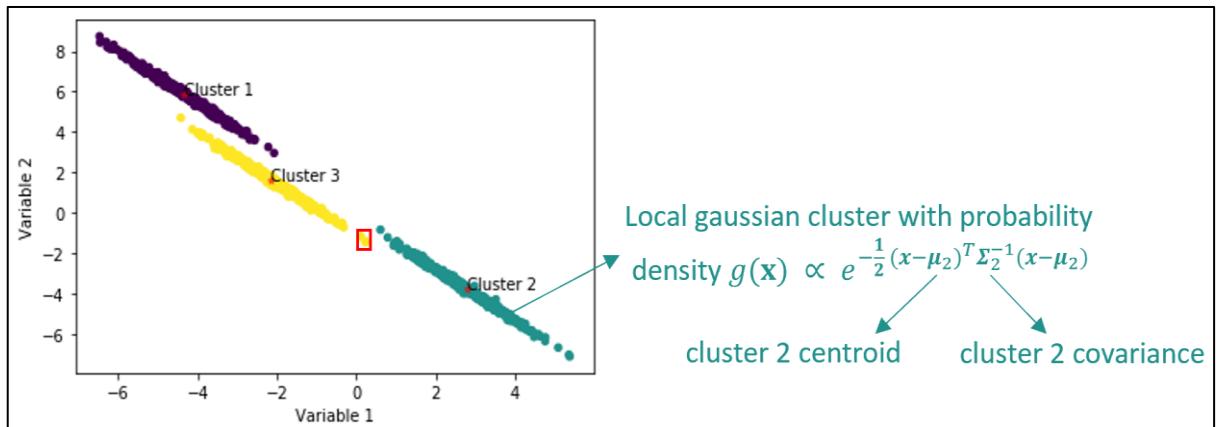


Figure 11.12: GMM based clustering of ellipsoidal data distribution

Another big advantage with GMMs is that we can compute the (posterior) probability of a data-point belonging to any cluster. This cluster membership measure is provided by `predict_proba` method. Hard clustering is performed by assigning the data-point to the cluster with highest probability. Let's compute the probabilities for a data-point that lies between clusters 3 and 2 (encircled in Figure 11.12).

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

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

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

Mathematical background

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

$$g(x|\theta_i) = \frac{1}{(2\pi)^{m/2} |\Sigma_i|^{1/2}} \exp\left[-\frac{1}{2}(x - \mu_i)^T \Sigma_i^{-1} (x - \mu_i)\right] \quad \text{eq. 2}$$

The overall density at any spatial location is given by

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

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

$$\sum_{j=1}^N \log \left(\sum_{i=1}^K \omega_i g(x_j|\theta_i) \right) \quad \text{eq. 4}$$

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

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

$$P^{(s)}(C_i|x_j) = \frac{\omega_i^{(s)} g(x_j | \mu_i^{(s)}, \Sigma_i^{(s)})}{\sum_{k=1}^K \omega_k^{(s)} g(x_j | \mu_k^{(s)}, \Sigma_k^{(s)})} \quad \text{eq. 5}$$

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

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

$$\begin{aligned}\mu_i^{(s+1)} &= \frac{\sum_{j=1}^N P^{(s)}(C_i|x_j)x_j}{\sum_{j=1}^N P^{(s)}(C_i|x_j)} \\ \Sigma_i^{(s+1)} &= \frac{\sum_{j=1}^N P^{(s)}(C_i|x_j)(x_j - \mu_i^{(s+1)})(x_j - \mu_i^{(s+1)})^T}{\sum_{j=1}^N P^{(s)}(C_i|x_j)} \\ \omega_i^{(s+1)} &= \frac{\sum_{j=1}^N P^{(s)}(C_i|x_j)}{N}\end{aligned}$$

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

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

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

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

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

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

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

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

Determining the number of clusters

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

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

    if BIC < lowestBIC:
        optimal_n_cluster = n_cluster
        lowestBIC = BIC

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

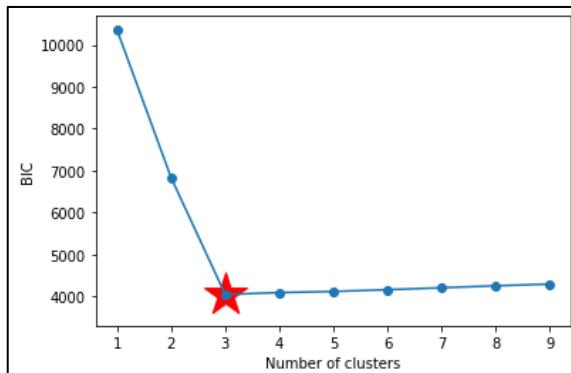


Figure 11.13: BIC plot for determining number of GMM components

There is another method, called F-J algorithm⁷⁷, which can be used to find the optimal number of GMM components⁷⁸ and model parameters in an integrated manner. Also, the number of components does not need to be specified beforehand in F-J method. Internally, the method initializes with a large number of components and adaptively adjusts this number by eliminating Gaussian components with insignificant weights. F-J method also utilizes EM

⁷⁷ Figueiredo & Jain, Unsupervised learning of finite mixture models, *IEEE Trans Pattern Anal Mach Intell*, 2002

⁷⁸ Yu & Qin, Multimode process monitoring with Bayesian inference-based finite Gaussian mixture models, *AIChE Journal*, 2008

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

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

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

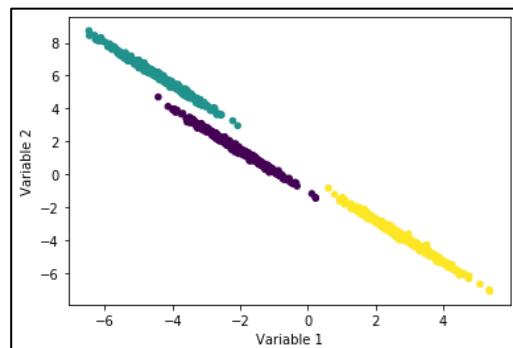


Figure 11.14: GMM based clustering of ellipsoidal data distribution via F-J method

Figure 11.15 shows the clustering for metal etch data via GMM method. BIC method correctly identifies the optimal number of components. Note that F-J method results in 4 local clusters for this dataset.

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

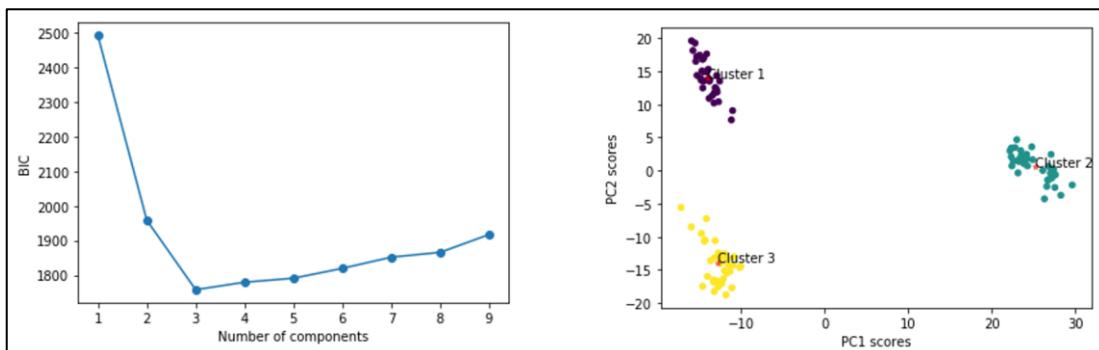


Figure 11.15: BIC plot and GMM clustering of metal etch data

11.5 Fault detection via GMM: Semiconductor Manufacturing Case Study

Due to the probabilistic formulation, GMM is widely applied for monitoring process systems. In this section, we will study one such application for the metal etch process. Figure 11.16 shows the metal etch calibration and faulty batches in the PCA score space. It is apparent that the faulty batches tend to lie away from the calibration clusters. Our objective is to develop a GMM-based monitoring tool that can automatically detect these faulty batches.

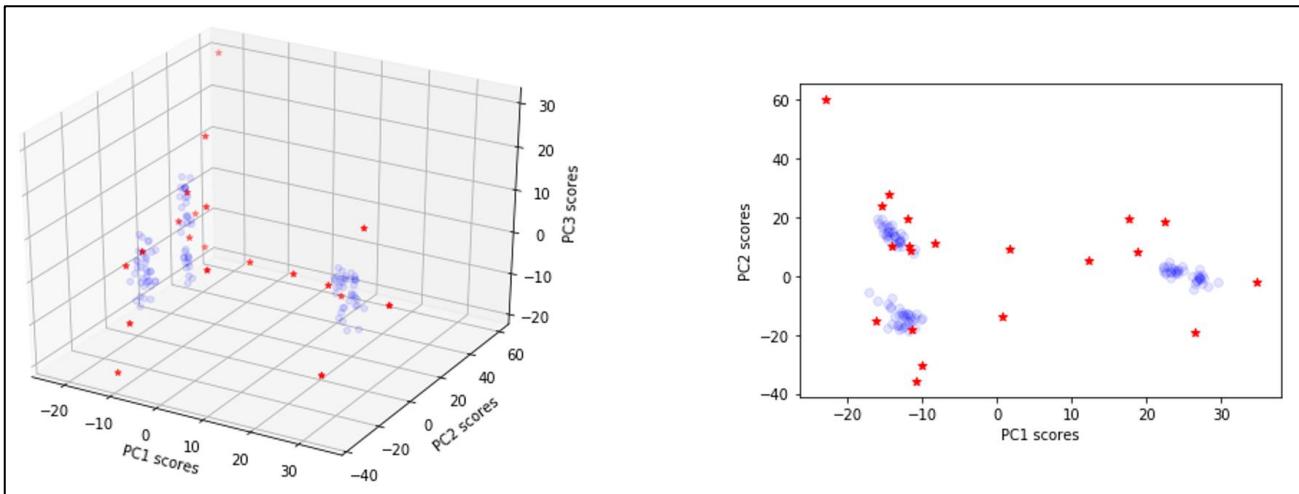


Figure 11.16: Calibration (in blue) and faulty (in red) batches in PCA score space

Fault detection index

A straightforward approach to detect a faulty batch would be to build separate ellipsoidal boundary or T^2 control chart for each cluster/mode and check if the T^2 metric for the faulty batch falls outside the control limits for all the clusters. However, monitoring many control charts can become tedious when the number of Gaussian components is high. For convenience, the local metrics/charts can be combined to generate a single global metric/chart. We will follow the approach of Xie & Shi⁷⁹ (an alternative approach is provided by Yu & Qin⁷⁷). For a test data point, x_t , local Mahalanobis distance is computed from each cluster

$$D_{local}^{(k)}(x_t) = (x_t - \mu_k)^T \Sigma_k^{-1} (x_t - \mu_k) \quad \text{eq. 6}$$

⁷⁹ Xie & Shi, Dynamic multimode process modeling and monitoring using adaptive Gaussian mixture models, Industrial & Engineering Chemistry Research, 2012

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

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

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

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

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

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

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

    for component in range(3):
        Dlocal = np.dot(np.dot((x-gmm.means_[component,:,:np.newaxis]).T,
                               np.linalg.inv(gmm.covariances_[component,:,:])),
                       (x-gmm.means_[component,:,:np.newaxis]))
        Dglobal_train[i] = Dglobal_train[i] + probs[0,component]*Dlocal

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

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

Figure 11.17 shows the control chart for the training data.

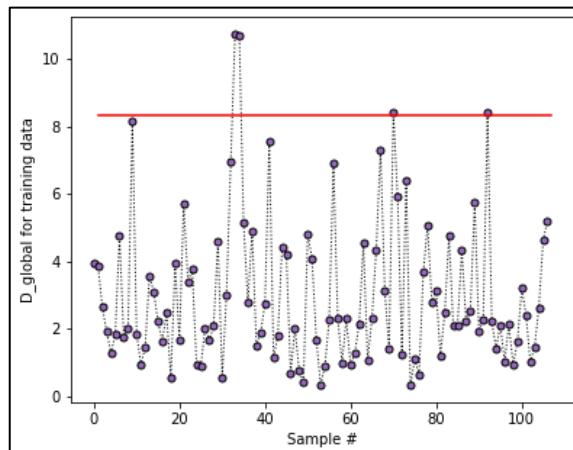


Figure 11.17: Global monitoring chart for metal etch calibration data

Fault detection for test data

Figure 11.18 shows that the global distance metric seems to be a suitable metric in detecting the faulty batches. The monitoring metric violates the threshold for 18 out of 20 batches. This good monitoring performance can be attributed to the proper multimode modeling of the data.

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

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

    if test_expt.shape[0] < 85:
        continue

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

unfolded_TestdataMatrix = unfolded_TestdataMatrix[1,:,:]

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

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

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

for component in range(3):

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

$$(x-gmm.means_[component,:,:np.newaxis]))$$

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

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

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

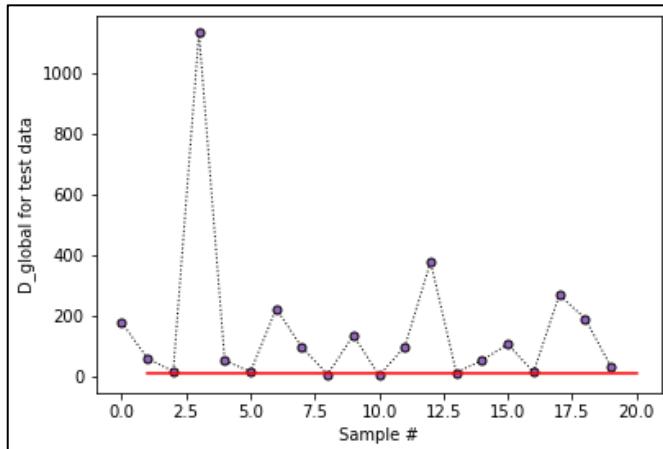


Figure 11.18: Global monitoring chart for test data

Summary

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

Part 4

Classical Machine Learning Methods for Process Monitoring

Chapter 12

Support Vector Machines for Fault Detection

In the previous chapters, we focused on multivariate statistical process monitoring methods that modelled process data through extraction of latent variables. In this part of the book, we will cover several classical ML techniques that come in handy in building process monitoring applications. These techniques do not attempt to build any statistical model of the underlying data distribution. Rather, the measurement space itself may be segregated into favorable/unfavorable regions, high-density/low-density regions, or pair-wise distances maybe computed to generate monitoring metrics, etc. SVM (support vector machine) is one such algorithm which excels in dealing with high-dimensional, nonlinear, and small or medium-sized data.

SVMs are extremely versatile and can be employed for classification and regression tasks in both supervised and unsupervised settings. SVMs, by design, minimize overfitting to provide excellent generalization performance. Infact, before ANNs became the craze in ML community, SVMs were the toast of the town. Even today, SVM is a must-have tool in every ML practitioner's toolkit. You will find more about SVMs as you work through this chapter. In terms of uses in process industry, SVMs have been employed for fault classification, fault detection, outlier detection, soft sensing, etc. We will focus on process monitoring-related usage in this chapter.

To understand different aspects of SVMs, we will cover the following topics

- Fundamentals of SVMs
- Kernel SVMs
- SVDD (support vector data description) for unsupervised classification
- Fault detection via SVDD for semiconductor manufacturing process

12.1 SVMs: An Introduction

The classical SVM is a supervised linear technique for solving binary classification problems. For illustration, consider Figure 12.1a. Here, in a 2D system, the training data-points belong to two distinct (positive and negative) classes. The task is to find a line/linear boundary that separates these 2 classes. Two sample candidate lines are also shown. While these lines clearly do the stated job, something seems amiss. Each of them passes very close to some of the training data-points. This can cause poor generalization: for example, the shown test observation ‘A’ lies closer to the positive samples but will get classified as negative class by boundary L2. This clearly is undesirable.

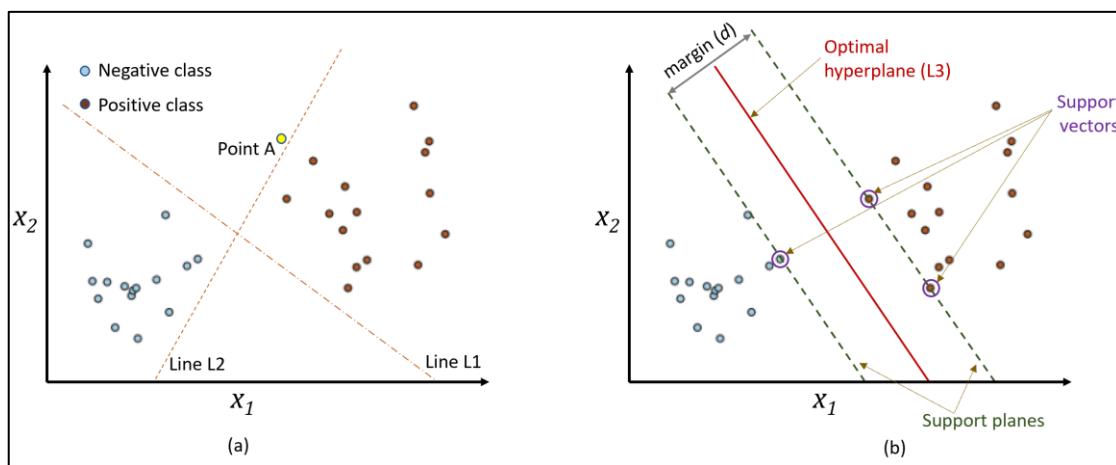


Figure 12.1: (a) Training data with test sample A (b) Optimal separating boundary

The optimal separating line/decision boundary, line L3 in Figure 12.1b, lies as far away as possible from either class of data. L3, as shown, lies midway of the support planes (planes that pass-through training points closest to the separating boundary). During model fitting, SVM simply finds this optimal boundary that corresponds to the maximum margin (distance between the support planes). In Figure 12.1, any other orientation or position of L3 will reduce the margin and will make L3 closer to one class than to the other. Large margins make model predictions robust to small perturbations in the training samples.

Points that lie on the support planes are called support vectors⁸⁰ and completely determine the optimal boundary, and hence the name, support vector machines. In Figure 12.1, if support vectors are moved, line L3 may change. However, if any non-support vectors are removed, L3 won’t get affected at all. We will see later how the sole dependency on the support vectors imparts computational advantage to the SVMs.

⁸⁰ Calling data-points as vectors may seem weird. While this terminology is commonly used in general SVM literature, support vectors refer to the vectors originating from origin with the data-points on support planes as their tips.

Mathematical background

Let there be N training samples (x, y) where x is an input vector in m -dimensional input space and y is the class label (± 1). Let the optimal separating hyperplane (a line in 2D space) be represented as $w^T x + b = 0$ where the model parameters (w, b) are found such that

$$\begin{aligned} w^T x_i + b &\geq 1 && \text{for positive samples } (y_i = 1) \\ w^T x_i + b &\leq -1 && \text{for negative samples } (y_i = -1) \end{aligned}$$

The supporting planes get represented as $w^T x + b = 1$ and $w^T x + b = -1$. The above equations are simply stating the requirements that the data points must lie on the correct side of their corresponding support planes. Using simple trigonometry, it can be shown that the margin is given by $\frac{2}{\|w\|}$ where $\|w\|$ is vector norm. Therefore, model parameters are found by maximizing $\frac{2}{\|w\|}$ (or equivalently, minimizing $\|w\|$) while meeting the above constraints. Specifically, the following optimization problem is solved

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} \|w\|^2 \\ \text{s.t.} \quad & y_i(w^T x_i + b) \geq 1, \quad i = 1, \dots, N \end{aligned} \tag{eq. 1}$$

Once the model has been fitted, class predictions for test sample, x_t , are made as follows.

$$\hat{y}_t = \text{sign}(w^T x_t + b) \rightarrow x_t \text{ belongs to positive class if it lies on positive side of separating hyperplane}$$

The expression inside the sign function is also called decision function and therefore, positive decision function results in positive class prediction and vice-versa.



The optimization formulation in Eq. 1 and all the others that we will see in this chapter share a very favorable property of possessing a unique global minimum. This is a huge advantage when compared to other powerful ML methods like neural networks where the issue of local minimums can be an inconvenience.

Simple linear classification illustration

We will use the toy dataset in Figure 12.1 to illustrate SVM-based classifier modeling in Sklearn.

```
# read data
import numpy as np
data = np.loadtxt('toyDataset.csv', delimiter=',')
X = data[:, [0, 1]]; y = data[:, 2]

# scale model inputs
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# fit SVM model
from sklearn.svm import SVC # for large datasets LinearSVC class is preferable
model = SVC(kernel='linear', C=100)
model.fit(X_scaled, y)

# get details of support vectors
print('# of support vectors:', len(model.support_))

>> # of support vectors: 3
```

The above code provides us the optimal separating boundary shown in Figure 12.1⁸¹. As with other Sklearn estimators, `predict()` method can be used to predict the class of any test observation. We will soon cover the hyperparameters (kernel and C) used in the above code.

Hard margin vs soft margin classification

The toy dataset represents an ideal data where all the training samples can be correctly classified via a linear boundary. However, in real-life, some data impurities always creep in as shown in Figure 12.2. Our SVM formulation (in Eq. 1) will fail to find a solution in this case.

⁸¹ Check out the online code to see how the separating boundary and support planes are plotted

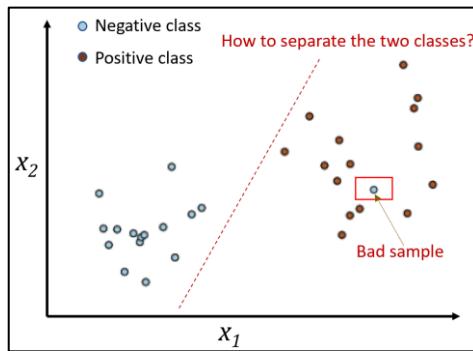


Figure 12.2: Presence of the shown bad sample makes perfect linear separation infeasible

To deal with such scenarios, we add a little flexibility into our SVM optimization program by modifying the constraints as shown below

$$\begin{aligned} w^T x_i + b &\geq 1 - \xi_i && \text{for } y_i = 1 \\ w^T x_i + b &\leq -1 + \xi_i && \text{for } y_i = -1 \end{aligned}$$

Here, we use slack variables (ξ_i) to allow each sample the freedom to end up on the wrong side of the support plane and potentially be misclassified during model fitting. However, we would like to keep the number of such violations low as well which we can achieve by penalizing the violations. The revised SVM formulation looks like this

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2} \|w\|^2 + C \sum_{i=1}^N \xi_i \\ \text{s.t.} \quad & y_i(w^T x_i + b) \geq 1 - \xi_i, \quad i = 1, \dots, N \\ & \xi_i \geq 0 \end{aligned} \quad \text{eq. 2}$$

The above formulation is called soft margin classification (as opposed to the previous hard margin classification). Sklearn implements soft margin formulation. The positive constant, C , is a hyperparameter ($C=1$ in Sklearn by default) and corresponds to the hyperparameter we saw in the previous code. For our toy dataset 2 (in Figure 12.2), with the previously shown code, we end up with the same separating boundary as shown in Figure 12.1. Class prediction expression remains the same as $\hat{y}_t = \text{sign}(w^T x_t + b)$.

C as regularization hyperparameter

The slack variables not only help find a solution in the presence of gross impurity, but they also help to avoid overfitting noisy data. For example, consider the scenario in Figure 12.3. If no misclassifications are allowed, we end up with a very small margin, while with a single

misclassification we get a much better margin with potentially better generalization. Therefore, we see that there is a trade-off between margin maximization and training accuracy.

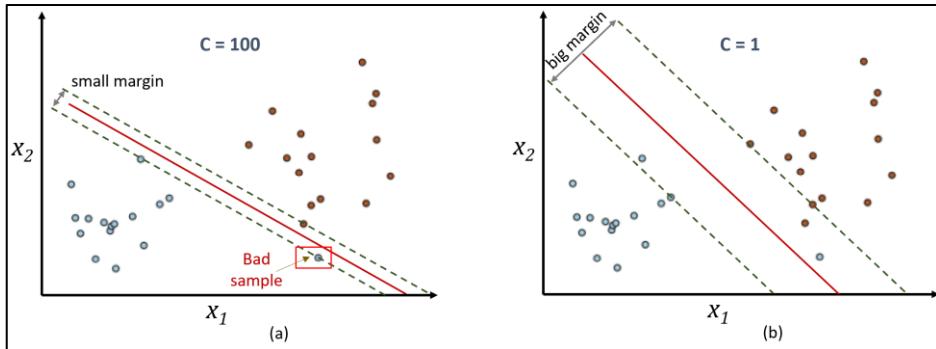


Figure 12.3: (a) Overfitted unregularized boundary (b) Regularized fit

The hyperparameter C is the knob to control the trade-off. A large value of C implies heavy penalization of the constraint violations which will prevent misclassifications, while small value of C allows more misclassifications during model fitting for better margin.

12.2 The Kernel Trick for Nonlinear Data

While soft margin classification formulation is quite flexible, it won't work for nonlinear classification problems where curved boundaries are warranted. Consider the dataset in Figure 12.4. It is clear that a linear boundary is inappropriate here.

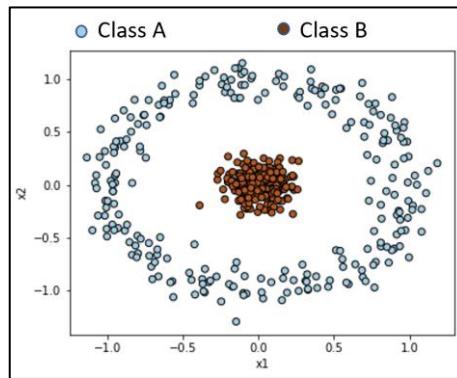


Figure 12.4: Nonlinearly distributed samples

However, all is not lost here. One idea to circumvent this issue is to map the original input variables/features into a higher dimensional space where they become linearly separable. For the data in Figure 12.4, the following transformation would work quite well

$$\varphi(x) = \varphi(x_1, x_2) = [z_1, z_2, z_3] = [x_1, x_2, x_1^2 + x_2^2]$$

As we can see in Figure 12.5, the data is easily linearly separable in the 3D space!

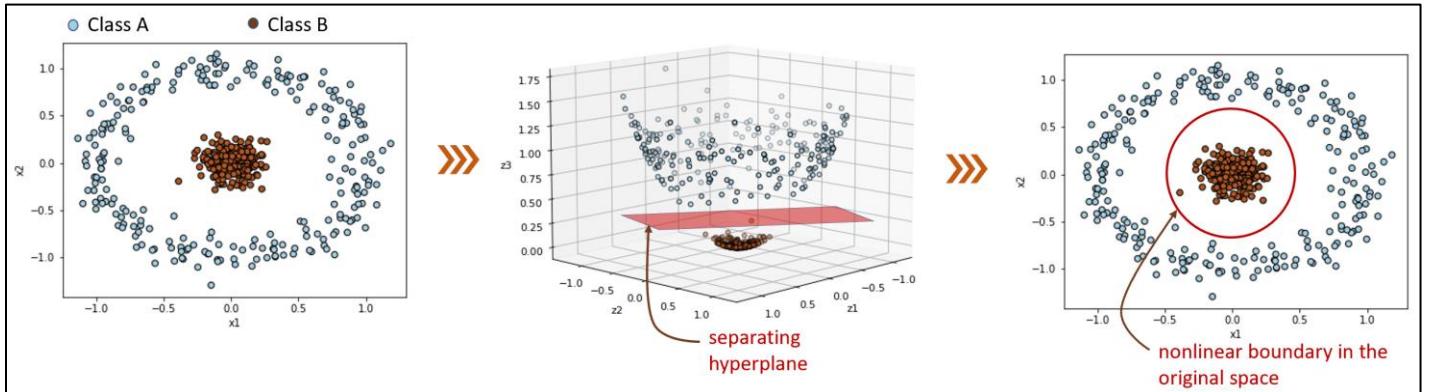


Figure 12.5: Nonlinear mapping to higher dimensional enabling linear segregation

SVM can be trained on the new feature space to obtain the optimal separating hyperplane. Any new test data point can be transformed via the same mapping function, φ , for its class determination via the fitted SVM model. While this solution looks great, there remains a small issue. How do we find an appropriate mapping for a high-dimensional input dataset? As it turns out, you don't need to find this mapping explicitly and this is made possible by a neat 'kernel trick'. Yes, the same kernel trick that we saw in Chapter 10! To see how kernels show up in SVM formulation, we will revisit the mathematical background of SVMs.

Mathematical background (revisited)

SVMs are not solved in the form shown in Eq. 2. Instead, the following equivalent form is solved

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j x_i^T x_j - \sum_{i=1}^N \alpha_i \\ \text{s.t.} \quad & \sum_{i=1}^N y_i \alpha_i = 0 \\ & 0 \leq \alpha_i \leq C \quad i = 1, \dots, N \end{aligned} \quad \text{eq. 3}$$

Here, the optimization parameters w, b have been replaced by α_s (also called Lagrange multipliers). This equivalent form is called dual form (which you may remember from your optimization classes; it's perfectly fine if you have not encountered this term before). Once α_s have been found, w and b can be computed via the following

$$w = \sum_{i=1}^N \alpha_i y_i x_i, \quad b = \frac{1}{N_s} \sum_{i \in \{SV\}} 1 - y_i w^T x_i$$

where N_s is number of support vectors and $\{SV\}$ is the set of support vector indices. Any test data point can be classified as

$$\hat{y}_t = \text{sign}(\sum_{i=1}^N \alpha_i y_i x_i^T x_t + b) \quad \text{eq. 4}$$

In the dual formulation, it is found that α_s are non-zero for only the support vectors and zero for the rest of the training samples. This implies that Eq. 4 can be reduced to

$$\hat{y}_t = \text{sign}(\sum_{i \in \{SV\}} \alpha_i y_i x_i^T x_t + b) \quad \text{eq. 5}$$

 Strictly speaking, support vectors need not lie on the separating hyperplane. For soft margin classification, data-points with non-zero slacks are also support vectors and their α_s are non-zero (defining characteristic of the support vectors). The presence/absence of the support vectors impacts the solution (the objective function and/or the model parameters).

At this point, you may be wondering why have we made things more complicated; why not solve the problem in the original form (Eq. 2) which seemed more interpretable? The reason for doing this will become clear to you very soon. For now, imagine that you are solving the nonlinear problem where SVM finds a separating hyperplane in the higher dimension. Eq. 3 will look like the following

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \varphi(x_i)^T \varphi(x_j) - \sum_{i=1}^N \alpha_i \\ \text{s.t.} \quad & \sum_{i=1}^N y_i \alpha_i = 0 \\ & 0 \leq \alpha_i \leq C \quad i = 1, \dots, N \end{aligned} \quad \text{eq. 6}$$

and Eq. 5 becomes

$$\hat{y}_t = \text{sign}(\sum_{i \in \{SV\}} \alpha_i y_i \varphi(x_i)^T \varphi(x_t) + b) \quad \text{eq. 7}$$

The most crucial observation here is that the transformed variables ($\varphi(x)$) appear only as inner (dot) products. This allows us to use the kernel trick. Once a kernel function is chosen, Eq. 6 becomes

$$\begin{aligned}
 \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j k(x_i, x_j) - \sum_{i=1}^N \alpha_i \\
 \text{s.t.} \quad & \sum_{i=1}^N y_i \alpha_i = 0 \\
 & 0 \leq \alpha_i \leq C \quad i = 1, \dots, N
 \end{aligned} \tag{eq. 8}$$

Above is the form in which a SVM model is fitted and predictions are made as follows

$$\hat{y}_t = \text{sign}(\sum_{i \in \{SV\}} \alpha_i y_i k(x_i, x_t) + b) \tag{eq. 9}$$

Like in KPCA and KPLS, usage of kernel functions allows us to obtain powerful nonlinear classifiers while retaining all the benefits of the original linear SVM method!

Sklearn implementation of support vector classifier

Let's try to find the nonlinear classifier boundary for the toy dataset in Figure 12.4 using gaussian kernel. We will also find optimal values of C and σ via grid-search and cross-validation.

```

# generate data
from sklearn.datasets import make_circles
X, y = make_circles(500, factor=.08, noise=.1, random_state=1)
# note that y = 0,1 here and need not be ±1; SVM does internal transformation accordingly

# find optimal hyperparameter via GridSearchCV
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

param_grid = {'C':[0.1, 1, 10, 100, 1000], 'gamma':[0.01, 0.1, 1, 10, 100]}
gs = GridSearchCV(SVC(), param_grid, cv=5).fit(X, y) # no scaling as inputs are already scaled

print('Optimal hyperparameter:', gs.best_params_)

>>> Optimal hyperparameter: {'C': 0.1, 'gamma': 1}

```

You will notice that Sklearn uses the hyperparameter γ which is simply $1/\sigma^2$. Optimal C and γ come out to be 0.1 and 1, respectively, with the classifier solution shown in Figure 12.6. The figure also shows the boundary regions for low and high values of the hyperparameters. As we saw before, large C leads to overfitting (boundary impacted by the noise). As far as γ is concerned, large value (or small σ) also leads to overfitting.

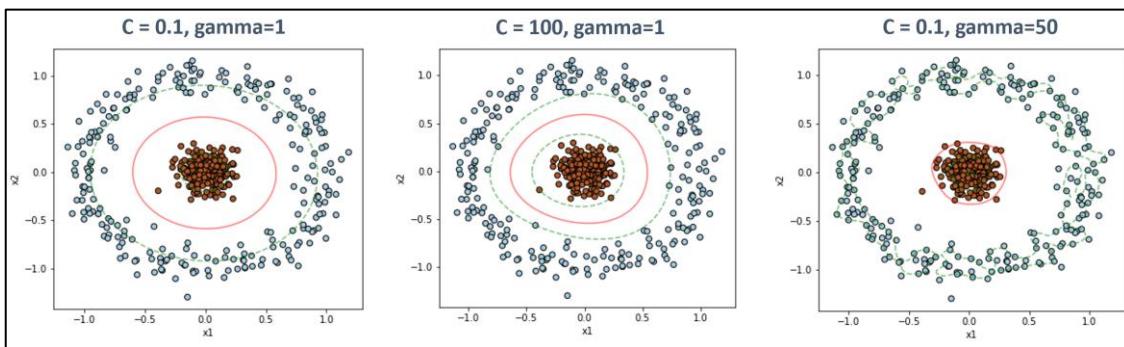


Figure 12.6: Nonlinear binary classification via kernel SVM and impact of hyperparameters.
[Code for plotting the boundaries is provided online]

We will look at an industrial-scale application of SVM for fault classification of rotating machines using vibration signals in Chapter 18.

12.3 SVDD: An Introduction

Support vector data description (SVDD) is the unsupervised form of SVM algorithm used for dealing with problems where training samples belong to only one class and the model objective is to determine if any test/new sample belongs to the same class of data or not. Consider the motivating example in Figure 12.7. Here, how do we obtain a model of the training data to be able to call sample A an outlier/abnormal? Such problems are quite common in process industry. Process monitoring and equipment health monitoring are some example areas where most/all the available process data may belong to only normal plant operations class and the modeling objective is to identify any abnormal data-point.

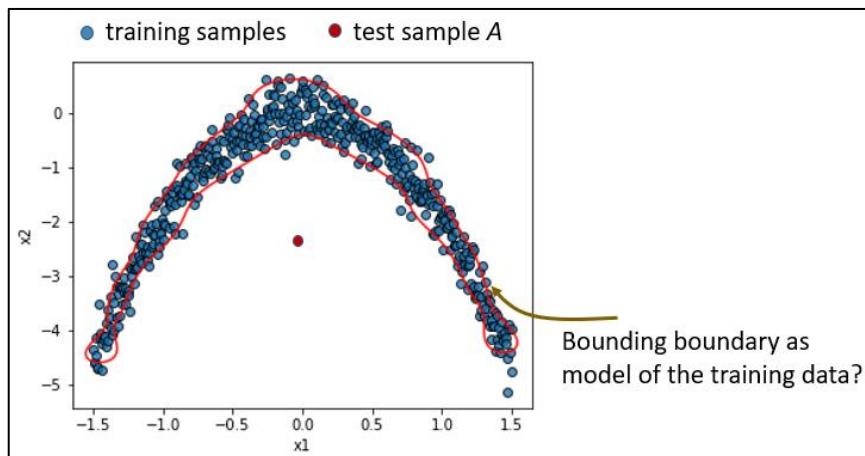
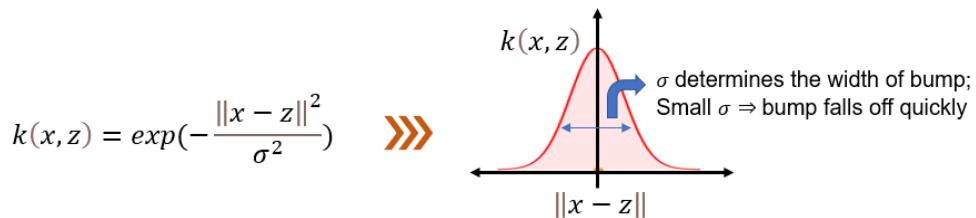


Figure 12.7: 2D training dataset with only one class. Boundary in red shows a potential description model of the dataset that can be used to distinguish the test sample A from training samples.

Gaussian kernels and classification

A better intuition behind the kernels can help us understand the impact of kernel hyperparameters on the classification boundaries. Kernels provide an indirect measure of similarity between any two points in the high-dimensional space. Consider again the Gaussian kernel

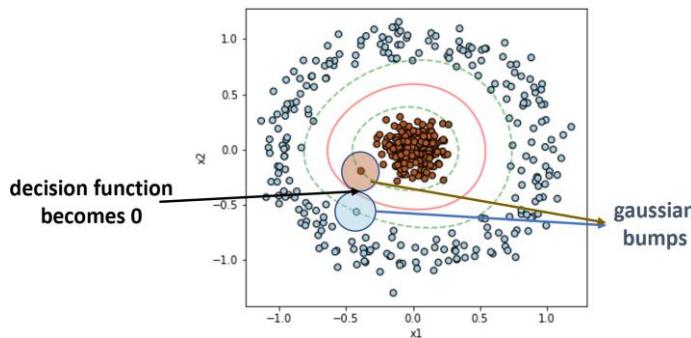


Here, if two points (x, z) are close to each-other in the original space, then their similarity (or kernel value) in the mapped space will be higher, compared to when x and z are far away from each-other. Now, let's look at the classifier prediction formula for a sample z

$$\hat{y} = \text{sign}\left(\sum_{i \in \{SV\}} \alpha_i y_i k(x_i, z) + b\right)$$

Sum of gaussian bumps

Therefore, the classifier is nothing but a sum of Gaussian bumps from the support vectors (plus an offset b)!



Given bandwidth (σ), during training, SVM tries to find the optimal values of the bump multipliers (α_s) such that the training samples get correct labels while keeping maximum separation between classification boundary and training samples. The boundary is simply the set of points where the net summations of bumps and offset become zero. Small values of σ lead to very localized bumps near any support vector, resulting in higher number of support vectors with too much 'wiggles' in the separating boundary which often indicates overfitting.

The idea behind SVDD is to envelop training data by a hypersphere (circle in 2D, sphere in 3D) containing maximum number of data-points within a smallest volume. Any new observation that lies farther than the hypersphere radius from hypersphere center can be regarded as abnormal observation. But the data in Figure 12.7 don't look like it can be suitably enveloped with a circle? That is correct and our recourse is to use kernel functions to implicitly project original data onto a higher dimensional space where data can be adequately enveloped within a compact hypersphere. The projection of the optimal hyperplane onto the original space will show up as a tight nonlinear boundary around the dataset!

Just like classical 2-class SVM, only a small set of training samples get to completely define the hypersphere. These data-points or support vectors lie on the circumference or outside of the hypersphere (or the nonlinear boundary in the original space).

Mathematical background

Assume again that $\varphi(x)$ represents a data-point in the higher dimensional feature space. In this space, the optimal hypersphere is found via the following optimization problem

$$\begin{aligned} \min_{R,a,\xi} \quad & R^2 + C \sum_{i=1}^N \xi_i \\ \text{s.t.} \quad & \|\varphi(x_i) - a\|^2 \leq R^2 + \xi_i, \quad i = 1, \dots, N \\ & \xi_i \geq 0 \end{aligned}$$

As is evident, the above program is trying to minimize the radius (R) of the hypersphere centered at 'a' such that most of the data-points lie within the hypersphere. Slack variables, ξ , allow certain samples to fall outside and the number of such violations is tuned via the hyperparameter C . As before, the problem is solved in its dual form

$$\begin{aligned} \min_{\alpha} \quad & \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j K(x_i, x_j) - \sum_{i=1}^N \alpha_i K(x_i, x_i) \\ \text{s.t.} \quad & \sum_{i=1}^N \alpha_i = 1 \\ & 0 \leq \alpha_i \leq C \quad i = 1, \dots, N \end{aligned}$$

Like SVM, the alphas indicate the position of training samples w.r.t. the optimal boundary. The following relationships hold true

Inside the boundary:	$\ \varphi(x_i) - a\ < R \Rightarrow \alpha_i = 0$
On the boundary:	$\ \varphi(x_i) - a\ = R \Rightarrow 0 < \alpha_i < C$
Outside the boundary:	$\ \varphi(x_i) - a\ > R \Rightarrow \alpha_i = C$

Support
Vectors

The optimal solution satisfies the following expression

$$R = \sqrt{k(x_s, x_s) - 2 \sum_{i \in \{SV\}} \alpha_i k(x_s, x_i) + \sum_{i \in \{SV\}} \sum_{j \in \{SV\}} \alpha_i \alpha_j k(x_i, x_j)}$$

where x_s is any support vector lying on the boundary. Any test observation x_t is abnormal if its distance from center a in the mapped space is greater than R where the distance is given as follows

$$\begin{aligned} Dist(\varphi(x_t), a)^2 &= \|\varphi(x_t) - a\|^2 \\ &= k(x_t, x_t) - 2 \sum_{i \in \{SV\}} \alpha_i k(x_t, x_i) + \sum_{i \in \{SV\}} \sum_{j \in \{SV\}} \alpha_i \alpha_j k(x_i, x_j) \end{aligned}$$

As you can see, specifications of kernel functions and other model hyperparameters is all that is needed; no knowledge of mapping φ is required.

OC-SVM vs SVDD

There is another technique closely related to SVDD, called one class SVM (OC-SVM). Infact, OC-SVM is the unsupervised SVM algorithm currently available in Sklearn. OC-SVM finds a separating hyperplane that best separates the training data from the origin. Its kernelized dual form is given by

$$\begin{array}{ll} \min_{\alpha} & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j k(x_i, x_j) \\ \text{s.t.} & \sum_{i=1}^N \alpha_i = 1 \\ & 0 \leq \alpha_i \leq C \quad i = 1, \dots, N \end{array}$$

You will notice that for Gaussian kernel, OC-SVM formulation becomes equivalent to that of SVDD because $k(x_i, x_i) = 1$ and we end up with the same values of the multipliers. The decision boundaries are the same as well. For other kernels with $k(x_i, x_i) \neq 1$, results would be different.

Bandwidth parameter & SVDD illustration

In this sub-section, we will attempt to use Sklearn's OneClassSVM to build the bounding boundary for the dataset in Figure 12.7. However, before we do that, we need to specify our strategy for determination of SVDD model hyperparameters, C and σ . If sufficient validation data samples belonging to both 'normal' and 'abnormal' classes are available, then cross-validation can be used to choose C and σ that give the best validation accuracy. However, very often this is not possible and like training dataset, validation dataset also only contains 'normal' class samples! To enable educated hyperparameter selection, let's take another look at them.

Previously, we saw that C controls the trade-off between volume of hypersphere and the number of misclassifications in the training dataset. C can also be written as

$$C = \frac{1}{Nf}$$

Where N is the number of samples and f is the expected fraction of outliers in the training dataset. Smaller value of f (correspondingly larger C) will lead to less samples being put outside the hypersphere. Infact if C is set to 1 (or greater) the hypersphere will include all the samples (as $\sum \alpha_i = 1$ and $\alpha = C$ outside the hypersphere). Therefore, C can be set with some educated presumptions on the outlier fractions. In absence of any advance knowledge, $f = 0.01$ is often specified to exclude away 1% of sample lying farthest from hypersphere center.

As far as σ is concerned, we previously saw that at low value of σ , data boundary becomes very wiggly with high number of support vectors, resulting in overfitting. Conversely, at high value of σ , boundary tends to become spherical in the original space itself resulting in underfitting (or non-compact bounding of data). One approach for bandwidth selection is to use empirical methods which are based on obtaining a kernel matrix (whose i,j^{th} element is $k(x_i, x_j)$) with favorable properties. One such method, modified mean criterion⁸², gives bandwidth as follows

$$\begin{aligned} \sigma &= \sqrt{\frac{\bar{D}^2}{\ln\left(\frac{N-1}{\delta^2}\right)}} \\ \bar{D}^2 &= \frac{\sum_{i < j} \|x_i - x_j\|^2}{\frac{N(N-1)}{2}} \\ \delta &= -0.14818008\varnothing^4 + 0.2846623624\varnothing^3 - 0.252853808\varnothing^2 + 0.159059498\varnothing - 0.001381145 \end{aligned}$$

⁸² Kalde & Sadek, The mean and median criterion for kernel bandwidth selection for support vector data description, IEEE 2017

$$\phi = \frac{1}{\ln(N-1)}$$

Another approach for bandwidth selection is to choose largest value of σ that gives the desired confidence level on the validation dataset. For example, for a confidence level of 99%, σ is increased until 99% of validation samples are correctly classified as inliers. Any higher value of σ will include more validation samples within the hypersphere. The modified mean criterion can be used as the initial guess with subsequent search made around it. Let's now find the nonlinear boundary for the dataset in Figure 12.7.

```
# read data
import numpy as np
X = np.loadtxt('SVDD_toyDataset.csv', delimiter=',')
# compute bandwidth via modified mean criteria
import scipy

N = X.shape[0]
phi = 1/np.log(N-1)
delta = -0.14818008*np.power(phi,4) + 0.2846623624*np.power(phi,3) - 0.252853808*np.power(phi,2)
    + 0.159059498*phi - 0.001381145
D2 = np.sum(scipy.spatial.distance.pdist(X, 'squared_euclidean'))/(N*(N-1)/2)
sigma = np.sqrt(D2/np.log((N-1)/delta*delta))
gamma = 1/(2*sigma*sigma)

# SVM fit
from sklearn.svm import OneClassSVM
model = OneClassSVM(nu=0.01, gamma=gamma).fit(X) # nu corresponds to f
```

Figure 12.8 shows the bounding boundary for different values of gamma with f (or ν in Sklearn) kept at 0.01. A value of gamma (= 1) close to that given by modified mean criterion method (= 0.58) provided a satisfactory boundary.

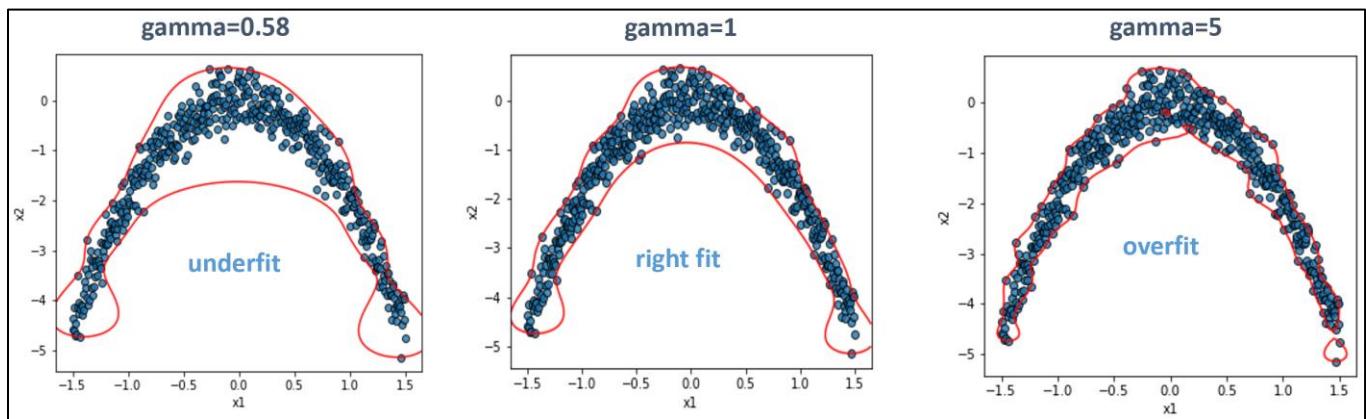


Figure 12.8: SVDD application for data description and impact of model hyperparameter.

We hope that by now you are convinced of the powerful capabilities of SVM for discriminating between different classes of data and compact bounding of normal operational data. A big requirement for successful application of SVM is that the training dataset should be very representative of the ‘normal’ dataset and fully characterize all the expected variations. Next, we will look at a case study with real process data.

12.4 Fault detection via SVDD: Semiconductor Manufacturing Case Study

To illustrate a practical application of SVDD for process monitoring, we will re-use the semiconductor manufacturing process data from Chapter 11. This batch process dataset contains 19 process variables measured over the course of 108 normal batches and 21 faulty batches. The batch durations range from 95 to 112 seconds. Figure 12.9 shows the training samples and the faulty test samples in the principal component space.

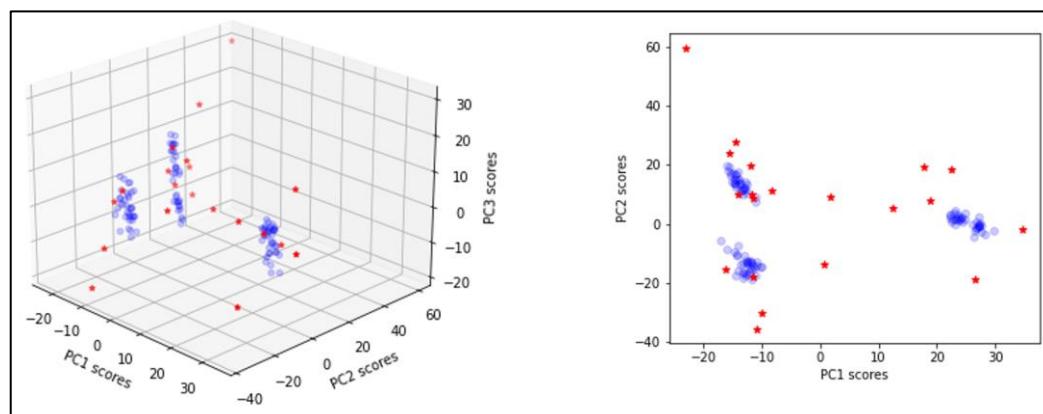


Figure 12.9: Normal (in blue) and faulty (in red) batches in PCA score space

For this illustration, the raw data has been processed using multiway PCA and the transformed 2D (score) data is provided in the *Metal_etch_2DPCA_trainingData.csv* file. Note that we could also implement SVDD in the original input space but pre-processing via PCA to remove variable correlation is generally a good practice. Moreover, we use the 2D PC space for our analysis just for the ease of illustrating the SVDD boundary. In actual deployment, you would work in higher dimensional PC space for better accuracy. Let's see if our model can identify the faulty samples as outliers or not in the multi-clustered dataset.

```
# read data
import numpy as np
X_train = np.loadtxt('Metal_etch_2DPCA_trainingData.csv', delimiter=',')

# fit SVM
from sklearn.svm import OneClassSVM
model = OneClassSVM(nu=0.01, gamma=0.025).fit(X_train) # gamma from modified mean
criterion = 0.0025

# predict for test data
X_test = np.loadtxt('Metal_etch_2DPCATestData.csv', delimiter=',')
y_test = model.predict(X_test) # y=-1 for outliers

print('Number of faults identified: ', np.sum(y_test == -1), ' out of ', len(y_test))

>>> Number of faults identified: 17 out of 20
```

Figure 12.10 shows the boundary around the training samples and the faulty samples labeled according to their correct or incorrect identification. Seventeen out of twenty faulty data samples have correctly been identified as outliers. This example illustrates the power of SVDD for compactly describing clustered datasets.

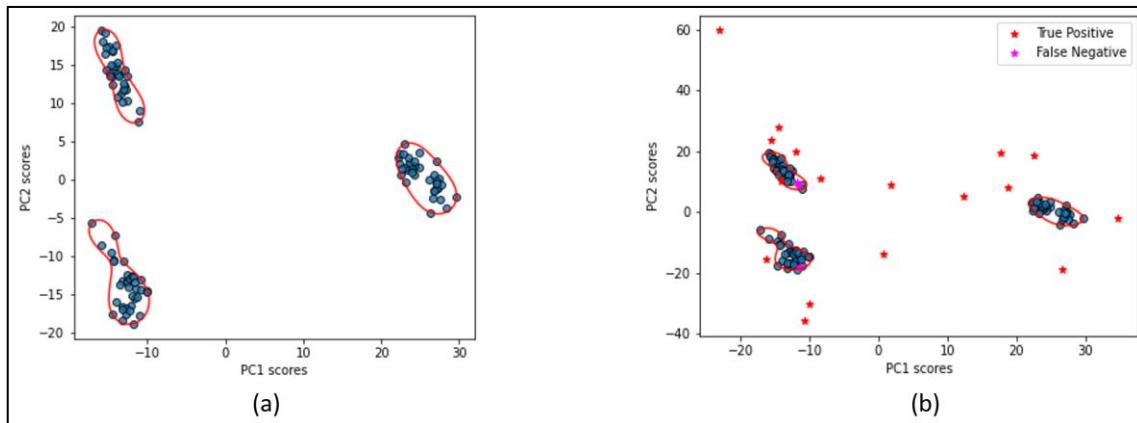


Figure 12.10: (a) SVDD/OC-SVM boundary (in red) around metal-etch training dataset in 2D PC space (b) Position of correctly and incorrectly diagnosed faulty samples

We should get the same results if we use the distances from the hypersphere center for fault detection. The results from SVDD and OC-SVM will differ if RBFs are not used as kernel. Unfortunately, Sklearn currently does not provide SVDD implementation. Nonetheless, a SVDD package is available on GitHub⁸³.

This concludes our look into support vector machines. SVMs are in a league of their own and are well-suited for industrial processes with difficult to estimate process parameters. With elegant mathematical background, just a few hyperparameters, excellent generalization capabilities, and guaranteed unique global optimum, SVMs are among the best ML algorithms.

Summary

In this chapter we studied the support vector machine algorithm and its varied forms for supervised and unsupervised machine learning. We saw its applications for binary classification, fault detection, and fault classification. Through kernelized learning, we learned the adaptation of SVM for nonlinear modeling. In summary, we have added a powerful tool to your data science toolkit. Next, we will continue building our toolkit and learn a few more powerful classical ML algorithms.

⁸³ <https://github.com/iqiukp/SVDD>

Chapter 13

Decision Trees and Ensemble Learning for Fault Detection

Imagine that you are in a situation where even after your best attempts your model could not provide satisfactory performance. What if we tell you that there exists a class of algorithms where you can combine several ‘versions’ of your ‘weak’ performing models and generate a ‘strong’ performer that can provide more accurate and robust predictions compared to its constituent ‘weak’ models? Sounds too good to be true? It’s true and these algorithms are called ensemble methods.

Ensemble methods are often a crucial component of winning entries in online ML competitions such as those on Kaggle. Ensemble learning is based on a simple philosophy that committee wisdom can be better than an individual’s wisdom! In this chapter, we will look into how this works and what makes ensembles so powerful. We will study popular ensemble methods like random forests and XGBoost.

The base constituent models in forests and XGBoost are decision trees which are simple yet versatile ML algorithms suitable for both regression and classification tasks. Decision trees can fit complex and nonlinear datasets, and yet enjoy the enviable quality of providing interpretable results. We will look at all these features in detail. Specifically, we will cover the following topics

- Introduction to decision trees and random forests
- Introduction to ensemble learning techniques (bagging, Adaboost, gradient boosting)
- Fault detection and classification for gas boilers using decision trees and XGBoost

13.1 Decision Trees: An Introduction

Decision trees (DTs) are inductive learning methods which derive explicit rules from data to make predictions. They partition the feature space into several (hyper) rectangles and then fit a simple model (usually a constant) in each one. As shown in Figure 13.1 for a binary classification problem in 2D feature space, the partition is achieved via a series of if-else statements. As shown, the model is represented using branches and leaves which lead to a tree-like structure and hence the name decision tree model. The questions asked at each node make it very clear how the model predictions (class A or class B) are being generated. Consequently, DTs become the model of choice for applications where ease of rationalization of model results is very important.

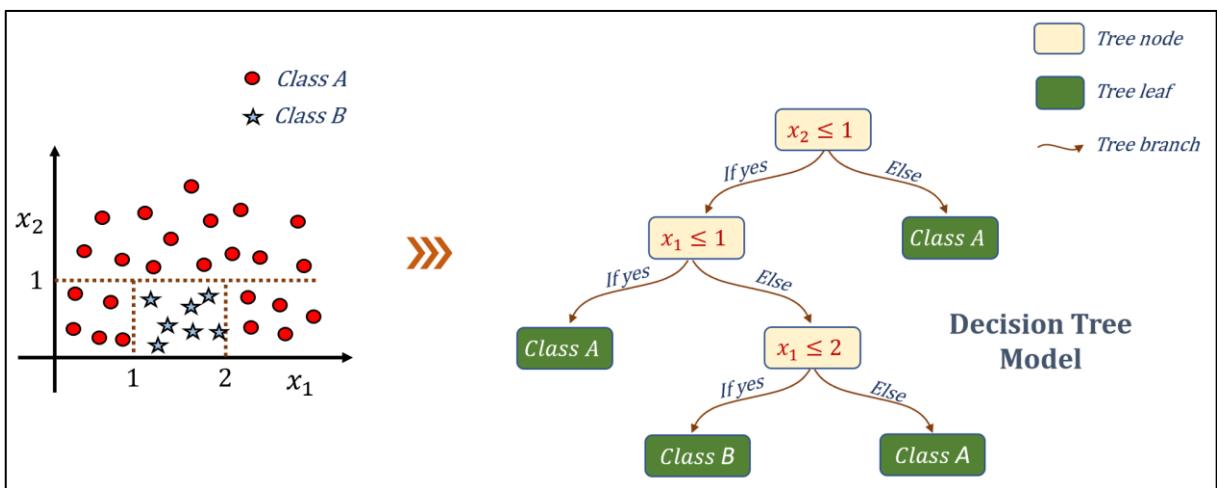


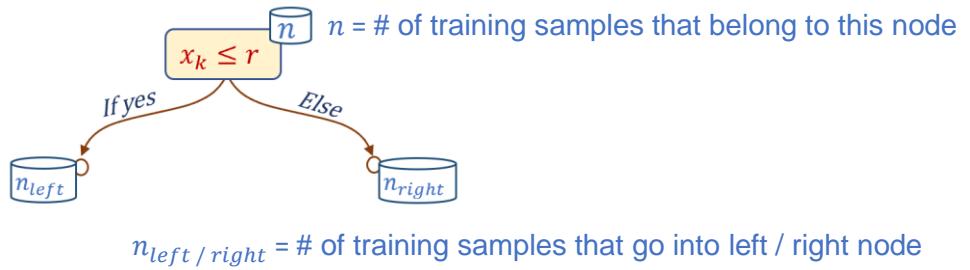
Figure 13.1: A decision tree with constant model used for binary classification in a 2D space

The trick in DT model fitting lies in deciding which questions to ask in the if-else statements at each node of the tree. During fitting, these questions split the feature space into smaller and smaller subregions such that the training observations falling in a subregion are similar to each-other. The splitting process stops when no further gains can be made or stopping criteria have been met. Improper choices of splits will generate a model that does not generalize well. In the next subsection, we will study a popular DT training algorithm called CART (classification and regression trees) which judiciously determines the splits.

Mathematical background

CART algorithm creates a binary tree, i.e., at each node two branches are created that split the dataset in such a way that overall data ‘impurity’ reduces. To understand this, consider

the following node⁸⁴ of a tree. Also assume that we are dealing with binary classification problem with input vector $x \in R^m$.



The algorithm needs to decide which one of the m input variables, x_k , will be used to split the set of n samples at this node and with what threshold r . CART makes this determination by minimizing the following objective

$$J(k, r) = \frac{n_{left}}{n} I_{left} + \frac{n_{right}}{n} I_{right}$$

where $I_{left/right}$ denote the data impurity of the left/right subsets of data and is given by

$$I = 1 - \sum_{q=1}^2 p_q^2 \quad \text{eq. 1}$$

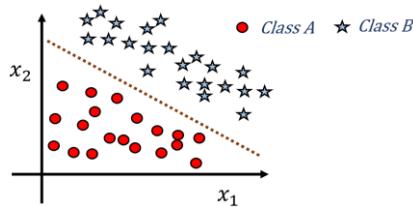
where p_q is the ratio of samples of the corresponding data subset belonging to class q . For example, if all the samples in a subset belong to class 1, then $p_1 = 1$ and $I = 0$. Therefore, if CART could find k and r such that the n samples get perfectly divided class-wise into the left and right subsets, then the minimum value of $J = 0$ will be obtained. However, this is usually not possible, and CART tries to do the best it can. The reduction in impurity (ΔI_{node}) achieved by CART at this node is given by $nI_{node} - \frac{n_{left}}{n} I_{left} - \frac{n_{right}}{n} I_{right}$.

CART simply follows the aforementioned branching scheme recursively. It starts from the top node (root node) and keeps splitting the subsets. If a node cannot be split any further (because impurity cannot be reduced anymore or some hyperparameter settings such as `min_samples_split`, `min_samples_leaf` prevent any further split), the node becomes a leaf or terminal node. For prediction, the leaf node corresponding to the test sample is found and the majority class from the leaf's training subset is assigned to the test sample. Note that a probabilistic prediction for class q can also be made by simply looking at the ratio of the leaf's training samples belonging to class q .

⁸⁴ There are algorithms like ID3 that create more than 2 branches at a node.



In classical DTs, as we have seen, the split decision rules (or split functions) at the nodes are based on a single variable. This results in axis-aligned hyperplanes that split the input space into several hyperrectangles. Complex split functions using multiple variables may also be used which may be more suitable for certain datasets – one such example is shown below



Fitting DTs with complex split functions, however, becomes computationally much more demanding. Therefore, a general recommendation is to transform your dataset using dimensionality reduction techniques like PCA, FDA before fitting tree using simple split functions.

For regression problems, a tree is built in the same way with a different objective function, $J(k, r)$, which now is given by

$$J(k, r) = \frac{n_{left}}{n} MSE_{left} + \frac{n_{right}}{n} MSE_{right}$$

Where $MSE_{left/right}$ denote the mean-squared error at the left/right node

$$MSE = \frac{\sum_{sample \in subset} (\hat{y} - y_{sample})^2}{\# \text{ of samples}}$$

\hat{y} is the average output of the samples belonging to a subset. Prediction for a test sample is also taken as the average output value of all training samples assigned to the test sample's leaf node.



You are not confined to using the constant predictive models at the leaves of a regression tree. Linear and polynomial predictive models may be more suitable for certain problems. Such DTs are called model trees. While Sklearn allows only the constant model, there exists a package⁸⁵ called 'linear-tree' that allows building model trees with linear models at the leaves.

⁸⁵ <https://github.com/cerlymarco/linear-tree>

Impurity metric

The impurity measure used in Eq. 1 is called *Gini impurity*. Another commonly employed measure is *entropy* and is given as follows for a dataset with 2 classes

$$I_H = - \sum_{q=1}^2 p_q \log(p_q)$$

I_H becomes 0 when $p_1=1$ (or $p_2=0$) and 1 when $p_1=p_2=0.5$. Therefore, reduction of entropy leads to more data purity. In practice, both Gini impurity and entropy provide similar results.

Simple Sklearn implementation

The code below illustrates implementation of a DT regression model to a quadratic dataset in Sklearn. Figure 13.2a plots the predicted vs actual output values. It is apparent that with no restriction on tree depth overfitting has occurred. Subplot 13.2b shows the regularizing impact of restricting the depth of the tree to only 3 levels.

```
# generate data
Import numpy as np
x = np.linspace(-1, 1, 50)[;, None]
y = x*x + 0.25 + np.random.normal(0, 0.15, (50,1))

# fit regularized DT model and predict
from sklearn import tree
model = tree.DecisionTreeRegressor(max_depth=3).fit(x, y)
y_pred = model.predict(x)
```

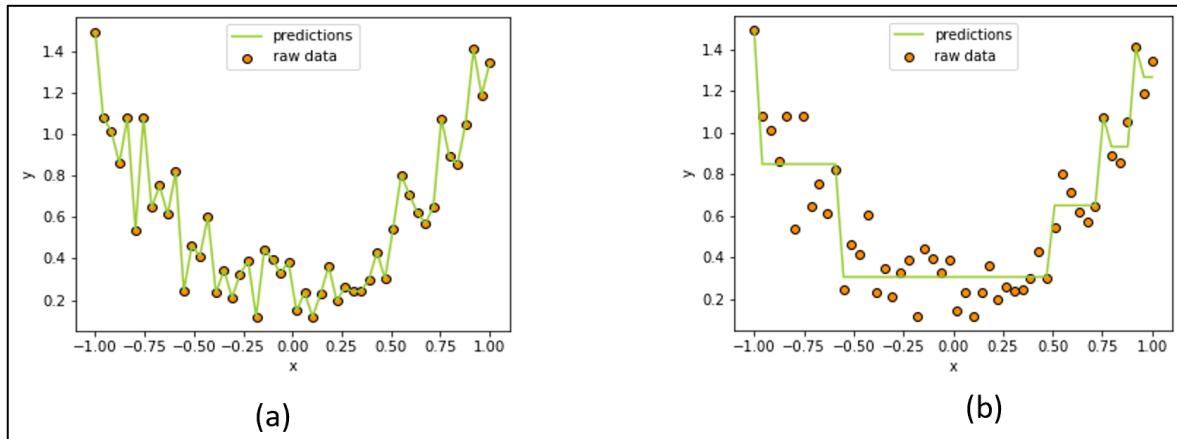
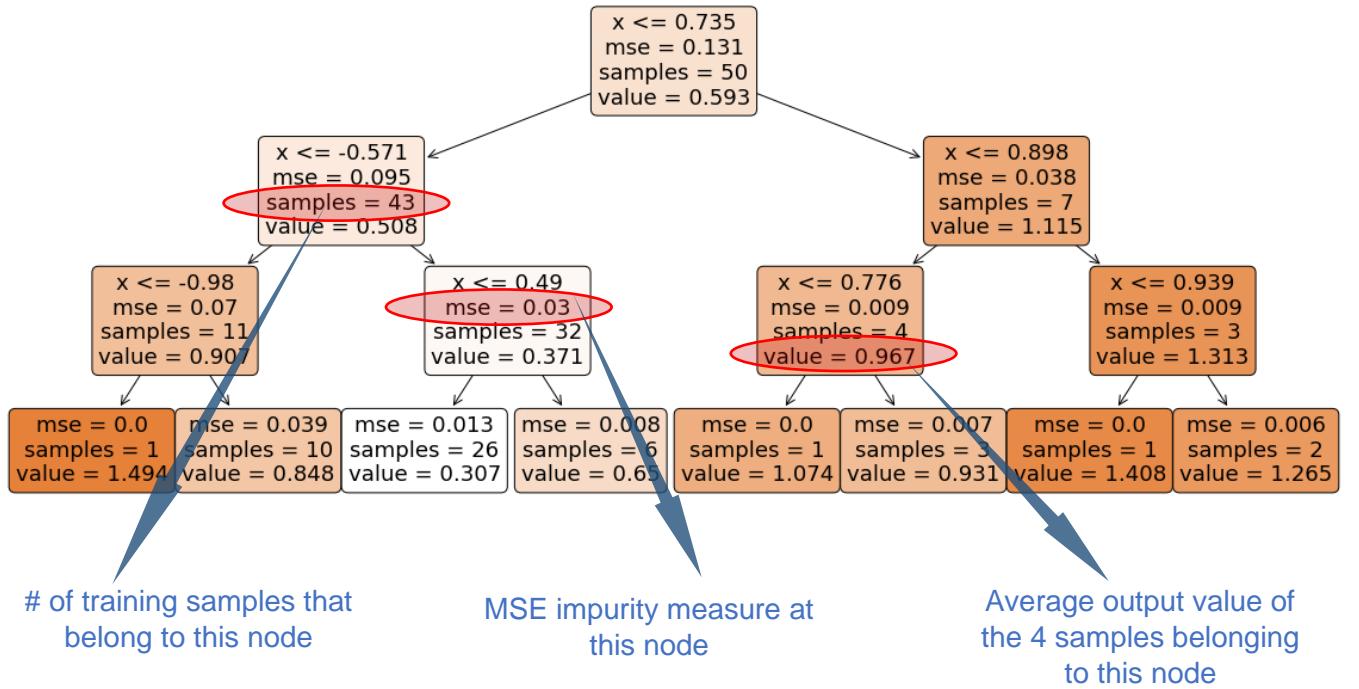


Figure 13.2: Decision tree regression predictions using unregularized and regularized models.

You can use the `plot_tree` function to plot the tree itself to understand how the training dataset has been partitioned by the DT model.

```
# plot tree
plt.figure(figsize=(20,8))
tree.plot_tree(model, feature_names=['x'], filled=True, rounded=True)
```



While DTs appear to be very flexible and useful modeling mechanism, they are seldom used as a standalone model. In Figure 13.2, we saw that DTs can easily overfit and give non-smooth or piece-wise constant approximations. Another disadvantage with DTs is instability, i.e., small variations in training dataset can result in a very different tree. However, there is a reason why we invested time in learning DTs. A single tree may not be useful, but when you combine multiple trees, you get amazing results. We will learn how this is made possible in the next section.

13.2 Random Forests: An Introduction

Random forest (RF) is a supervised nonlinear regression and classification modeling technique that is made up of several decision trees. These DTs are fit independently to their training datasets and RF's predictions are made by averaging the predictions of the trees. Figure 13.3 illustrates this scheme. Combining outputs from trees results in low variance or more robust model. The trick lies in how these constituent trees are generated. The trees are generated in such a way that their prediction errors are uncorrelated, resulting in RF being less prone to overfitting (we will concretize this concept mathematically soon).

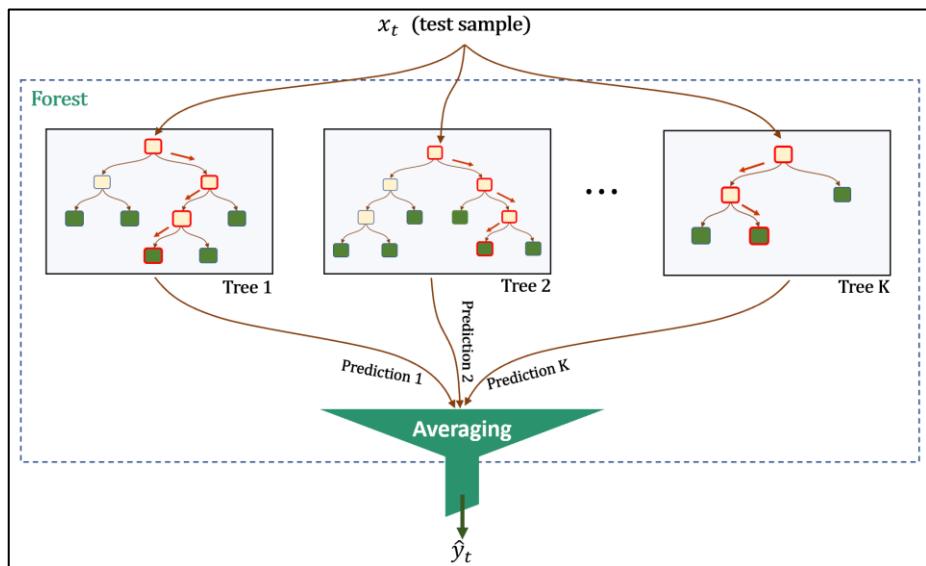


Figure 13.3: A random forest prediction is combination of predictions from multiple decision trees. [For a classification problem, Sklearn returns the class corresponding to the highest average class probability]

In random forest, the trees are grown to full extent, and therefore, hyperparameter selection for the trees is not a concern. This makes RF's training and execution simple and quick. Infact RFs have very small number of tunable hyperparameters. RFs also lend themselves useful for computation of variable importances. All these qualities have led to the popularity of random forests.

Mathematical background

For training random forests, different trees need to be generated that are as 'distinct from each-other' as possible but at the same time provide good descriptions of the training dataset. This variety among the trees are achieved via the following two means:

- 1) Using different training datasets for each tree: If each tree is trained on the same dataset, they will end up being identical, make the same kind of errors, and therefore combining them will offer little benefit. Since we only have a single training dataset to train the RF, bootstrapping is employed. If original dataset has N samples, bootstrapping allows creation of multiple datasets, each with N_b ($\leq N$) samples, such that each new dataset is also a good representative of the underlying process that generated the original dataset. Each bootstrap dataset is generated by randomly selecting N_b samples with replacement from the original dataset. In RF, $N_b = N$ and the bootstrapping scheme is illustrated below for $N=10$.

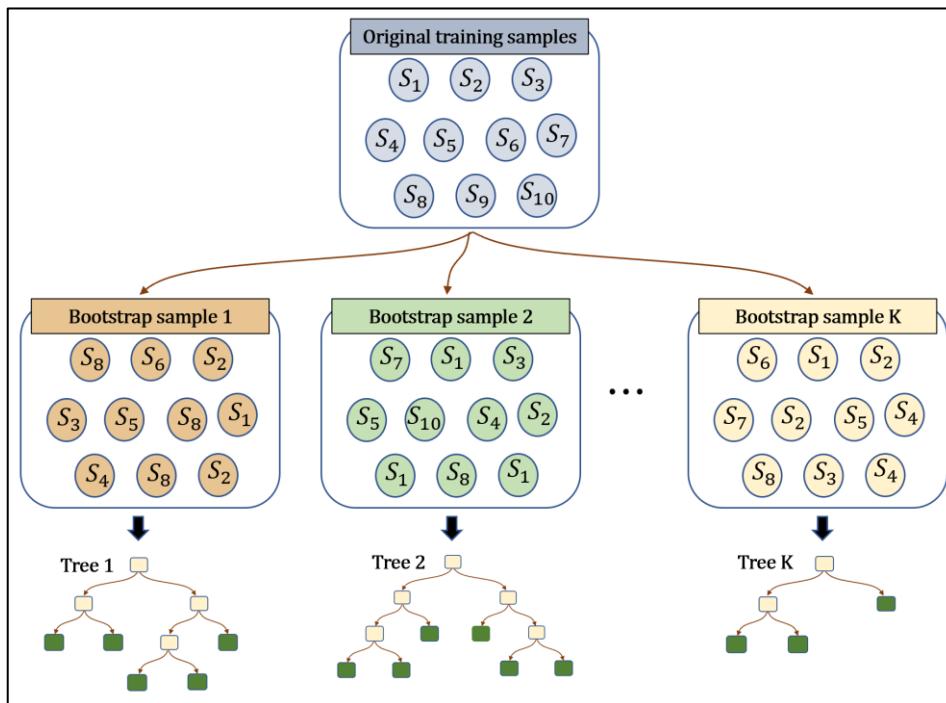


Figure 13.4 Creation of separate DT models using bootstrap samples. S_i denotes the i^{th} training sample.

- 2) Using random subsets of input variables to find the optimal split: A very non-intuitive, somewhat surprising, but incredibly effective aspect of RF training is that not all the input variables are considered for determining the optimal split function at any node of any tree in the forest. Instead, a random subset of variables is chosen and then the node impurity is minimized with these chosen variables. This random selection is performed at every node. If input variable $x \in R^m$, then the number of random split variables (M) is recommended to be the floored squared root of m .

The above two tricks during training result in trees being minimally correlated to each-other. Figure below summarizes the RF model fitting procedure. For illustration, it is assumed that $x \in R^9$ and $M=3$.

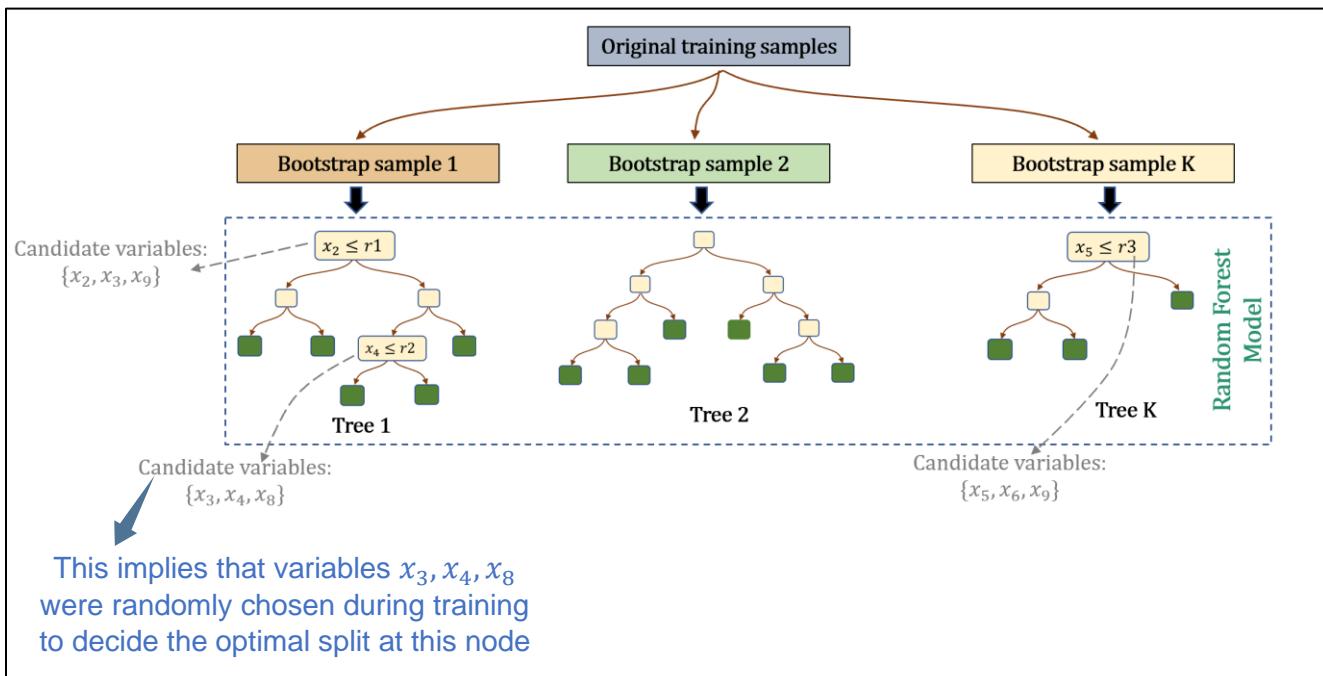


Figure 13.5: Illustration of RF training mechanism

You can see that there are only two main hyperparameters to decide: number of trees and the size of variable subset. Another advantage with RF is that the constituent trees can be trained in parallel.

Simple Sklearn implementation

Let's fit a RF model to the quadratic dataset from Figure 13.2. Figure 13.6 shows the fit with 20 trees in the forest. It is apparent that RF provides much smoother predictions compared to those from a DT model as we saw before. Predictions from two different constituent trees of the forest are also shown. We can see that RF smoothes out the wiggles in constituent tree predictions and makes the combined model more stable.

```
# fit RF model and predict
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(n_estimators=20).fit(x, y)
y_pred = model.predict(x)

# get predictions from constituent trees and plot
tree_list = model.estimators_ # list of DT models in the RF
y_pred_tree1 = tree_list[5].predict(x)
y_pred_tree2 = tree_list[15].predict(x)
```

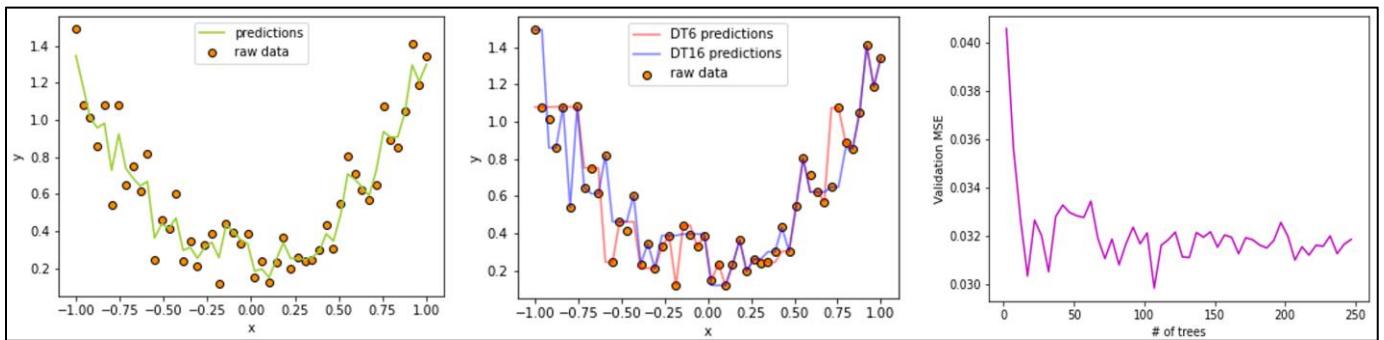


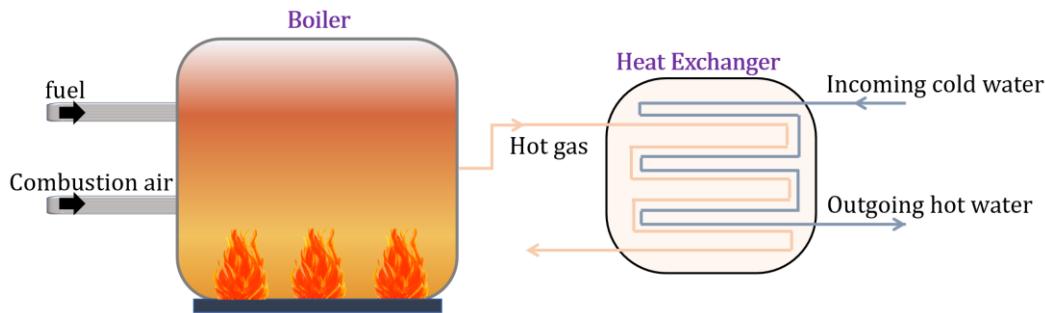
Figure 13.6: (Left) Random forest regression predictions, (middle) predictions from a couple of constituent decision trees, (right) Impact of number of trees in RF model on validation error

Figure 13.6 also shows the validation error for different number of trees in the forest. What we see here is a general characteristic of RFs: as number of trees increase, the validation errors plateau out. Therefore, if computation time is not a concern, it is preferable to use as many trees as possible until the error levels out. As far as M is concerned, a lower value leads to greater reduction in variance but increases the model bias. Do note that RF primarily brings down the variance (compared to using single DT models) and not the bias. Therefore, if DT itself is underfitting, then RF won't be able to provide high accuracy (or low bias). This is why full-grown trees are used in RF. You will find out the reason behind this later in this chapter.

13.3 Fault Classification Using Random Forests: Gas Boiler Case Study

To illustrate the usage of random forests for fault classification, we will use simulated dataset⁸⁶ from a gas boiler system presented in the publicly available paper titled '*Machine learning algorithms for classification of boiler faults using a simulated dataset*'. The representative system is shown below. Fuel combusts in combustion chamber and the resulting hot gas heats up the water stream. Three types of faults, viz, usage of excessive air, gas-side fouling of the heat exchanger, and water-side scaling, have been simulated. A total of 27280 simulation have been provided that includes NOC samples for a range of operational parameters (gas fuel rate from 1 kg/s to 4 kg/s, water mass flow rate from 3 kg/s to 12.5 kg/s, and combustion air temperature from 283 K to 303 K). The original paper reports random forests among the top performing models. Let's see if we can replicate the reported performance.

⁸⁶ Shohet et al., Simulated boiler data for fault detection and classification. Available at <https://ieee-dataport.org/open-access/simulated-boiler-data-fault-detection-and-classification>, IEEE Dataport, 2019. Data shared under Creative Commons Attribution license (<https://creativecommons.org/licenses/by/4.0/>).



```
# import packages
import numpy as np, pandas as pd, matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier

# read data
data = pd.read_csv('Boiler_emulator_dataset.txt', delimiter=',')

data.drop(data[data.Class == 'Lean'].index, inplace=True) # remove rows where Class == Lean
input_data = data.iloc[:,[0,1,3,4]].values # dropping column Treturn which has constant values
output_label_text = data.iloc[:, -1]

# convert text labels to numeric labels
le = LabelEncoder().fit(output_label_text)
output_labels = le.transform(output_label_text)
print(le.classes_)

>>> ['ExcessAir' 'Fouling' 'Nominal' 'Scaling']

# separate training and test data
X_train, X_test, y_train, y_test = train_test_split(input_data, output_labels, test_size=0.3,
                                                    stratify=output_labels, random_state=1)

# scale data
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# fit random forest model
clf = RandomForestClassifier()
clf.fit(X_train_scaled, y_train)
```

```

y_train_pred = clf.predict(X_train_scaled)
y_test_pred = clf.predict(X_test_scaled)

# generate and plot confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sn

conf_mat = confusion_matrix(y_test, y_test_pred)

plt.figure(figsize=(12,8)), sn.set(font_scale=2)
sn.heatmap(conf_mat, fmt='0f', annot=True, cmap='Blues', xticklabels=le.classes_,
           yticklabels=le.classes_)

plt.ylabel('True Fault Class', fontsize=30, color='maroon')
plt.xlabel('Predicted Fault Class', fontsize=30, color='green')

```



The confusion matrix shows that random forest classifier does a pretty good job (that too with default values of hyperparameters) at correctly identifying the sample's classes. This concludes our study of RFs. Hopefully, we have been able to convince you that RFs could be quite a powerful weapon in your ML arsenal. Let's proceed to learn about ensemble learning to understand what is it that imparts power to RFs.

13.4 Introduction to Ensemble Learning

The idea of combining multiple ‘not so good’/weak models to generate a strong model is not restricted to random forests. The idea is more generic and is called ensemble learning. Specific methods that implement the idea (like RFs) are called ensemble methods. Figure 13.7 below shows an ensemble modeling scheme employing a diverse set of base models. The shown model is heterogeneous ensemble model as individual models employ different learning methodologies. In contrast, in the homogeneous models, the base models use the same learning algorithm.

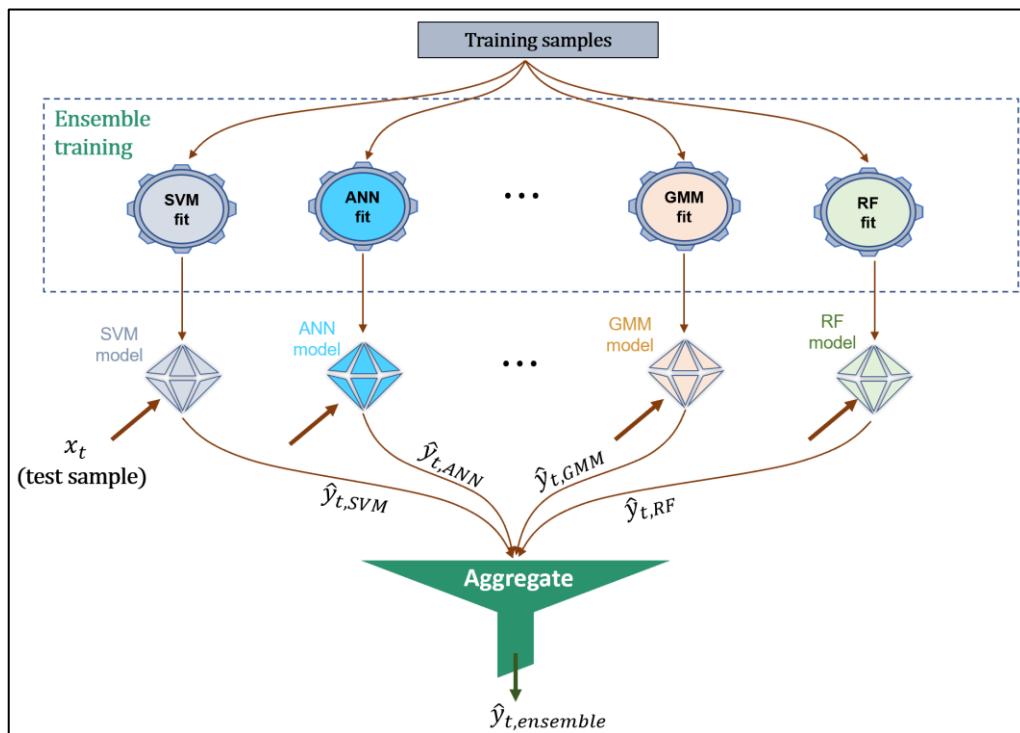


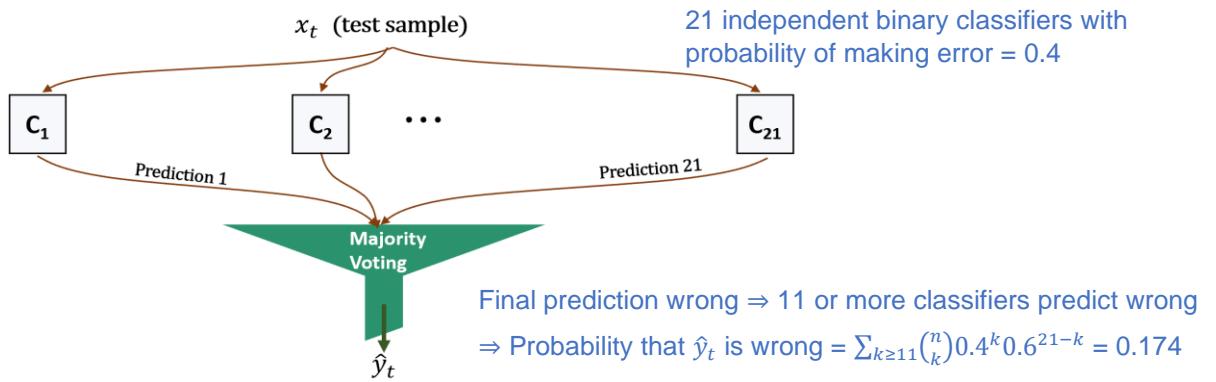
Figure 13.7: Heterogeneous ensemble learning scheme⁸⁷. RF, which itself is an ensemble method, is used as a base model here.

There are several options at our disposal to aggregate the predictions from the base models. For classification, we can use majority voting and pick the class that is predicted by most of the base models. Alternatively, if base models return class probabilities, then soft voting can also be used. For regression, we can use simple or weighted averaging.

In the ensemble world, a weak model is any model that has poor performance due to either high bias or high variance. However, together, the weak models combine to achieve better,

⁸⁷ Another popular heterogeneous ensemble technique is stacking where base models’ predictions serve as inputs to another meta-model which is trained separately

more accurate (lower bias), and/or robust (lower variance) predictions. Can we then combine any set of poor performing models and obtain a super accurate ensemble model? Both yes and no! There are certain criteria that must be met to be able to reap the benefits of ensemble learning. First, the base models must individually perform better than random guessing. Second, the base models must be diverse (i.e., the errors made on unseen data must be uncorrelated). Consider the following simple illustration to understand these requirements.



In above illustration, the ensemble model will have an accuracy of 82.6% although each base model is only 60% accurate. While it is relatively easy to fulfill this first criterion of building base models with > 50% accuracies, it is tough to obtain diversification/independence among the base models. In Figure 13.7, if all the base models make identical mistakes all the time, combining them will offer no benefit. We already saw some diversification mechanisms in RF. There are other ways as well and the schematic below provides an overview of some popular ensemble diversification techniques.

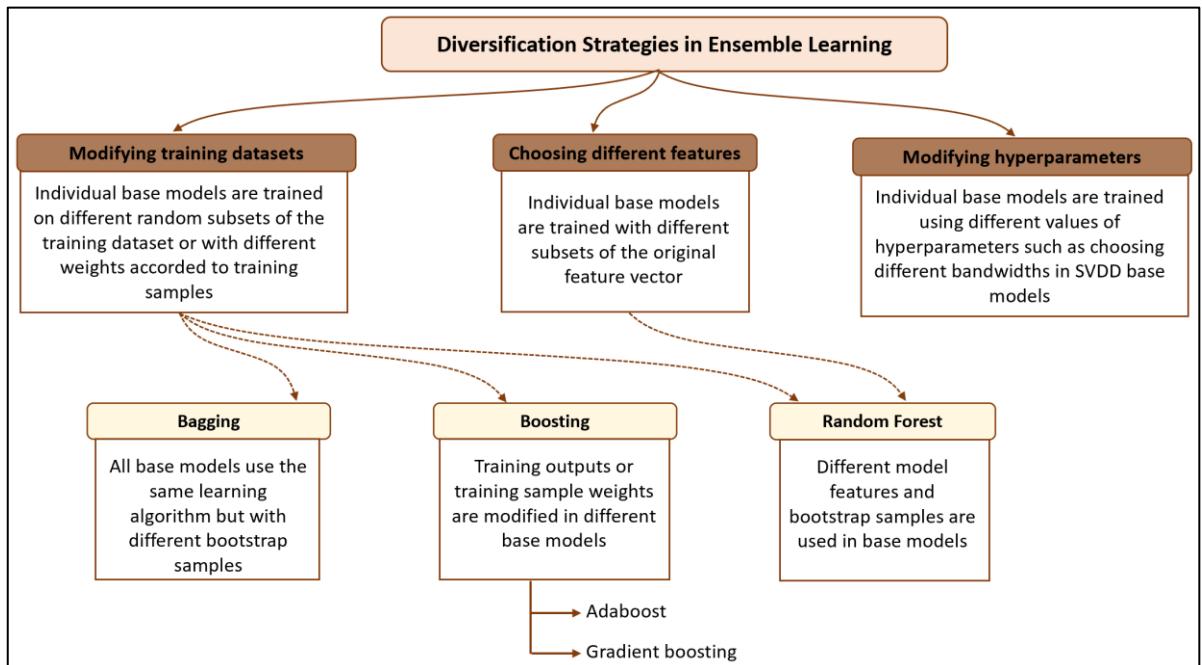


Figure 13.8: An overview of ensemble modeling techniques with homogeneous base models



The choice of ensemble technique should be based on the base model's characteristics. If the problem with the base model is its high variance, then choose the ensemble method that tends to reduce variance (such as bagging). On the other hand, if the problem is high bias, then use methods like boosting and stacking that will produce an ensemble model with reduced bias.

Let's now become more familiar with these techniques.

Bagging

In bagging (bootstrap aggregating) technique, the diverse set of base models are generated by employing the same modeling algorithm with different bootstrap samples of the original training dataset. Unlike RF, the input variables are kept the same in each base model and the size of bootstrap dataset is usually kept less than the original number of training samples. Moreover, the base model is not restricted to be a DT model. As the figure below shows, the base models are trained independently (allowing parallel trainings) and ensemble prediction is obtained by combining base models' predictions.

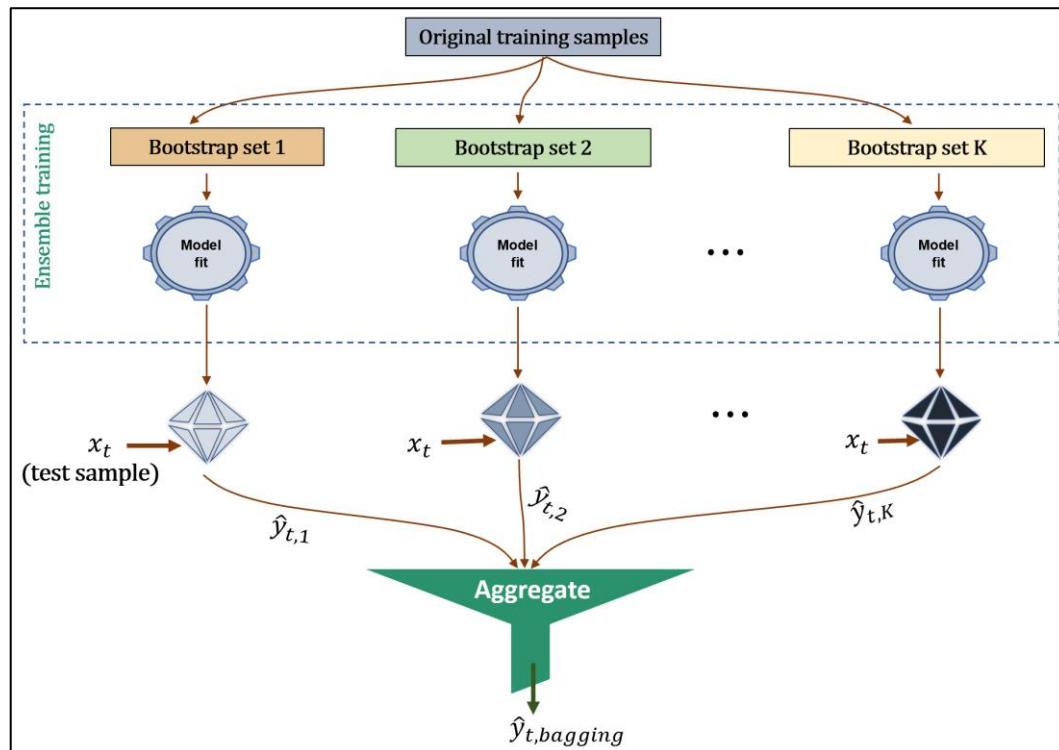


Figure 13.9: Bagging ensemble technique. Note that if you have multiple independently sampled original training datasets, then bootstrapping is not needed.

The defining characteristic of bagging is that it helps in reducing variance but not bias. It can be shown that if $\bar{\rho}$ is the average correlation among base model predictions then variance of y_{ensemble} equals $\bar{\sigma}^2 \left(\bar{\rho} + \frac{1-\bar{\rho}}{K} \right)$ where $\bar{\sigma}^2$ is variance of an individual base model's predictions and K is number of base models. Therefore, $\bar{\rho} = 1$ implies no reduction in variance.

Bagging can be used for both regression and classification. Sklearn provides BaggingClassifier and BaggingRegressor for them, respectively. A simple illustration below shows how bagging can help achieve smoother results (classification boundaries in this case).

```
# generate training samples
from sklearn import datasets
noisy_moons = datasets.make_moons(n_samples=200, noise=0.3, random_state=10)
X, y = noisy_moons

# fit bagging model (Sklearn uses decision trees as base models by default)
from sklearn.ensemble import BaggingClassifier
Bagging_model = BaggingClassifier(n_estimators=500, max_samples=50, random_state=100).fit(X,
y) # K=500 and each DT is trained on 50 training samples randomly drawn with replacement
```

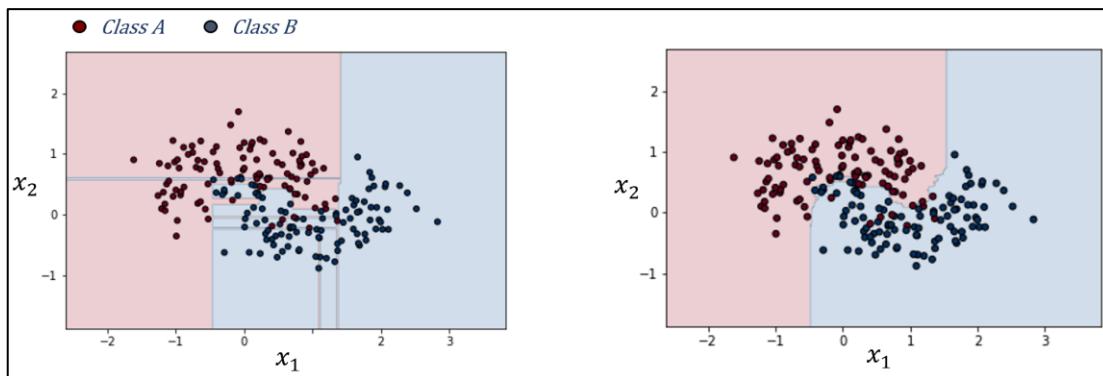


Figure 13.10: Classification boundaries obtained using (left) fully grown decision tree and (right) bagging with fully grown decision trees

Boosting

In boosting ensemble technique, the base models are again obtained using the same learning algorithm but are fitted sequentially and not independently from each other. During training, each base model tries to 'correct' the errors made by the boosted model at the previous step. At the end of the process, we obtain an ensemble model that shows lower bias⁸⁸ than the base models.

⁸⁸ Variance may or may not decrease. Boosting has been seen to cause overfitting sometimes.

Boosting is preferred if base model exhibit underfitting/high bias. Like bagging, boosting is not restricted to DTs as base models, but if DTs are used, shallow trees (trees with only a few depths) are recommended that don't exhibit high variance. Moreover, shallow trees or any other less complex base model are computationally tractable to train sequentially during boosting process. Boosting can be used for both regression and classification, and there are primarily two popular boosting methods, namely, Adaboost and Gradient Boosting.

Adaboost (Adaptive Boosting)

In Adaboost boosting technique, a new base model corrects the errors of its predecessor base model by focusing on the samples that were wrongly predicted. As shown in Figure 13.11, all samples are assigned equal weights of $\frac{1}{N}$ and the first (weak) base model is trained. Weights of wrongly predicted samples are increased, and weights of correctly predicted samples are decreased. Second base model is trained, and the cycle continues until the specified number of base models are obtained. In essence, each successive base model focusses more on the difficult (to predict) training samples. Post training, the ensemble model's prediction is taken as the weighted combination of base models' predictions. Model weight assigned to each base model depends on the accuracy achieved on its weighted training samples (higher accuracy \Rightarrow higher α).

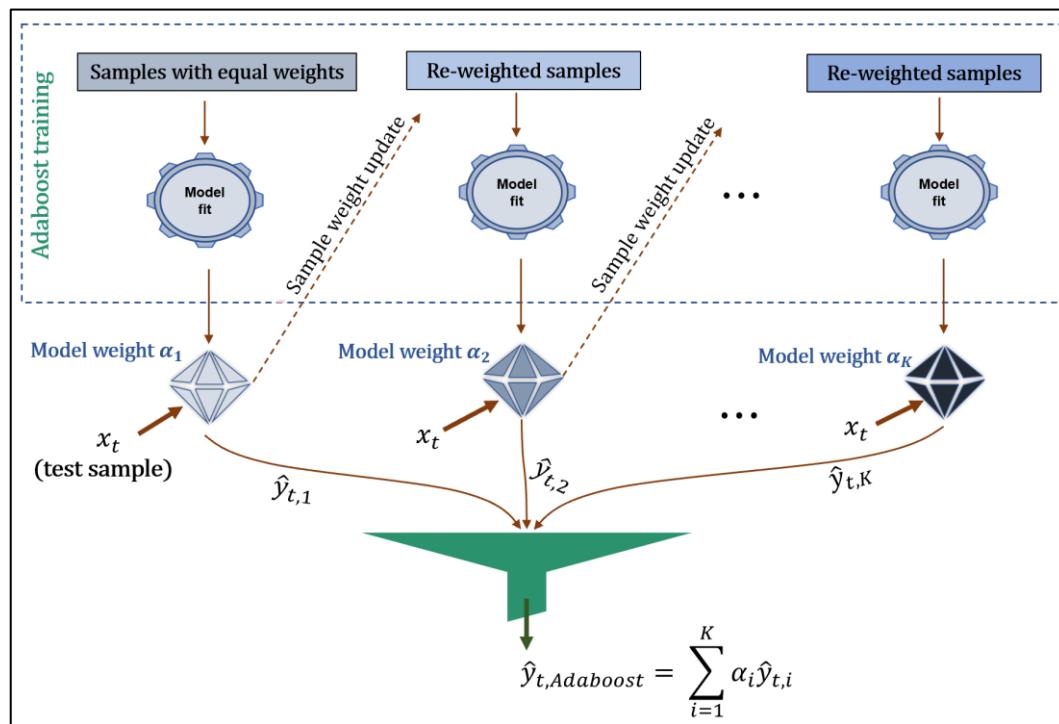


Figure 13.11: Adaboost ensemble scheme

Gradient Boosting

Adopting an approach different from Adaboost, Gradient boosting approach corrects the errors of its predecessor by sequentially training a base model using the residual error made by the previous base model. Figure 13.12 shows the algorithm's scheme.

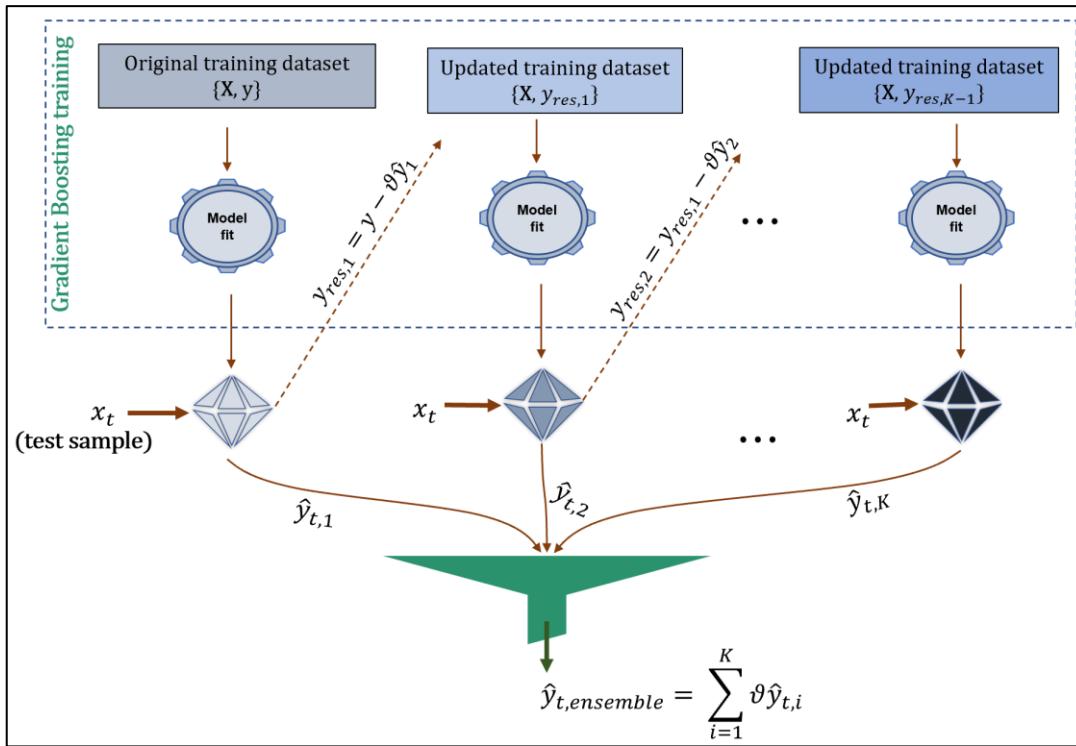


Figure 13.12: Gradient Boosting ensemble scheme. Here, \hat{y}_i denote prediction for training samples from the i^{th} base model. Note that the hyperparameter ϑ need not be constant (as used here) for the different base models.

In the scheme above, the hyperparameter $\vartheta \in (0,1]$ is called shrinkage parameter or learning rate. It is used to prevent overfitting and is recommended to be assigned a small value like 0.1. Very small learning rate will necessitate usage of large number of base models. The number of iterations or base models, K , is another important hyperparameter that needs to be tuned carefully. Too small K will not achieve sufficient bias reduction while too large K will cause overfitting. It can be optimized using cross-validation or early stopping (keeping track of validation error at different stages/iterations of ensemble model training and stopping when errors stop decreasing or model residuals do not have any more pattern that can be modeled).

Sklearn provides `GradientBoostingClassifier` and `GradientBoostingRegressor` classes for regression and classification purposes, respectively. While these classes use DTs as base models/weak learners, gradient boosting methodology is not limited to DTs. When DTs are used, it is called Gradient Tree Boosting or Gradient Boosted Decision Trees (GBDTs).

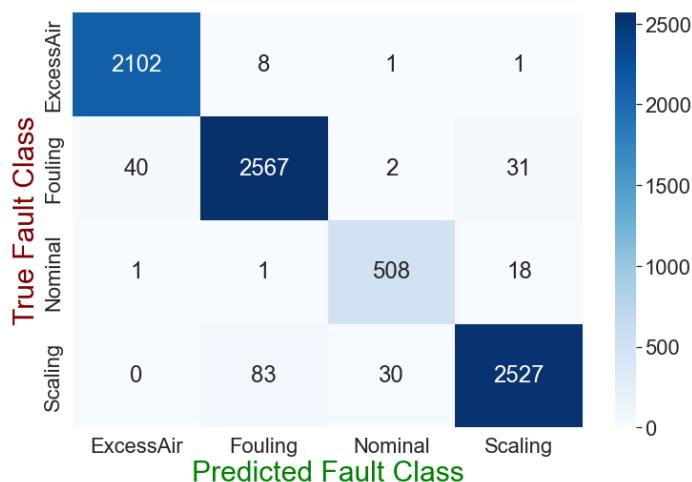
13.5 Fault Classification Using XGBoost: Gas Boiler Case Study

If you have been active in the ML world in the recent times, then you must have heard the term XGBoost. It stands for *eXtreme Gradient Boosting* and is a popular library that implements several tricks and heuristics to make gradient boosting-based model training very effective, especially for large datasets. XGBoost uses DTs as base models. Let's apply XGBoost for the gas-boiler fault classification problem and see if we can better RF model's performance.

```
# fit xgboost model
import xgboost as xgb
clf = xgb.XGBClassifier(use_label_encoder=False)
clf.fit(X_train_scaled, y_train)
y_train_pred = clf.predict(X_train_scaled)
y_test_pred = clf.predict(X_test_scaled)

# generate and plot confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sn
conf_mat = confusion_matrix(y_test, y_test_pred)

plt.figure(figsize=(12,8)), sn.set(font_scale=2)
sn.heatmap(conf_mat, fmt='0f', annot=True, cmap='Blues', xticklabels=le.classes_,
           yticklabels=le.classes_)
plt.ylabel('True Fault Class', fontsize=30, color='maroon')
plt.xlabel('Predicted Fault Class', fontsize=30, color='green')
```



A quick cursory glance of the confusion matrix suggests similar performance as that from RF model. Perhaps, a systematic hyperparameter optimization may provide even better performance. Our aim was to demonstrate the ease with which reasonably good classifier models can be built with default settings of XGBoost.

This chapter was a whirlwind tour of ensemble learning which is an important pillar of modern machine learning. Each topic introduced in the chapter has several more advanced aspects that we could not cover here. However, you now have the requisite fundamentals and familiarization in place to explore these advanced aspects.

Summary

In this chapter, we learnt the decision tree modeling methodology. We saw how random forests can help overcome the high variance issues with trees. Then, we studied the broader concept of ensemble learning which can be used to overcome the high bias and/or high variance issues with weak models. Two popular ensemble methods, viz, bagging and boosting, were conceptually introduced. Finally, we used the gas boiler dataset to illustrate the ease with which fault classification models can be built using random forests and XGBoost (a popular implementation of gradient boosting trees). This chapter has added several powerful techniques to your ML arsenal and you will find yourself using these quite often.

Chapter 14

Proximity-based Techniques for Fault Detection

Most of the anomaly detection techniques that we have studied so far work by finding some structure in training dataset, such as the low-dimensional manifold in PCA, NOC boundary in SVDD, optimal separating hyperplane in SVM, etc. However, another popular class of methods exists that utilizes a very straightforward and natural notion of anomalies as data points that are far away or isolated from the NOC data samples; logically, these methods are classified as proximity-based methods.

Proximity of a data point can simply be defined as its distance (as done in *k*-NN method) from its neighbors. An abnormal data point lies far away from other NOC data and therefore, its nearest neighbors' distances will be large compared to those for NOC samples. Another related but slightly different notion of proximity is the density or number of other data points in a local region around a test sample. Local outlier factor (LOF) is a popular method in this category wherein samples not lying in dense region are classified as anomalies. The third technique, isolation forest (IF), that we will study in this chapter uses the similar notion that anomalies are 'far and between'. Here, the data space is split until each data point gets 'isolated'. Anomalies can be isolated easily and require very few splits compared to NOC samples that lie close to each other.

You may have realized that these techniques generate interpretable results and are easy to understand. Correspondingly, they come in pretty handy to analyze complex system whose characteristics may not be well-known *a priori*. Let's now get down to business. We will cover the following topics

- Introduction to k-NN technique
- Introduction to LOF technique
- Introduction to isolation forest technique
- Applications of k-NN, LOF, and IF for fault detection in semiconductor manufacturing process

14.1 KNN: An Introduction

The k-nearest neighbors (k-NN or KNN) algorithm is a versatile technique based on a simple intuitive idea that the label/value for a new sample can be obtained from the labels/values of closest neighboring samples (in the feature space) from the training dataset. The parameter k denotes the number of neighboring samples utilized by the algorithm. As shown in Figure 14.1, k-NN can be used for both classification and regression. For classification, k-NN assigns test sample to the class that appears the most amongst the k neighbors. For regression, the predicted output is the average of the value of the k neighbors. Due to its simplicity, k-NN is widely used for pattern classification and was included in the list of top 10 algorithms in data mining.⁸⁹

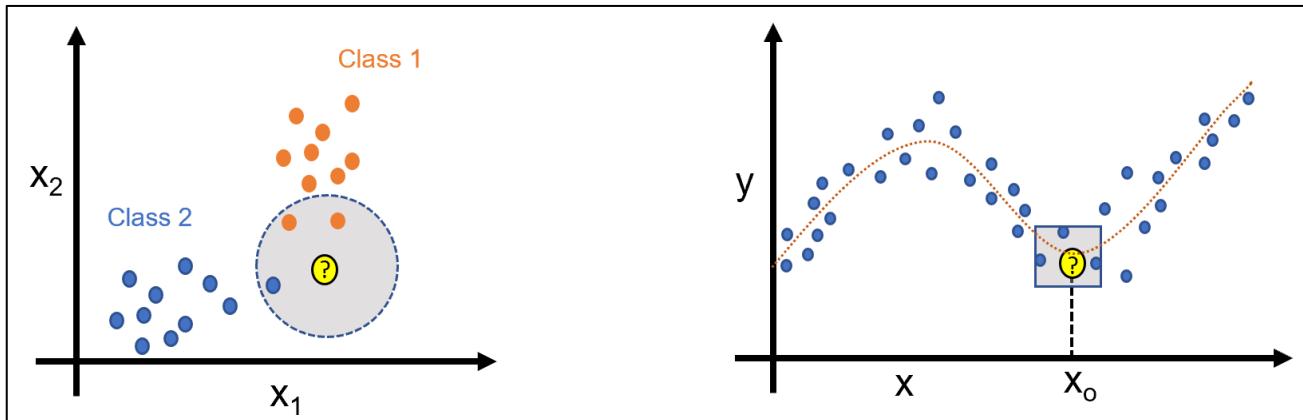


Figure 14.1: k-NN illustration for classification (left) and regression (right). Yellow data point denotes unknown test sample. The grey-shaded region represents the neighborhood with 3 nearest samples.



k-NN belongs to the class of lazy learners where models are not built explicitly until test samples are received. At the other end of the spectrum, eager learners (like, SVM, decision trees, ANN) ‘learn’ explicit models from training samples. Unsurprisingly, training is slower, and testing is faster for eager learners. KNN requires computing the distance of the test sample from all the training samples, therefore, k-NN also falls under the classification of instance-based learning. Instance-based learners make predictions by comparing the test sample with training instances stored in memory. On the other hand, model-based learners do not need to store the training instances for making predictions.

⁸⁹ Wu et al., Top 10 algorithms in data mining. Knowledge and Information systems, 2008.

Conceptual background

Apart from an integer k and input-output training pairs, k-NN algorithm needs a distance metric to quantify the closeness of a test sample with the training samples. The standard Euclidean metric is commonly employed. Once the nearest neighbors have been determined, two approaches, namely uniform and distance-based, can be employed to decide weights assigned to each neighbor which impacts the neighbor's contribution in prediction. In uniform weighting, all k neighbors are treated equally while, in distance-based weighting, each of the k neighbors is weighted by the inverse of their distance from the test sample so that closer neighbors will have greater contributions. The figure below illustrates the difference between the two weight schemes for a classification problem

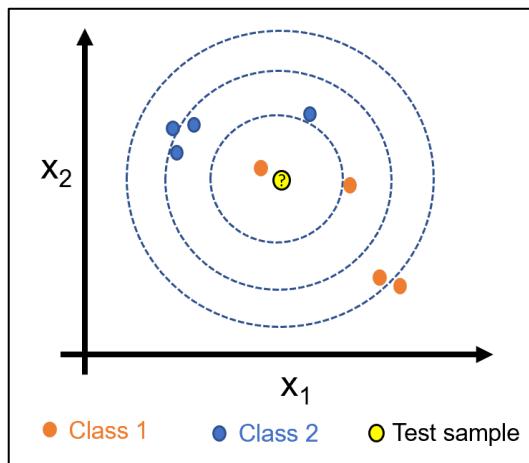


Figure 14.2: Illustration on impact of weight-scheme on k-NN output. Dashed circles are for distance references.

In Figure 14.2, with uniform weighting (also called majority voting for classification problems), the test sample is assigned to class 1 for $k = 1$ or 3 and class 2 for $k = 6$. For $k = 8$, no decision can be made. With distance-weighting, test sample is always classified as class 1 for $k = 1, 3, 6$, or 8. This illustration shows that distance weighting can help reduce the prediction dependence on the choice of k .

For predictions, k-NN needs to compute the distance of test samples from all the training samples. For large training sets, this computation can become expensive. However, specialized techniques, such as KDTree and BallTree, have been developed to speed up the extraction of neighboring points without impacting prediction accuracies. These techniques utilize the structure in data to avoid computing distances from all training samples. The `NearestNeighbors` implementation in scikit-learn automatically selects the algorithm best suited to the problem at hand. The `KNeighborsRegressor` and `KNeighborsClassifier` modules are provided by Scikit-learn for regression and classification, respectively.

A couple of things to pay careful attention in k-NN include variable selection and variable scaling. Variables that are not important for output predictions should be removed; otherwise unimportant variables will undesirably impact the determination of nearest neighbors. Further, the selected variables should be properly scaled to ensure that variables with large magnitudes do not dwarf the contribution of other variables during distance computations.

Deciding k-NN hyperparameters

The choice of the hyperparameter, k , is critical and is the knob to balance underfitting and overfitting. Small k can lead to overfitting where noisy measurements unfavorably impact the predictions. Large k can reduce the effect of noise but can lead to underfitting as it over-smooths the predictions for regression or make classification boundaries less distinct. Unfortunately, there is no standard guidance on the choice of k . The optimal value depends on the data and can be estimated via cross-validation.

Applications of k-NN for process systems

As alluded to before, k-NN method is often employed for equipment condition monitoring⁹⁰. For process-level monitoring, k-NN classification can be used to classify process abnormalities into distinct fault classes if sufficient historical faulty samples are available. As opposed to k-NN fault classification, an interesting adaptation of k-NN for fault detection was proposed by He & Wang that made use of only normal operation data⁹¹. We will study this application in more detail in the next section.

A few other notable applications of k-NN for process systems include the work of Facco et al. on automatic maintenance of soft sensors⁹², Borghesan et al. on forecasting of process disturbances⁹³, Cecilio et al. on detecting transient process disturbances⁹⁴, and Zhou et al. on fault identification in industrial processes⁹⁵. These applications may not utilize the k-NN method directly for classification or regression but use the underlying concept of similarity of nearest neighbors.

⁹⁰ Dong Wang, K-nearest neighbors-based methods for identification of different gear crack levels under different motor speeds and loads: Revisited, *Mechanical Systems and Signal Processing*, 2016

⁹¹ He and Wang, Fault detection using k-nearest neighbor rule for semiconductor manufacturing processes, *IEEE Transactions on Semiconductor Manufacturing*, 2007.

⁹² Facco et al., Nearest-neighbor method for the automatic maintenance of multivariate statistical soft sensors in batch processing, *Industrial Engineering & Chemistry Research*, 2010.

⁹³ Borghesan et al., Forecasting of process disturbances using k-nearest neighbors, with an application in process control, *Computers and Chemical Engineering*, 2019.

⁹⁴ Cecilio et al., Nearest neighbors methods for detecting transient disturbances in process and electromechanical systems, *Journal of Process Control*, 2014.

⁹⁵ Zhou et al., Fault identification using fast k-nearest neighbor reconstruction, *Processes*, 2019

Application of k-NN for fault detection

Fault detection by k-NN²⁸ (FD-KNN) is based on a simple idea that distance of a faulty test sample from the nearest training samples (obtained from normal operating plant conditions) must be greater than a normal sample's distance from the neighboring training samples. Incorporating this idea into the process monitoring framework, a monitoring metric (termed k-NN squared distance) is defined for each training sample as follows

$$D_i^2 = \sum_{j=1}^k d_{ij}^2$$

where d_{ij}^2 is the distance of i^{th} sample from its j^{th} nearest neighbor. After computing k-NN squared distances for all the training samples, a threshold corresponding to the desired confidence limit can be computed. A test sample would be considered faulty if its k-NN squared distance is greater than the threshold.

An advantage of FD-KNN is that it is applicable to process systems with complex characteristics, such as multimodality, non-Gaussianity, and non-linearity. To illustrate a complete application of FD-KNN, we will use the metal-etch dataset that we used previously in Chapter 11 for GMM-based process monitoring. If you recall, the batch data was unfolded before scaling and PCA application. We will do the same pre-processing transformations here using Sklearn's Pipeline feature⁹⁶. Following He & Wang's work, k was set to 5.

```
# scale data & fit PCA model via pipeline
from sklearn.pipeline import Pipeline

pipe = Pipeline([('scaler', StandardScaler()), ('pca', PCA(n_components = 3))])
score_train = pipe.fit_transform(unfolded_dataMatrix)

# k-nearest neighbors of each training sample in score space
from sklearn.neighbors import NearestNeighbors

nbrs = NearestNeighbors(n_neighbors=6).fit(score_train) # a data-point is its own neighbor
d2_nbrs, indices = nbrs.kneighbors(score_train)
d2_sqrd_nbrs = d2_nbrs**2
D2 = np.sum(d2_sqrd_nbrs, axis = 1)
D2_log = np.log(D2) # logarithm used just for scaling purposes
```

⁹⁶ Note that we are using PC scores as model inputs rather than original variables. This is primarily for visualization convenience.

```
D2_log_CL = np.percentile(D2_log,95)
```

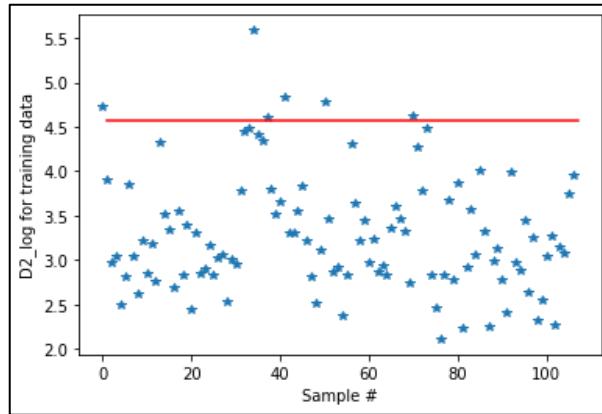


Figure 14.3 FD-KNN monitoring chart for metal-etch training samples

Figure 14.3 shows the resulting monitoring chart for the training samples. Figure 14.4 shows the monitoring chart for the faulty test samples. 15 out of 20 samples are correctly flagged as faulty while 5 samples are misdiagnosed as normal.

```
# scale and PCA on faulty test data
score_test = pipe.transform(unfolded_TestdataMatrix)

# D2_log_test
d2_nbrs_test, indices = nbrs.kneighbors(score_test)
d2_nbrs_test = d2_nbrs_test[:,0:5] # we want only 5 nearest neighbors
d2_sqrd_nbrs_test = d2_nbrs_test**2
D2_test = np.sum(d2_sqrd_nbrs_test, axis = 1)
D2_log_test = np.log(D2_test)
```

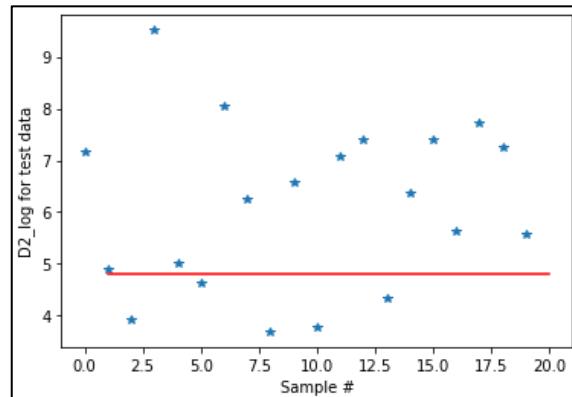


Figure 14.4: FD-KNN monitoring chart for metal-etch test samples

14.2 LOF: An Introduction

The simple nature and powerful capabilities of k-NN has put it amongst the top data-mining algorithms. However, it is not rare to encounter scenarios where k-NN-based abnormality detection gives poor performance. One such scenario involves NOC training samples distributed among different clusters with varying densities as shown in Figure 14.5. A few test samples are also shown in the figure. K-NN-based FD will correctly identify X₂, X₃, and X₄ as anomalies, but will fail to detect X₁ as an abnormal sample. This will happen because the NOC samples in cluster A are sparsely distributed which will make the threshold for fault detection large enough that the average k-NN-distance of X₁ will lie below the threshold. For our naked eye, X₁ is an obvious outlier because it is located ‘far’ away from its ‘local’ neighbors which are densely distributed. But how do we modify the k-NN algorithm to embed this logic? Well, one thing that can be done is to compare the local density of a test sample with the local densities of its neighbors. If the test sample has substantially lower density than its neighbors, then it is potentially an anomaly. This is the guiding principle behind the local outlier factor (LOF) algorithm.

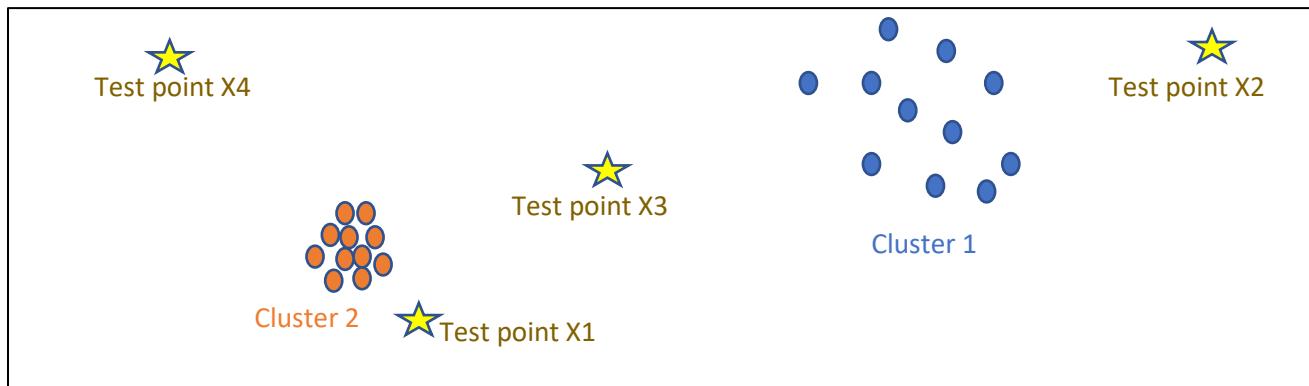


Figure: 14.5 Multi-clustered NOC data with varying densities

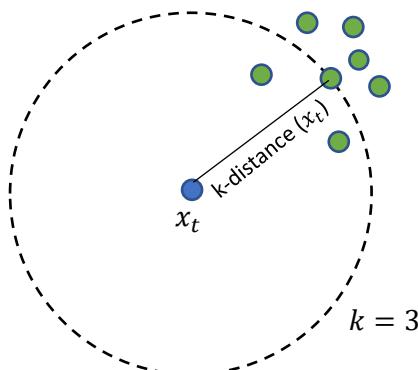
Computing LOF for a sample

Let $x_t \in \mathbb{R}^m$ be a sample. To find its LOF, we need to find the local density at the location of x_t . Alternatively, we need to find how easy is it to reach x_t from x_t 's neighbors. Let's look at all the steps involved one-by-one.

- *k-distance of x_t*

For a specified hyperparameter k , *k-distance* of x_t is simply the distance between x_t and the k^{th} nearest neighbor of x_t , i.e.,

$$\text{k-distance}(x_t) = d_k(x_t, x^*) = \sqrt{\sum_{i=1}^m (x_{ti} - x_i^*)^2} \quad x^* \text{ is } k^{\text{th}} \text{ nearest neighbor}$$



➤ *k-distance neighborhood of x_t*

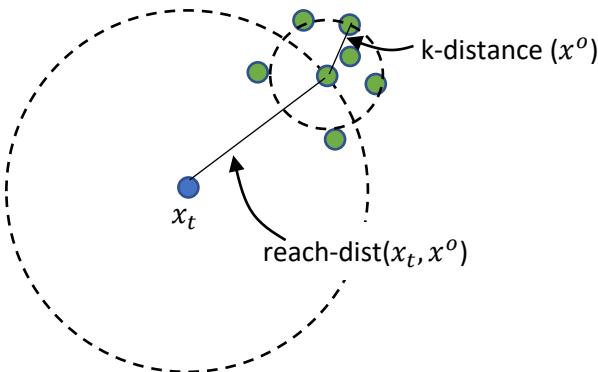
k-distance neighborhood of x_t , denoted as $N_k(x_t)$, is the set of points whose distances from x_t is not greater than *k-distance* of x_t . It usually would contain k data points (the k -nearest neighbors) but can be more than k in case of a tie.

➤ *Reachability distance of x_t from its neighbors*

Let x^o belong to $N_k(x_t)$. The reachability distance of x_t from x^o is defined as follows

$$\text{reach-dist}(x_t, x^o) = \max\{\text{k-distance}(x^o), d_k(x_t, x^*)\}$$

If x_t is far away from x^o compared to x^o 's neighbors, then reachability distance of x_t from x^o is the actual distance else $\text{reach-dist}(x_t, x^o)$ equals $\text{k-distance}(x^o)$



➤ *Local reachability density of x_t*

The local (reachability) density of x_t is simply the inverse of average reachability distance of x_t from its neighbors.

$$lrd_k(x_t) = 1 / \left(\frac{\sum_{x^o \in N_k(x_t)} \text{reach} - \text{dist}(x_t, x^o)}{|N_k(x_t)|} \right)$$

| $N_k(x_t)$ | number of points in $N_k(x_t)$

➤ *LOF of x_t*

The local reachability density of x_t is compared with those of the neighbors to compute the LOF. Specifically, the local outlier factor is the average of the ratio of local reachability density of the neighbors to that of x_t .

$$LOF_k(x_t) = \frac{\sum_{x^o \in N_k(x_t)} lrd_k(x^o)}{|N_k(x_t)| lrd_k(x_t)}$$

\uparrow
 $LOF \sim 1 \Rightarrow x_t$'s density is similar to that of neighbors.

$LOF < 1 \Rightarrow x_t$'s density is higher than neighbors' densities.

$LOF > 1 \Rightarrow x_t$'s density is lower than neighbors' densities
and may be an outlier.

A threshold for LOF is computed during training based on a ‘contamination parameter’ that specifies the proportion of anomalies in training data (or the acceptable false alarm rate if training dataset contains only NOC samples).



The hyperparameter k influences the performance of anomaly detection and its optimal value is application dependent. A small value for k will lead to inaccurate determination of local densities while a large value will diminish the differences between NOC samples and anomalies. Values between 20 to 30 work well in general.

Application of LOF for fault detection

Below we show how a monitoring chart could be generated using LOF model.

```
# LOF of each training sample
from sklearn.neighbors import LocalOutlierFactor
lof_model = LocalOutlierFactor(n_neighbors=5, novelty=True, contamination=0.05)
lof_model.fit(score_train)
```

```

lof_train = -lof_model.negative_outlier_factor_ # negative_outlier_factor_ gives the opposite
                                                LOF of the training samples
lof_CL = -lof_model.offset_

```

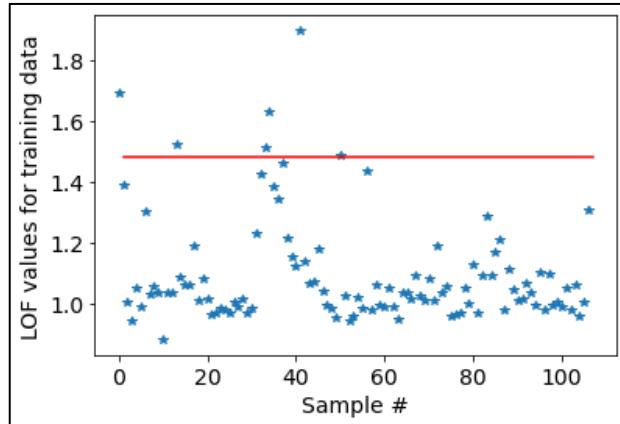


Figure 14.6 FD-LOF monitoring chart for metal-etch training samples

Figure 14.6 shows the monitoring chart for the training samples. Note that a specification of contamination of 5% results in 5 training samples being flagged as abnormal. Figure 14.7 shows the monitoring chart for the faulty test samples. 16 out of 20 samples are correctly flagged as faulty.

```

# scale and perform PCA on faulty test data; then find LOF values
score_test = pipe.transform(unfolded_TestdataMatrix)
Lof_test = -lof_model.score_samples(score_test)

print('Number of flagged faults (using control chart): ', np.sum(lوف_test > lof_CL))
# can also use predict() function of LOF class to flag faulty samples
print('Number of flagged faults (using predict): ', np.sum(lof_model.predict(score_test) == -1))

```

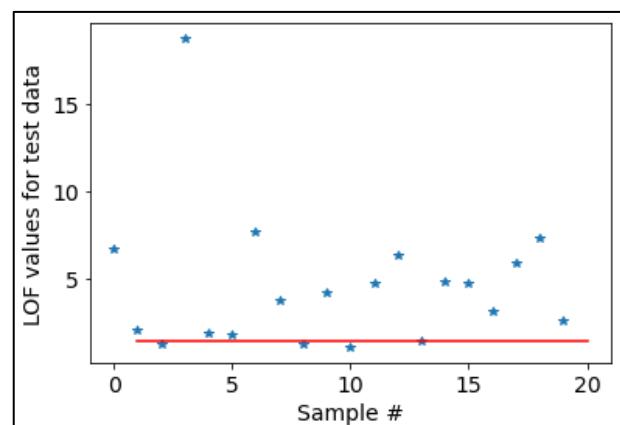


Figure 14.7: FD-LOF monitoring chart for metal-etch test samples

14.3 Isolation Forest: An Introduction

Isolation forest (IF) is an ensemble of binary decision trees, also called isolation trees. IF is an unsupervised variant of random forests and are popularly employed for outlier detection and novelty/abnormality detection. Model training involves construction of several isolation trees, and each tree is generated in such a way that every training sample ends up in its own separate leaf (i.e., gets ‘isolated’). Figure 14.8 provides a simple ‘illustration’. Here, a cluster of four NOC data points and two sample isolation trees are shown. It is apparent that each tree splits the measurement space through several binary decision rules to ‘isolate’ every training sample. Post model-training, a test sample is run through all the trained trees and its abnormality is judged based on the average depth the sample reaches in the trees. In general, an anomaly tends to lie close to the root node and therefore has shorter path length compared to the NOC points that travel deep into the trees. This is the guiding principle of isolation forests.

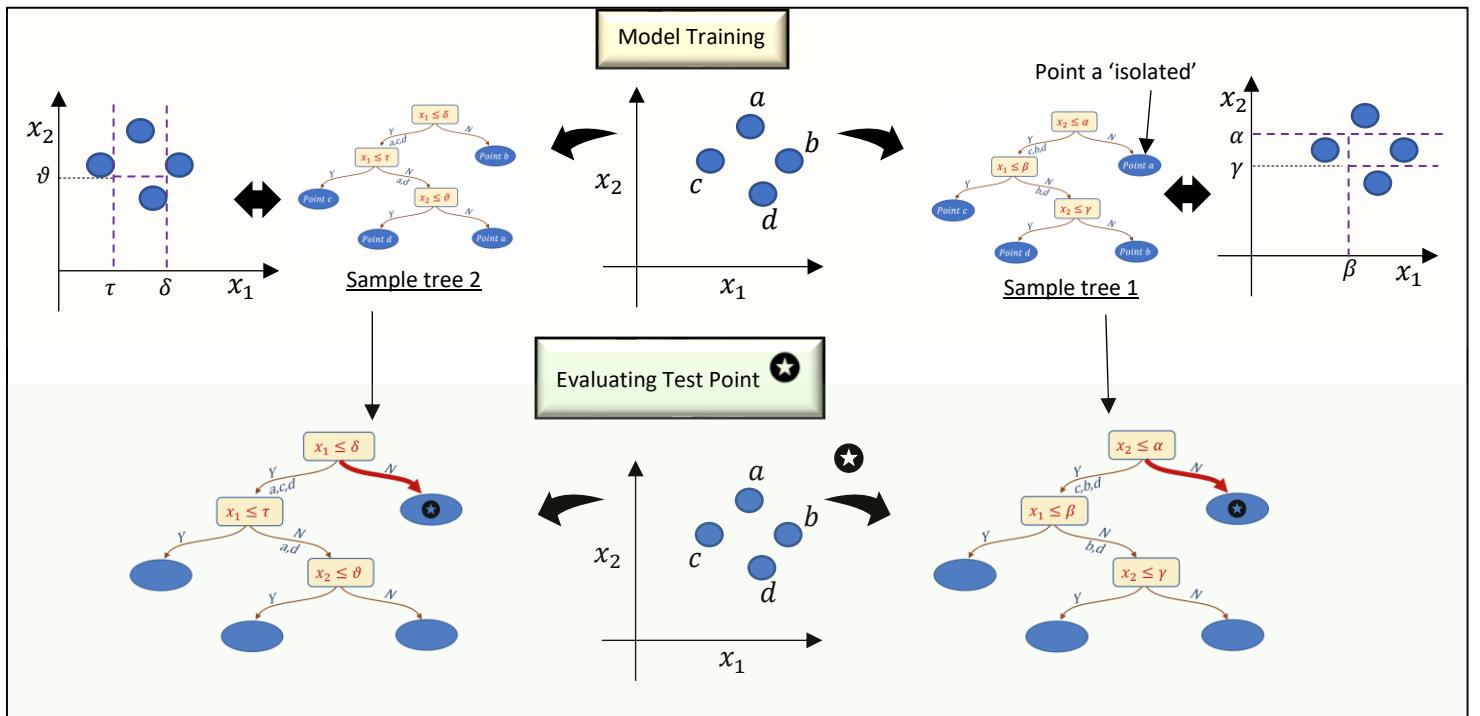


Figure 14.8: Illustration of isolation forest training and evaluation

In Chapter 13, we had seen that random forests acquire strong predictive capabilities through diversification of the constituent decision trees. In IFs, diversification is achieved by using a random subset of training samples (sampled without replacement) for training each tree. Furthermore, at every node of a tree, a feature/variable and the split value⁹⁷ is randomly chosen to form a decision rule.

⁹⁷ The split value is chosen within the maximum and minimum values of the chosen variable.

Application of isolation forest for fault detection

Below we show how a monitoring chart could be generated using IF model.

```
# IF score of each training sample
from sklearn.ensemble import IsolationForest
IF_model = IsolationForest(contamination=0.05)
IF_model.fit(score_train)

IFscore_train = -IF_model.score_samples(score_train) # score_samples returns the anomaly
# score of the input samples; the lower, the more abnormal.
IF_CL = np.percentile(IFscore_train, 95)
```

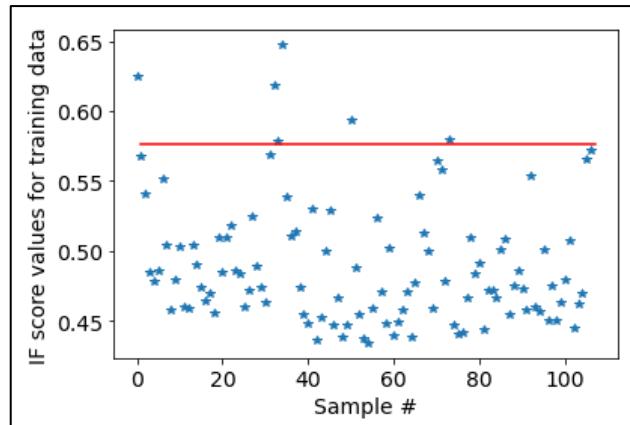


Figure 14.9 FD-IF monitoring chart for metal-etch training samples

Figure 14.9 shows the monitoring chart for the training samples. Note that 95th percentile of the training scores is used as the control limit which results in 6 training samples being flagged as abnormal. Figure 14.10 shows the monitoring chart for the faulty test samples. Only 7 out of 20 samples are correctly flagged as faulty, suggesting that the IF model is inappropriate for this dataset.

```
# scale and perform PCA on faulty test data; then find IF score values
score_test = pipe.transform(unfolded_TestdataMatrix)
IFscore_test = -IF_model.score_samples(score_test)

print('Number of flagged faults (using control chart): ', np.sum(IFscore_test > IF_CL))
# can also use predict() function of IF class to flag faulty samples
print('Number of flagged faults (using predict): ', np.sum(IF_model.predict(score_test) == -1))
```

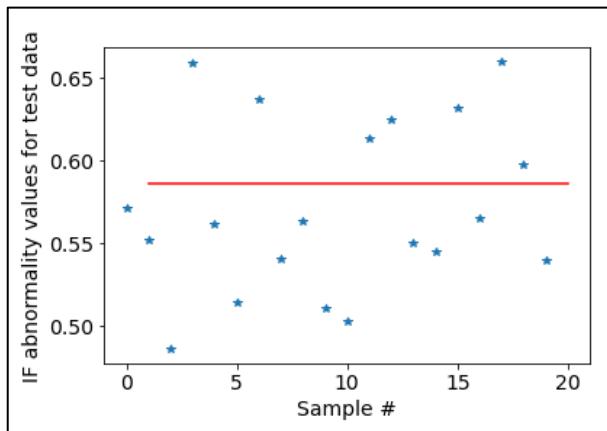
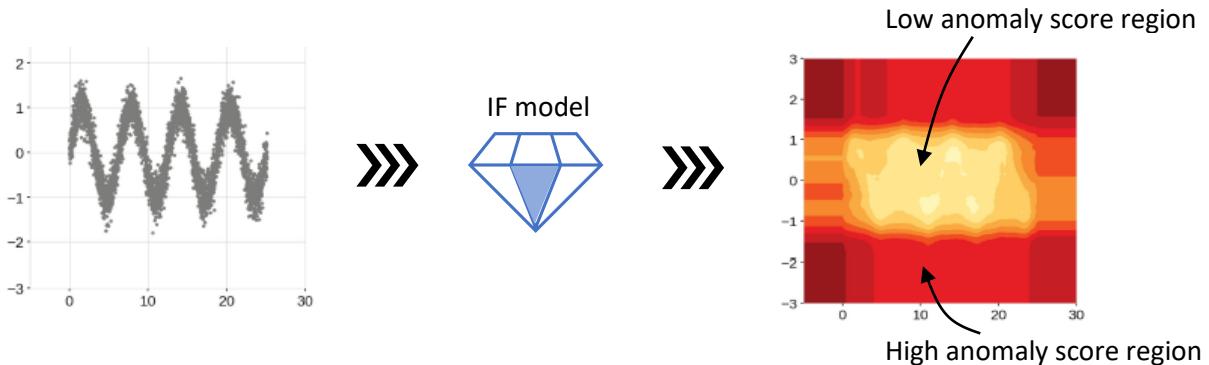


Figure 14.10: FD-IF monitoring chart for metal-etch test samples

Extended Isolation Forests

Consider the following NOC samples with sinusoidal distribution* and the corresponding score map. The score map* indicates poor performance of IF algorithm. The score map would make one believe that the NOC data is present in a rectangular blob and not in a sinusoidal pattern.



The IF failed in the above illustration due to the restriction of using horizontal and vertical splits only. An alternative algorithm, called Extended Isolation Forest (<https://github.com/sahandha/eif>), has been provided by Hariri et al. that uses random slopes for the splits and therefore achieves better performance (albeit at a higher computational cost).

*Diagrams made available under Creative Commons Attribution 4.0 License
(creativecommons.org/licenses/by/4.0) by Hariri et al. (ieeexplore.ieee.org/document/8888179)

Summary

In this chapter, we studied anomaly detection methods that rely on different notions of proximity. Specifically, we covered *k*-NN, local outlier factor, and isolation forests. Although, isolation forests are technically not included in the class of proximity-based techniques, we clubbed them with other proximity-based methods in this chapter due to their treatment of anomalies as samples that are ‘far and between’. With this, we have covered the major classical ML techniques that are widely employed for fault detection in process systems. We will next move onto artificial neural networks-based techniques for process monitoring.

Part 5

Artificial Neural Networks for Process Monitoring

Chapter 15

Fault Detection & Diagnosis via Supervised Artificial Neural Networks Modeling

It won't be an exaggeration to say that artificial neural networks (ANNs) are currently the most powerful modeling construct for describing generic nonlinear processes. ANNs can capture any kind of complex nonlinearities, don't impose any specific process characteristics, and don't demand specification of process insights prior to model fitting. Furthermore, several recent technical breakthroughs and computational advancements have enabled (deep) ANNs to provide remarkable results for a wide range of problems. Correspondingly, ANNs have re-caught the fascination of data scientists and the process industry is witnessing a surge in successful applications of ML-based process control, predictive maintenance, inferential modeling, and process monitoring.

ANNs can be used in both supervised and unsupervised learning settings. While we will cover the supervised learning-based FDD applications of ANNs in this chapter, unsupervised learning is covered in the next chapter. Supervised fitting of ANN models are applicable if you have adequate number of historical faulty samples (so that you can fit a fault classification model) or your signals are categorizable into predictors and response variables (so that you can fit a regression model and monitor residuals). Different forms of ANN architectures have been devised (such as FFNNs, RNNs, CNNs) to deal with datasets with different characteristics. CNNs are mostly used with image data and therefore, we will study FFNN and RNN in this chapter.

There is no doubt that ANNs have proven to be monstrously powerful. However, it is not easy to tame this monster. If the model hyperparameters are not set judiciously, it is very easy to end up with disappointing results. The reader is referred to Part 3 of Book 1 of this series for a detailed exposition on ANN training strategies and different facets of ANN models. In this chapter, the focus is on exposing the user to how ANNs can be used to build process monitoring applications. Specifically, the following topics are covered

- Introduction to ANNs
- Introduction to RNNs
- Process monitoring using ANNs via external analysis

15.1 ANN: An Introduction

Artificial neural networks (ANNs) are nonlinear empirical models which can capture complex relationships between input-output variables via supervised learning or recognize data patterns via unsupervised learning. Architecturally, ANNs were inspired by human brain and are a complex network of interconnected neurons as shown in Figure 15.1. An ANN consists of an input layer, a series of hidden layers, and an output layer. The basic unit of the network, neuron, accepts a vector of inputs from the source input layer or the previous layer of the network, takes a weighted sum of the inputs, and then performs a nonlinear transformation to produce a single real-valued output. Each hidden layer can contain any number of neurons.

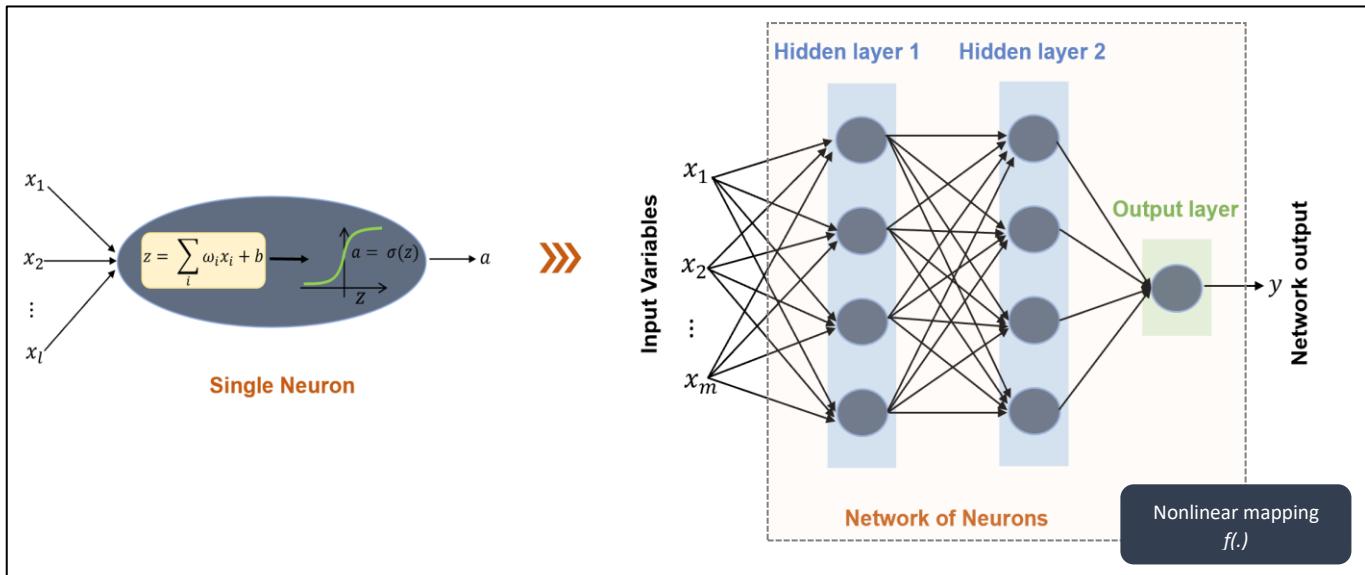


Figure 15.1: Architecture of a single neuron and feedforward neural network with 2 hidden layers

The network shown in Figure 15.1 is an example of a fully-connected feed-forward neural network (FFNN), the most common type of ANN. In FFNN, signals flow in only one direction, from the input layer to the output layer via hidden layers. Neurons between consecutive layers are connected fully pairwise and neurons within a layer are not connected.



What is deep learning

In a nutshell, using an ANN with a large number of hidden layers to find relationship/pattern in data is deep learning (technically, ≥ 2 hidden layers implies a deep neural network (DNN)). Several recent algorithmic innovations have overcome the model training issues for DNNs which have resulted in the DNN-led AI revolution we are witnessing today.

15.2 Process Modeling via FFNN: Combined Cycle Power Plant (CCPP) Case Study

We will use data from a CCPP to illustrate the ease with which neural network models can be built in Python. The dataset⁹⁸ comes from a combined cycle power plant composed of gas turbine (GT), steam turbine (ST) and heat recovery steam generator as shown in Figure 15.2. Here, energy from fuel combustion generates electricity in a gas turbine and residual energy in the hot exhaust/flue gas from GT is recovered to produce steam. This steam is used to generate further electricity in a steam turbine. The combined electric power generated by both GT and ST over a period of 6 years (with hourly averages) is provided in the dataset. Hourly average values of ambient temperature (AT), ambient pressure (AP), relative humidity (RH), and exhaust vacuum (V) are also provided. These variables influence the net hourly electrical energy output (also provided in the dataset) of the plant operating at full load and will be the target variable in our ANN model.

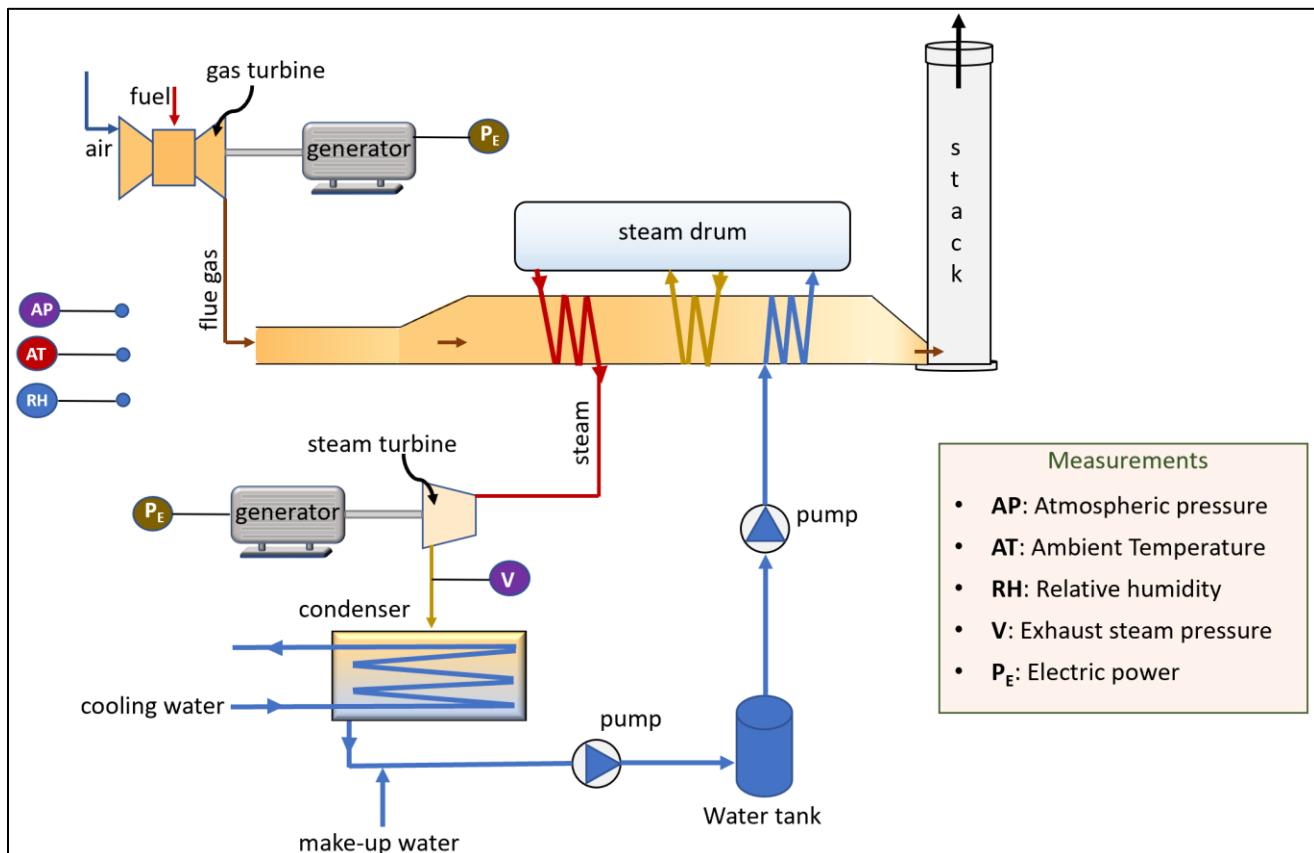


Figure 15.2: Simplified schematic of combined cycle power plant

⁹⁸ UCI Machine Learning Repository: <https://archive.ics.uci.edu/ml/datasets/combined+cycle+power+plant>

Let us first explore the dataset.

```
# import required packages and read data
import numpy as np, pandas as pd
import matplotlib.pyplot as plt

data = pd.read_excel('Folds5x2_pp.xlsx', usecols = 'A:E').values
X y = data[:,0:4], data[:,4][:,np.newaxis]

# plot input vs output for each input
plt.figure(), plt.plot(X[:,0], y, '*'), plt.title('AT vs EP')
plt.figure(), plt.plot(X[:,1], y, '*'), plt.title('V vs EP')
plt.figure(), plt.plot(X[:,2], y, '*'), plt.title('AP vs EP')
plt.figure(), plt.plot(X[:,3], y, '*'), plt.title('RH vs EP')
```

Figure 15.3 clearly indicates the impact of input variables on the electrical power (EP).

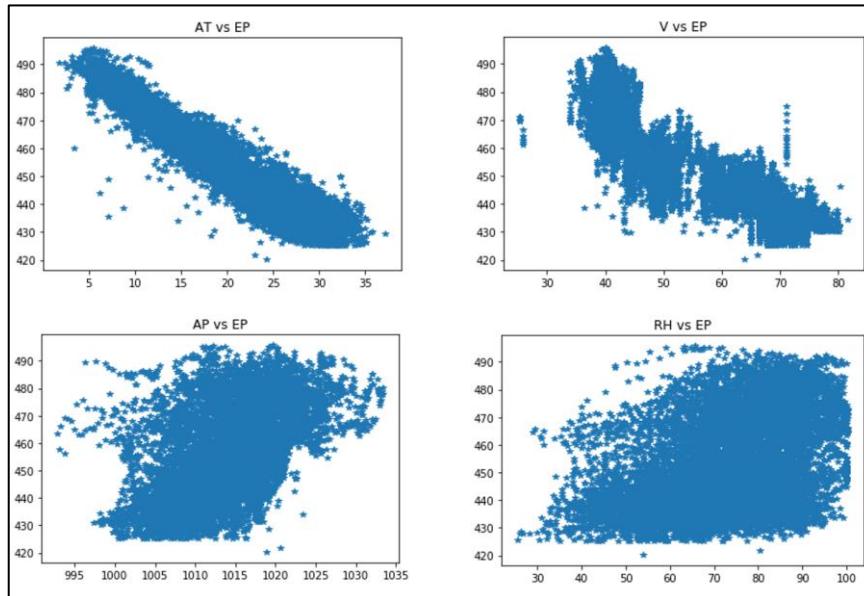


Figure 15.3: Plots of influencing variables (on x-axis) vs Electrical Power (on y-axis)

There is also a hint of nonlinear relationship between exhaust vacuum and power. While it may seem that AP and RH do not influence power strongly, it is a known fact that power increases with increasing AP and RH individually⁹⁹. Let us now build a FFNN model with 2 hidden layers to predict power. We first split the dataset into training and test data, and then scale the variables.

⁹⁹ Pinar Tufekci, Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods, Electrical Power and Energy Systems, 2014

```
# separate train 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_scaled = X_scaler.fit_transform(X_train)
X_test_scaled = X_scaler.transform(X_test)

y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train)
y_test_scaled = y_scaler.transform(y_test)
```

To build FFNN model, we will import relevant Keras libraries and add different layers of the network sequentially. The Dense library is used to define a layer that is fully-connected to the previous layer.

```
# import Keras libraries
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

# define model
model = Sequential()
model.add(Dense(8, activation='relu', kernel_initializer='he_normal', input_shape=(4,)))
# 8 neurons in 1st hidden layer
model.add(Dense(5, activation='relu', kernel_initializer='he_normal'))
# 5 neurons in 2nd layer
model.add(Dense(1))
# 1 neuron in output layer
```

The above 4-line code completely define the structure of the FFNN. Next, we will compile and fit the model.

```
# compile model
model.compile(loss='mse', optimizer='Adam') # mean-squared error is to be minimized

# fit model
model.fit(X_train_scaled, y_train_scaled, epochs=25, batch_size=50)

# predict y_test
y_test_scaled_pred = model.predict(X_test_scaled)
y_test_pred = y_scaler.inverse_transform(y_test_scaled_pred)
```

The above lines are all it takes to build FFNN and make predictions. Quite convenient, isn't it? Figure 15.4 compares actual vs predicted power for the test data.

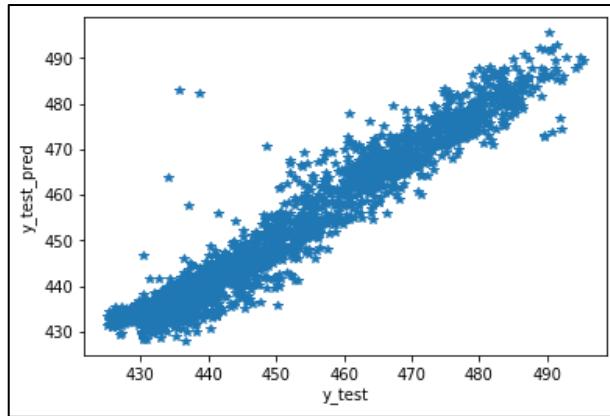


Figure 15.4: Actual vs predicted target for CCPP dataset (obtained $R^2 = 0.93$)

For the relatively simple CCPP dataset, we can obtain a reasonable model with just 1 hidden layer with 2 neurons. Nonetheless, this example has now familiarized us with the process of creating a DNN.

Understanding flow of data in a FFNN

If you execute the command `model.summary()` for the model developed in the previous section, you will be presented with the following information which details the number of parameters to be estimated in each layer of the network. Let's understand how these numbers are obtained which will help clarify the flow of information in FFNN.

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 8)	40
dense_1 (Dense)	(None, 5)	45
dense_2 (Dense)	(None, 1)	6
<hr/>		
Total params: 91		
Trainable params: 91		
Non-trainable params: 0		

We know that each input sample is 4-dimensional ($x \in R^4$). In the forward pass (also called forward propagation), the augmented input is first processed by the neurons of the first hidden layer. In the j^{th} neuron of this layer, the weighted sum of the inputs are non-linearly transformed via an activation function, g

$$a_j = g(w_j^T x + b_j)$$

$w_j \in R^4$ are the weights applied to the inputs and b_j is the bias added to the sum. Thus, each neuron has 5 parameters (4 weights and a bias) leading to 40 parameters for all the 8 neurons of the 1st layer that need to be estimated. Outputs of all the 8 neurons form vector $a^{(1)} \in R^8$

$$a^{(1)} = g^{(1)}(W^{(1)}x + b^{(1)})$$

where each row of $W^{(1)} \in R^{8 \times 4}$ contains the weights of a neuron. The same activation function is used by all the neurons of a layer. $a^{(1)}$ becomes the input to the 2nd hidden layer.

$$a^{(2)} \in R^5 = g^{(2)}(W^{(2)}a^{(1)} + b^{(2)})$$

where $W^{(2)} \in R^{5 \times 8}$. Each neuron in the 2nd layer has 8 weights and a bias parameter, leading to 45 parameters in the layer. The final output layer had a single neuron with 6 parameters and no activation function, giving the network output as follows

$$a^{(3)} = \hat{y} = (w^{(3)})^T a^{(2)} + b^{(3)}$$

where $w^{(3)} \in R^5$.

15.3 RNN: An Introduction

Recurrent neural networks (RNNs) are ANNs for dealing with sequential data, where the order of occurrence of data holds significance. In the FFNN-based NARX model that we studied in the previous section, there is no provision for implicit or explicit specification of temporal/sequential nature of data, i.e., $x(k-2)$ comes before $x(k-1)$ for example. There is no efficient mechanism to specify this temporal order of data in a FFNN. RNNs accomplish this by processing elements of a sequence recurrently and storing a hidden state that summarizes the past information during the processing. The basic unit in a RNN is called a RNN cell which simply contains a layer of neurons. Figure 15.5 shows the architecture of a RNN consisting of a single cell and how it processes a data sequence with ten samples.

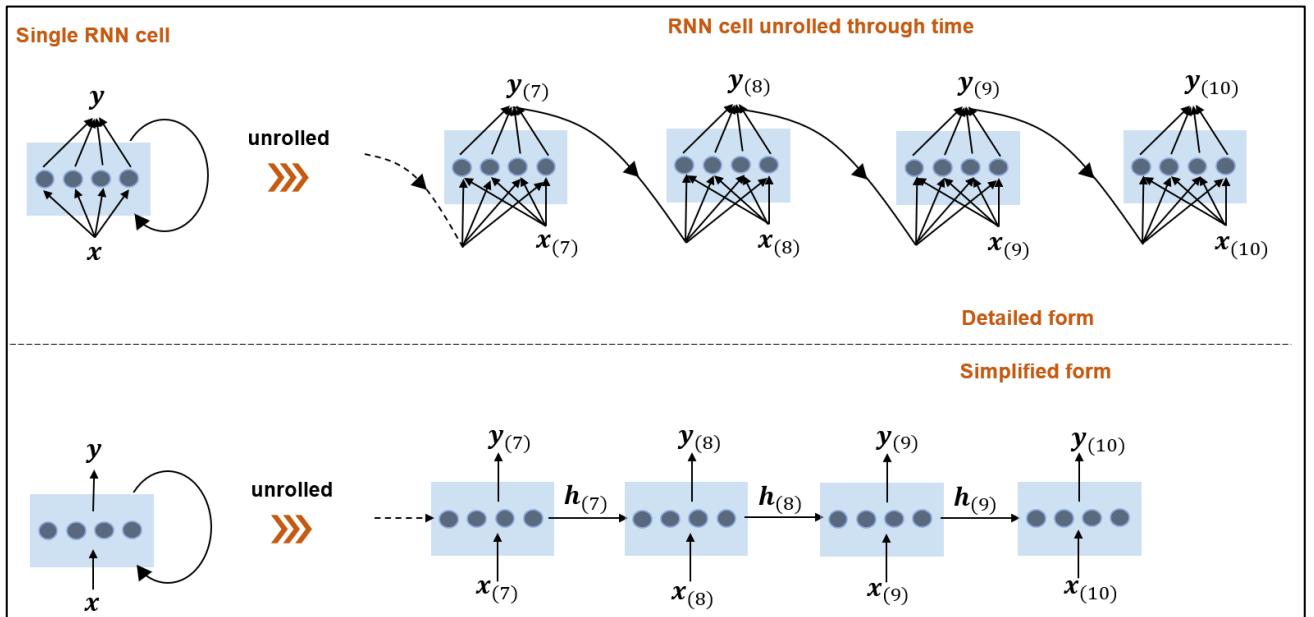


Figure 15.5: Representation of an RNN cell in rolled and unrolled format. The feedback signal in rolled format denotes the recurrent nature of the cell. Here, the hidden state (h) is assumed to be same as intermediate output (y). $h_{(0)}$ is usually taken as zero vector.

We can see that the ten samples are processed in the same order of their occurrence and not at once. An output, $y_{(i)}$, is generated at the i^{th} step and then fed to the next step for processing along with $x_{(i+1)}$. By way of this arrangement, $y_{(i+1)}$ is a function of $x_{(i+1)}$ and $y_{(i)}$. Since, $y_{(i)}$ itself is a function of $x_{(i)}$ and $y_{(i-1)}$, $y_{(i+1)}$ effectively becomes a function of $x_{(i+1)}$, $x_{(i)}$, and $y_{(i-1)}$. Continuing the logic further implies that the final sequence output, $y_{(10)}$, is a function of all ten inputs of the sequence, that is, $x_{(10)}, x_{(9)}, \dots, x_{(1)}$. This ‘recurrent’ mechanism leads to efficient capturing of temporal patterns in data.

RNN outputs

If the neural layer in the RNN cell in Figure 15.5 contains n neurons (n equals 4 in the shown figure), then each $y_{(i)}$ or $h_{(i)}$ is a n -dimensional vector. For simple RNN cells, $y_{(i)}$ equals $h_{(i)}$. Let x be a m -dimensional vector. At any i^{th} step, we can write the following relationship

$$y_{(i)} = g(W_x x_{(i)} + W_y y_{(i-1)} + b)$$

where $W_x \in R^{n \times m}$ with each row containing the weight parameters of a neuron as applied to x vector, $W_y \in R^{n \times n}$ with each row containing the weight parameters of a neuron as applied to y vector, $b \in R^n$ contains the bias parameters, and g denotes the activation function. The same neural parameters are used at each step.

If all the outputs of the sequence are of interest, then it is called sequence-to-sequence or many-to-many network. However, very often only the last step output is needed, leading to sequence-to-vector or many-to-one network. Moreover, the last step output may need to be further processed and so a FC layer is often added. Figure 15.6 shows one such topology.

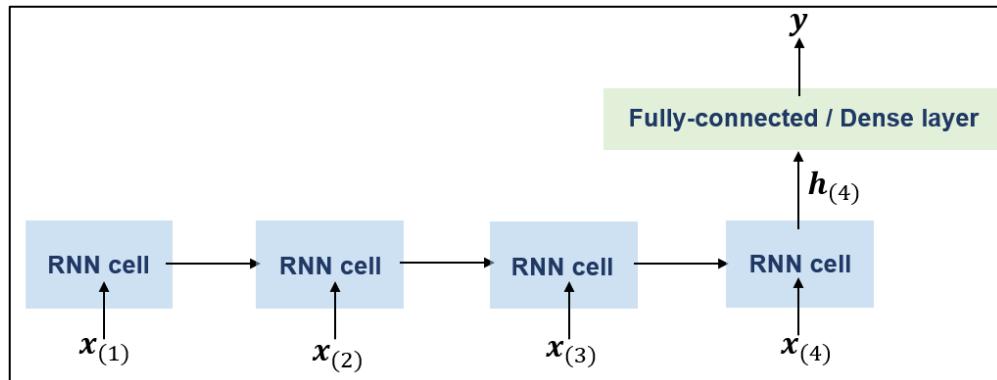


Figure 15.6: Sequence to vector RNN topology with a FC layer

LSTM networks

RNNs are powerful dynamic models, however, the vanilla RNN (with a single neural layer in a cell) introduced before faces difficulty learning long-term dependencies, i.e., when number of steps in a sequence is large ($\sim \geq 10$). This happens due to the vanishing gradient problem during gradient backpropagation. To overcome this issue, LSTM (Long Short-Term Memory) cells have been devised which are able to learn very long-term dependencies (even greater than 1000) with ease. Unlike vanilla RNN cells, LSTM cells have 4 separate neural layers as shown in Figure 15.7. Moreover, in a LSTM cell, the internal state is stored in two separate

vectors, $h_{(t)}$ or hidden state and $c_{(t)}$ or cell state. Both these states are passed from one LSTM cell to the next during sequence processing. $h_{(t)}$ can be thought of as the short-term state/memory and $c_{(t)}$ as the long-term state/memory and hence the name LSTM.

The vector outputs of the FC layers interact with each-other and the long-term state via three ‘gates’ where element-wise multiplications occur. These gates control what information go into the long-term and short-term states at any sequence processing step. A quick description of each gate’s purpose is the following:

- **Forget gate:** determines what parts of long-term state, $c_{(t)}$, are retained and erased
- **Input gate:** determines what parts of new information (obtained from processing of $x_{(t)}$ and $h_{(t-1)}$) are stored in long-term state
- **Output gate:** determines what parts of long-term state are passed on as short-term state

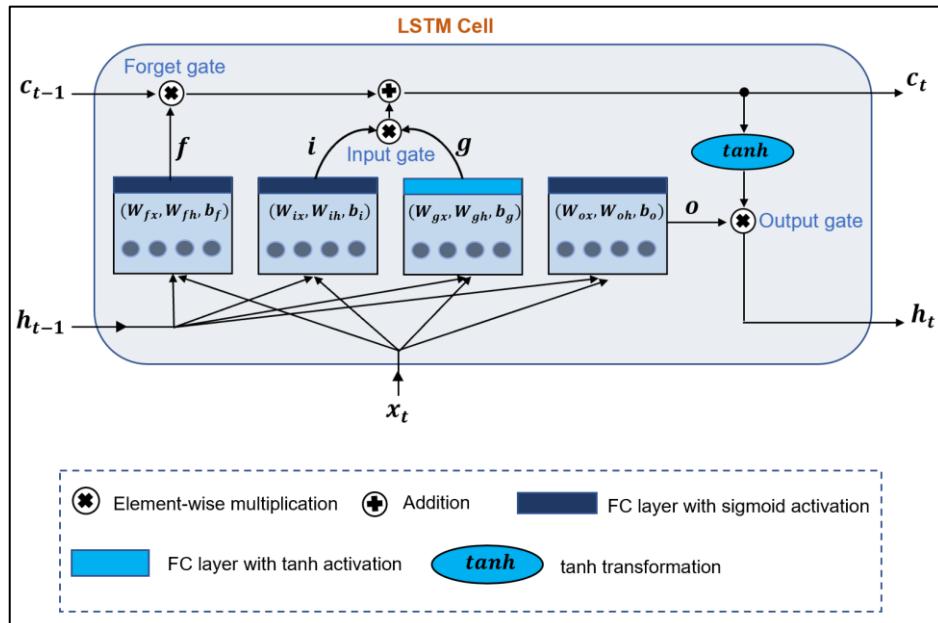
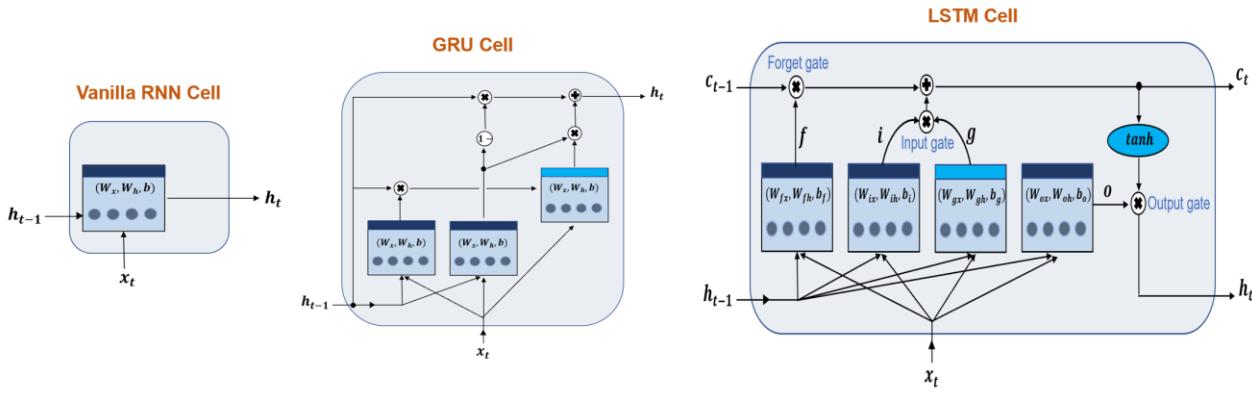


Figure 15.7: Architecture of a LSTM cell. Three FC layers use sigmoid activation functions and one FC layer uses *tanh* activation function. Each of these neural layers have its own parameters W_x , W_h , and b

This flexibility in being able to manipulate what information are passed down the chain during sequence processing is what makes LSTM networks so successful at capturing long-term patterns in sequential datasets. Consequently, LSTM network is the default RNN architecture employed now-a-days. In Chapter 20, we will employ an LSTM model to predict remaining useful life of gas turbines. Let’s next learn how to use supervised ANN models for process fault detection.

Vanilla RNN cell vs GRU cell vs LSTM cell

There is another popular variant of RNN cell, called GRU cell. As shown in the illustration below, GRU cell is simpler than LSTM cell. GRU cell has 3 neural layers and its internal state is represented using a single vector, $h_{(t)}$. For several common tasks, GRU cell-based RNN seems to provide similar performance as LSTM cell-based RNN and therefore, it is slowly gaining more popularity.



15.4 ANN-based External Analysis for Fault Detection in a Debutanizer Column

Consider Figure 15.8 below that illustrates the ANN-based external analysis¹⁰⁰ strategy for process monitoring. The idea is simple: predict response variables using predictor variables via ANN, compute the residuals, and then monitor the residuals using classical statistical techniques.

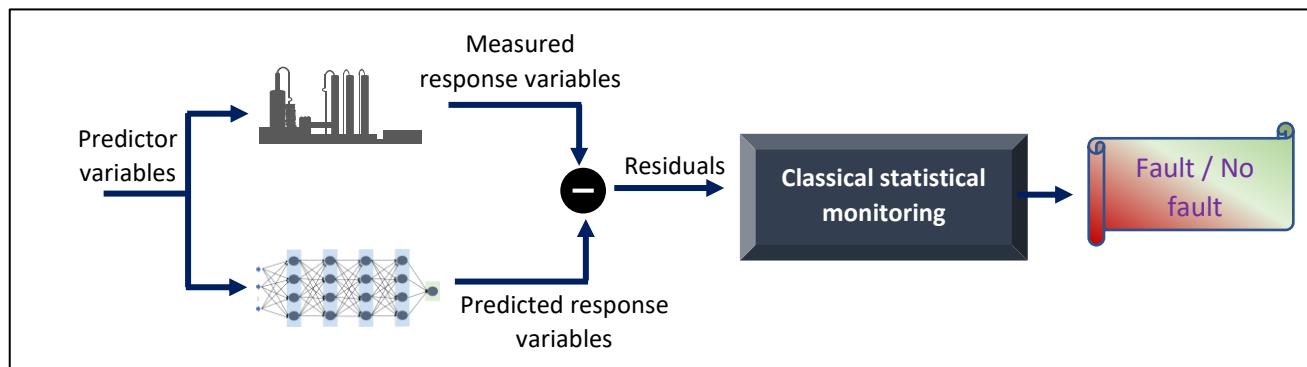


Figure 15.8: External analysis approach for process monitoring

¹⁰⁰ Yamamoto et al., Applications of statistical process monitoring with external analysis to an industrial monomer plant. *IFAC Advanced Control of Chemical Processes*, 2003.

Let's apply this strategy to detect faults in a debutanizer column from a petroleum refinery. Debutanizer columns are standard units in petroleum refineries and are used to convert raw naphtha feed into LPG (as top product) and gasoline (as bottom product). The butane (C4) content in gasoline product is desired to be kept low and is monitored regularly via gas chromatography. To add another layer of supervision, a model is desired that can provide 'soft' measurements of the C4 content and therefore, any deviation in actual C4 content from the expected value can be used to raise an alert to the plant operators. For this purpose, we will build a fault detection model using FFNN-based externally analysis. The dataset¹⁰¹ is provided as supplementary material at <https://link.springer.com/book/10.1007/978-1-84628-480-9>.

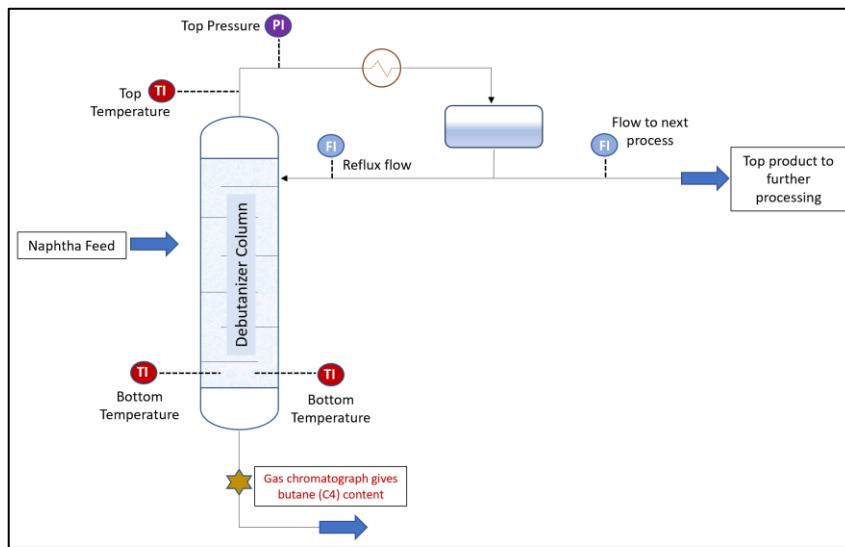


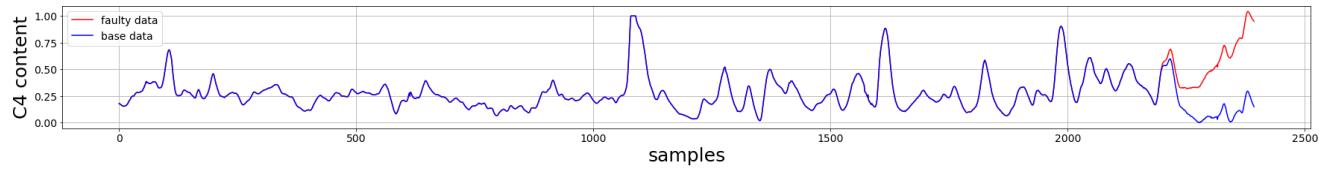
Figure 15.9: Debutanizer column with location of measured variables

The base dataset contains 2394 samples of (normalized) input-output process values. An artificial sensor drift is introduced in the last 200 samples to simulate faulty sensor conditions. Let's see if we can detect this fault at the earliest. We start with loading the data.

```
# import required packages and read data
import numpy as np, pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import r2_score

data = np.loadtxt('debutanizer_data_withFault.txt') # drift starts from last 200 sample onwards
```

¹⁰¹ Fortuna et. al., Soft sensors for monitoring and control of industrial processes, Springer, 2007



```
# separate training and test data
data_train = data[:-300,:]
data_test = data[-300:,:] # last 300 samples used as test data

X_train, y_train = data_train[:,0:-1], data_train[:, -1][:, np.newaxis]
X_test, y_test = data_test[:,0:-1], data_test[:, -1][:, np.newaxis]

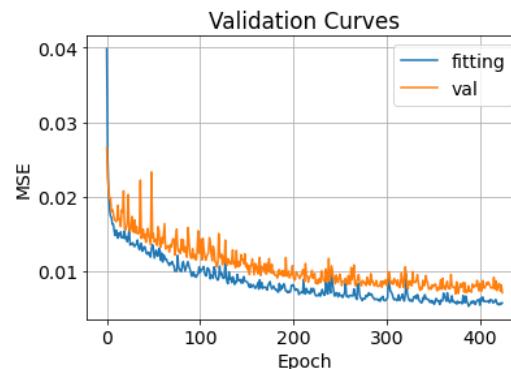
# separate estimation and validation data
X_est, X_val, y_est, y_val = train_test_split(X_train, y_train, test_size = 0.2, random_state = 100)
```

We are now ready to fit our ANN model.

```
# define, compile, and fit model
model = Sequential()
model.add(Dense(40, kernel_regularizer=regularizers.L1(0.000001), activation='relu',
                kernel_initializer='he_normal', input_shape=(7,)))
model.add(Dense(20, kernel_regularizer=regularizers.L1(0.000001), activation='relu',
                kernel_initializer='he_normal'))
model.add(Dense(1, kernel_regularizer=regularizers.L1(0.000001)))

model.compile(loss='mse', optimizer=Adam(learning_rate=0.005))
es = EarlyStopping(monitor='val_loss', patience=50)
history = model.fit(X_est, y_est, epochs=2000, batch_size=64, validation_data=(X_val, y_val),
                     callbacks=es)
```

The plot¹⁰² below shows the evolution of mean squared prediction error over the fitting and validation datasets. As can be expected, mse is higher for validation data and the flattening of the curves indicate model fitting convergence.



¹⁰² The online code shows how to generate such validation curves

```
# predict C4 content
y_test_pred = model.predict(X_test)
y_val_pred = model.predict(X_val)
y_est_pred = model.predict(X_est)
y_train_pred = model.predict(X_train)

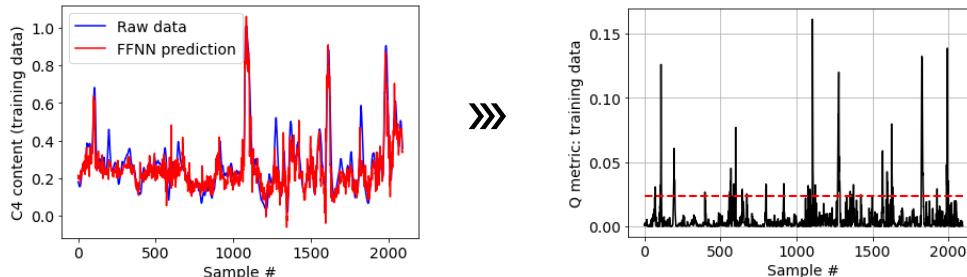
print('R2 for validation data:', r2_score(y_val, y_val_pred))
print('R2 for fitting data:', r2_score(y_est, y_est_pred))

>>> R2 for validation data: 0.71
>>> R2 for fitting data: 0.78
```

The prediction accuracy is reasonably good and therefore, we can proceed with fault detection model development with the obtained ANN model. Let generate monitoring chart for the training data.

```
# Q metric for training samples
error_train = y_train - y_train_pred
Q_train = np.sum(error_train*error_train, axis = 1)
Q_CL = np.percentile(Q_train, 95)

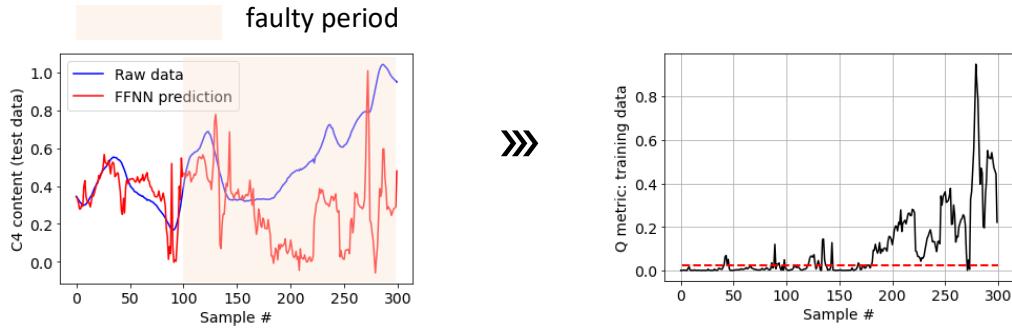
plt.figure(), plt.plot(Q_train, color='black')
plt.plot([1,len(Q_train)],[Q_CL,Q_CL], linestyle='--',color='red', linewidth=2)
plt.xlabel('Sample #'), plt.ylabel('Q metric: training data')
```



The control chart for the test data shows prompt detection of the fault with monitoring statistic mostly lying below the alert threshold during the first (fault-free) hundred samples.

```
# Q metric for test samples
error_test = y_test - y_test_pred
Q_test = np.sum(error_test*error_test, axis = 1)

plt.figure(), plt.plot(Q_test, color='black')
plt.plot([1,len(Q_test)],[Q_CL,Q_CL], linestyle='--',color='red', linewidth=2)
plt.xlabel('Sample #'), plt.ylabel('Q metric: training data')
```



15.5 Fault Classification using ANNs

If historical faulty samples are available, then one can use FFNN or RNN models to directly predict the fault classes of test samples. The figure below shows a sample architecture that can help achieve this.

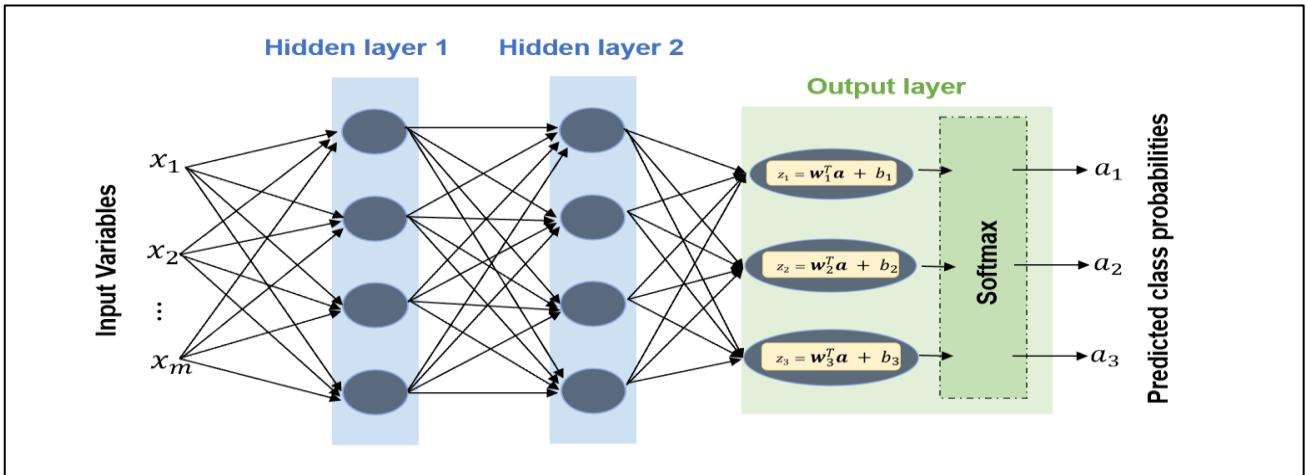


Figure 15.10: FFNN architecture for mutually exclusive output classes

Softmax activation function in output layer

Softmax is an exponential function that generates normalized activations so that they sum up to 1. In Figure 15.10, activation a_j ($j \in [1,2,3]$) is generated as follows

$$a_j = g_{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^3 e^{z_k}}$$

In ANN world, the pre-activations (z_j) that are fed as inputs to the softmax function are also called logits. The softmax activations lie between 0 and 1, and hence, they are interpreted as class-membership probabilities. The predicted class label is taken as the class with maximum probability (or activation)

$$\hat{y} = \text{argmax} (a_j, j \in [1,2,3])$$

The predicted class probabilities (and not the predicted class label) are directly used during model training.

Loss functions

For binary-class classification problems, binary cross-entropy is the default loss function. Let y (can take value 0 and 1) be the true label for a data sample and p (or \hat{y}) be the predicted probability (of $y = 1$) obtained from sigmoid output layer. The cross-entropy loss is given by

$$\text{Cross - entropy Loss} = -y * \log(p) - (1 - y) * \log(1 - p) = \begin{cases} -\log(1 - p), & y = 0 \\ -\log(p), & y = 1 \end{cases}$$

The above expression is generalized as follows for a multiclass classification, where overall loss is sum of separate losses for each class label

$$\text{Multi - class cross - entropy Loss} = - \sum_{c=1}^{\# \text{ of classes}} y_c \log(p_c)$$

where, y_c and p_c are binary indicator and predicted probability of a sample belonging to class c , respectively. Note that for multi-class classification, the target variable will be in one-hot encoded form.

Summary

In this chapter, we familiarized ourselves with the basic ANN architectures that are employed to handle static and dynamic processes. We looked at ways to deploy ANN models for process fault detection; specifically, we worked through an implementation of FFNN-based process monitoring solution wherein the residuals between actual measurements and model predictions are used to ascertain the presence of process faults. In the following chapter, we will see how process monitoring solutions can be built using unsupervised neural network models.

Chapter 16

Fault Detection & Diagnosis via Unsupervised Artificial Neural Networks Modeling

In the previous chapter, we looked at supervised fitting of artificial neural networks where either the faults labels were available for historical samples or the process variables were divided into predictors and response variable sets. However, you are very likely to encounter situations where you only have NOC samples in your training dataset without any predictor/response division. In Part 3 of this book, we studied a powerful technique suitable for such datasets, called PCA; PCA, however, is limited to linear processes. Nonetheless, the underlying mechanism of extracting the most representative features of training dataset and compressing it into a feature space with reduced dimensionality need not be limited to linear systems. ANNs excel at handling nonlinear systems and extracting hidden patterns in high-dimensional datasets. Unsurprisingly, clever neural network-based architectures have been devised to enable unsupervised fitting of nonlinear datasets. Two popular models in this category are autoencoders (AEs) and self-organizing maps (SOMs)

Autoencoders are ANN-based counterparts of PCA for nonlinear processes. Here, low-dimensional latent feature space is derived via nonlinear transformation and, just like we did for PCA, the systematic variations in the feature space and the reconstruction errors are handled separately to provide the monitoring statistics. Autoencoders are very popular for building FDD solutions for nonlinear processes. They are also commonly used to provide intermediate low-dimensional features which are then used for subsequent modeling (clustering, fault classification, etc.). SOM is another variant of neural network-based architecture that project a high-dimensional dataset onto a 2D grid (yes, you read that right!). Here, latent variables are not derived, albeit the focus is on ensuring that the topology of the projected data is similar to that in the original measurement space. This feature renders SOMs very useful for data visualization, clustering, and fault detection applications.

We will undertake in depth study of both these powerful techniques in this chapter. Specifically, the following topics are covered

- Introduction to autoencoders and self-organizing maps
- Fault detection and diagnosis using autoencoders: application to FCCU process
- Fault detection and diagnosis using SOMs: application to semiconductor dataset

16.1 Autoencoders: An Introduction

An autoencoder (AE) in its basic form is a 3-layered ANN consisting of an input layer, a hidden layer, and an output layer as shown in Figure 16.1. An AE takes an input $x \in \mathbb{R}^n$ and predicts a reconstructed $\hat{x} \in \mathbb{R}^n$ as an output. To prevent the network from trivially copying x to \hat{x} , the hidden layer is constrained to be much smaller than n (the number of neurons in the hidden layer, say m , gives the dimension of the latent/feature space). This forces the network to capture only the systematic variations in input data and learn only the most representative features as the latent variables. The nonlinear activation function of the neurons in the hidden layer enables the latent variables to be nonlinearly related to the input variables. During model fitting, the gap between x and \hat{x} (termed reconstruction error) is minimized to find network parameters. The basic AE network can be made deeper by adding more hidden layers resulting in deep (or stacked) autoencoders.

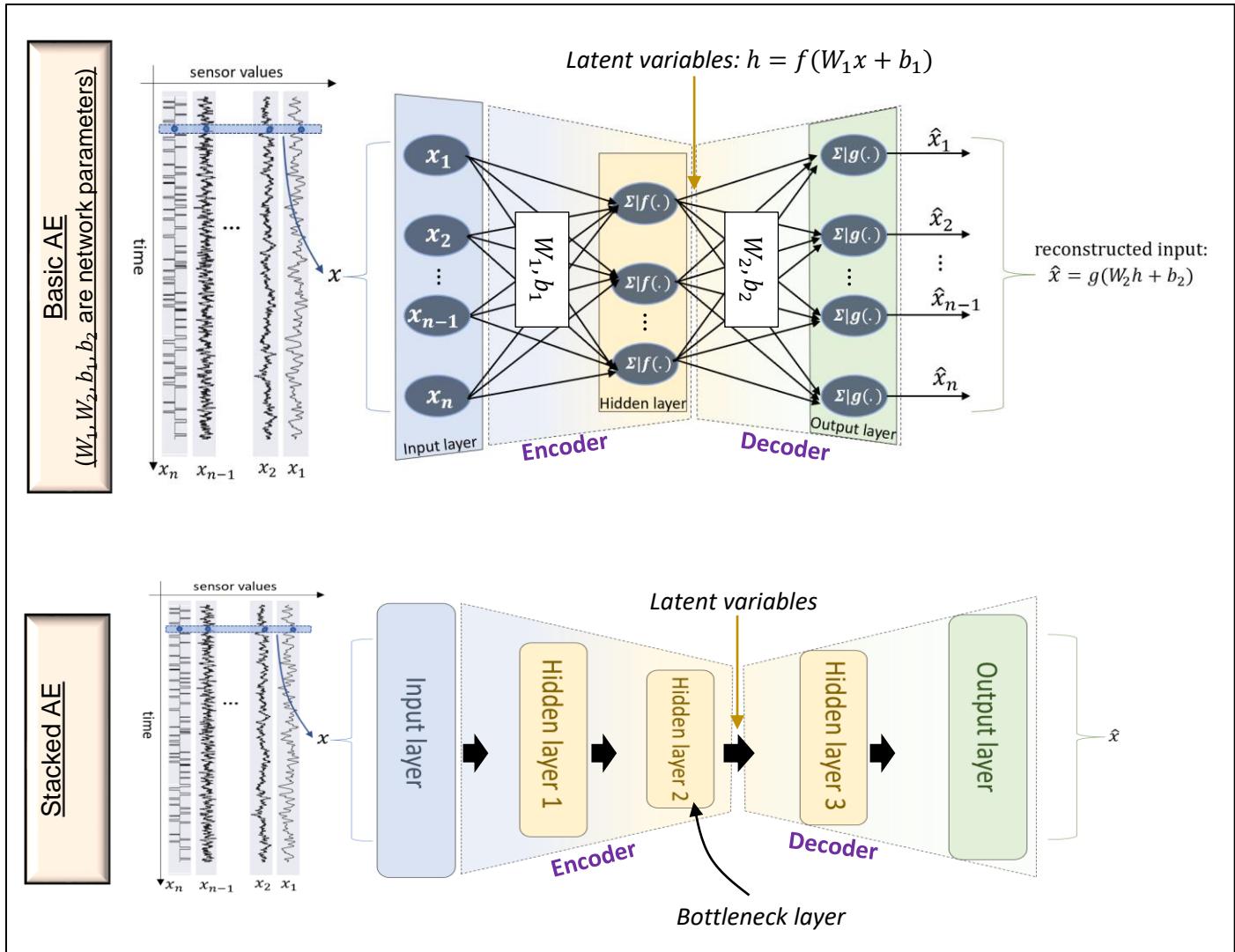
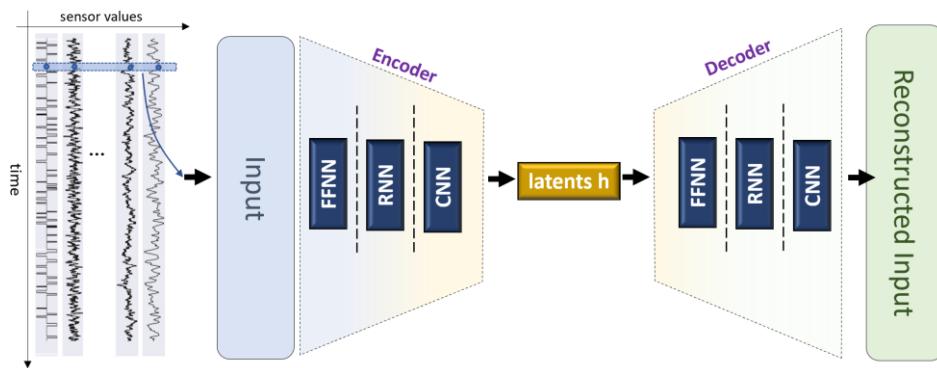


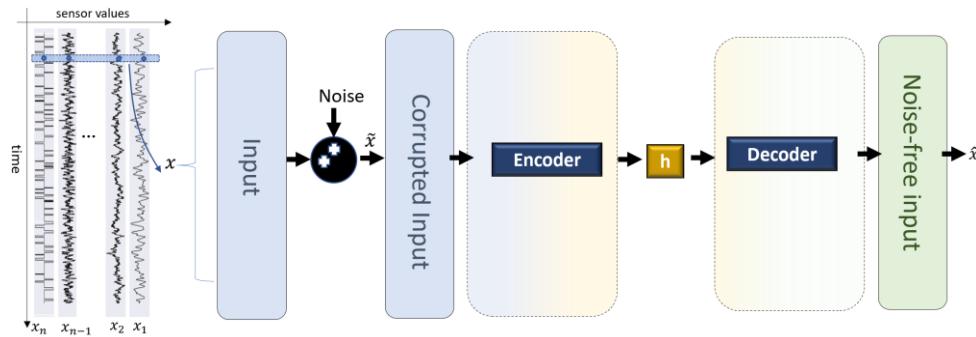
Figure 16.1: Autoencoder architecture

The symmetrical and sandwich nature of the deep AE architecture should be apparent wherein the sizes of the layers first decrease and then increase. Care must be taken though to not use too many hidden layers; otherwise, the network will overfit and may simply learn the identity mapping from x to \hat{x} ! Moreover, in the previous figure, you will notice that AE architecture is divided into an encoder part and a decoder part. An encoder projects or codifies an input sample x to lower dimensional feature h . The decoder maps the feature vector back to the input space. The encoder-decoder form makes the AR architecture very flexible. Once an AE has been trained, one can use the encoder as a standalone network to obtain the latent variables. Moreover, you are not limited to using only FFNN in the encoders and decoders. RNNs and CNNs are also frequently employed. RNN-based AE is used as a nonlinear counterpart of dynamic PCA.



Vanilla AE vs Denoising AE

The form of autoencoder we saw in Figure 16.1 is the conventional or vanilla form wherein the network is forced to find patterns in data by constraining the size of coding/latent variable (m) to be less than the size of input variable (n). This is also called an undercomplete autoencoder. An alternative way of forcing an autoencoder to learn only the systematic variation in data is by corrupting input data by adding synthetic noise and then training the network to reconstruct the uncorrupted input. Such autoencoders are called denoising autoencoders and its representative architecture is shown below. Note that we did not explicitly represent encoder having number of neurons in hidden layer less than the number of input variables. Denoising AE allow having $m \geq n$.



Dimensionality reduction via autoencoders

To see autoencoders in action, let's apply it for the dimensionality reduction of a simulated dataset from a fluid catalytic cracking unit (FCCU¹⁰³) shown in Figure 16.2. FCCUs are critical units in modern oil refineries and convert heavy hydrocarbons into lighter and valuable products such as LPG, gasoline, etc. As shown, the FCCU operation involves catalytic reaction, catalyst regeneration, and distillation. A total of 46 signals are made available as outputs (recorded every minute). Data has been provided in 7 CSV files. Each file contains data from one simulation. One of the CSV files contain NOC data over a period of 7 days with varying feed flow. Five faults have been simulated one at a time in 5 separate simulations. We will work with the 7 days of NOC data.

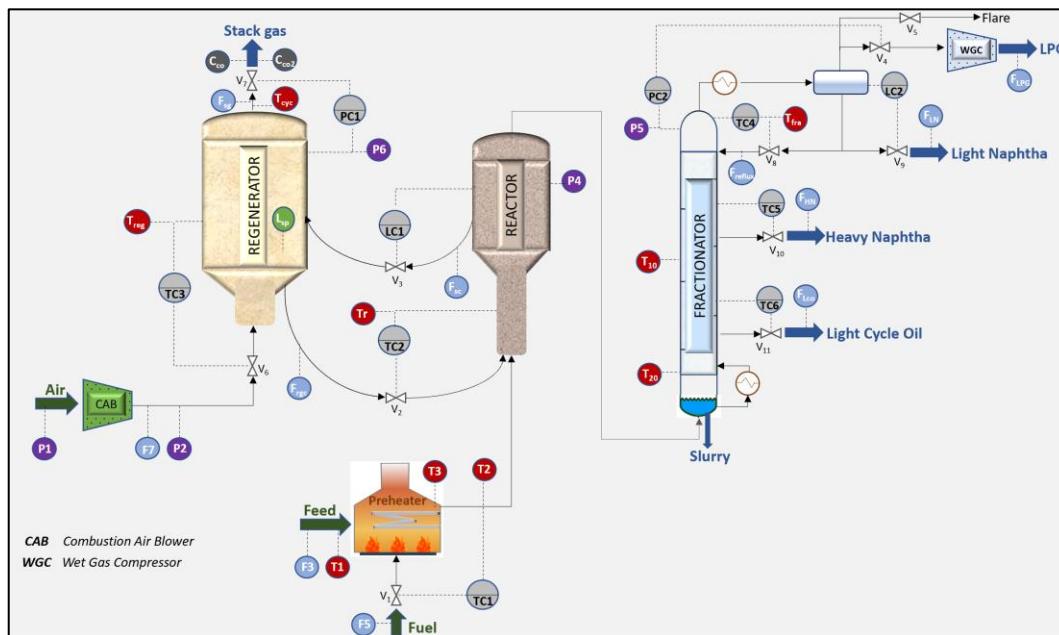


Figure 16.2: Fluid catalytic cracking unit with available measurements

We know that most of the variability in the data is driven from the variations in the feed flow and therefore, we will attempt to generate a 1D latent space ($m=1$).

```
# import required packages
import numpy as np, pandas as pd, matplotlib.pyplot as plt
import tensorflow
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

¹⁰³ Details on the system and datasets available are provided in detail at <https://mlforpse.com/fccu-dataset/>.

```

from tensorflow.keras.layers import Input, Dense
from tensorflow.keras import regularizers
from tensorflow.keras.models import Model

# read data, split into fitting and validation datasets, and scale
X_train = pd.read_csv('NOC_varyingFeedFlow_outputs.csv', header=None).values
X_train = X_train[:,1:] # first column contains timestamps
X_fit, X_val, _, _ = train_test_split(X_train, X_train, test_size=0.2, random_state=10)

scaler = StandardScaler()
X_fit_scaled, X_val_scaled = scaler.fit_transform(X_fit), scaler.transform(X_val)
X_train_scaled = scaler.transform(X_train)

# define and compile model
input_layer = Input(shape=(X_fit_scaled.shape[1],)) # input layer
encoded = Dense(1, activation='relu')(input_layer) # encoder layer
decoded = Dense(X_fit_scaled.shape[1], activation='linear')(encoded) # decoder layer

autoencoder = Model(inputs=input_layer, outputs=decoded)
encoder = Model(inputs=input_layer, outputs=encoded)
autoencoder.compile(optimizer='adam', loss='mse')

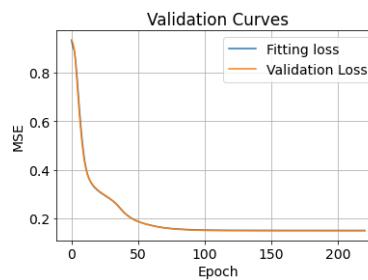
```

You may notice that the way we have defined our ANN model is slightly different than that in the previous chapters. This is done to enable us to use the fitted encoder separately later on. Let's now fit our autoencoder.

```

# fit model
from tensorflow.keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='val_loss', patience=10)
history = autoencoder.fit(X_fit_scaled, X_fit_scaled, epochs=300, batch_size=256,
                           validation_data=(X_val_scaled, X_val_scaled), callbacks=es)

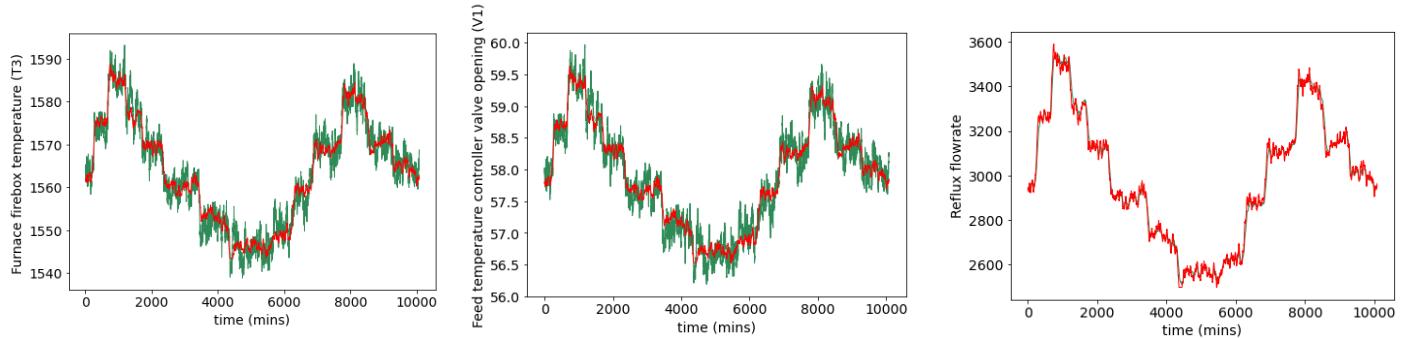
```



The validation curves suggest that the model has converged. Let's check if a 1D latent space is good enough to be able to reconstruct the original data.

```
# predict for overall training dataset
X_train_scaled_pred = autoencoder.predict(X_train_scaled)
X_train_pred = scaler.inverse_transform(X_train_scaled_pred)

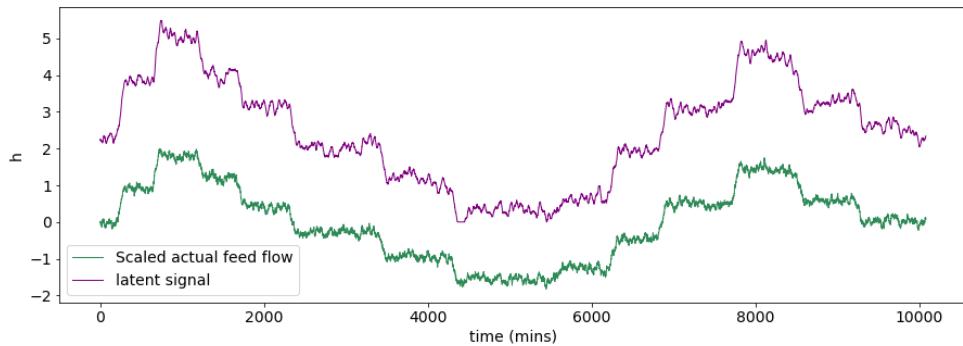
var = 7
plt.figure(), plt.plot(X_train[:,var], 'seagreen', linewidth=1), plt.plot(X_train_pred[:,var],'red')
plt.xlabel('time (mins)'), plt.ylabel('Furnace firebox temperature (T3) ')
```



The reconstructed data follows the systematic variations in original data pretty well. A look into the latent signal can help explain these results. The plot below shows that the latent variable is primarily an encoding of the feed flow. The encoded feed flow is then used to reconstruct the rest of the variables by the decoder.

```
# predict latents
h_train = encoder.predict(X_train_scaled)

plt.figure(figsize=[15,5]), plt.plot(X_train_scaled[:,0],'seagreen', label='Scaled actual feed flow')
plt.plot(h_train, 'purple', label='latent signal')
plt.xlabel('time (mins)'), plt.ylabel('h'), plt.legend()
```

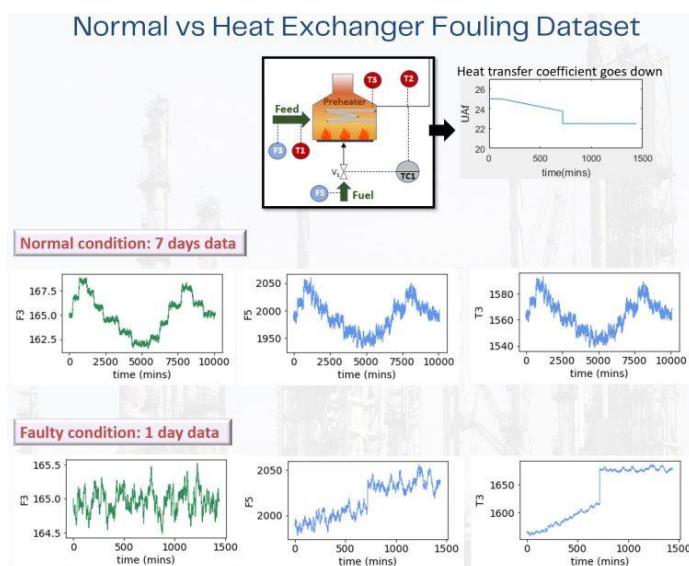




In the above example, the dimension of the feature space was known beforehand. However, in general, this becomes a hyperparameter that needs to be optimized along with other hyperparameters such as the number of hidden layers, the learning rate, batch-size, etc. A generic guidance is that the compression of neurons between successive layers (in encoder) should not be very steep. A compression ratio of 0.5 is often a good starting value.

16.2 Process Monitoring using Autoencoders: FCCU Case Study

Fault detection and diagnosis via AE proceed along the same way as done for PCA. The reconstruction error and latent variables are computed for each training sample, and monitoring metrics are generated. Let's see how the computations are performed. We will re-utilize the FCCU dataset. We will build an autoencoder model using the 7 days of NOC data and the heat exchanger fouling simulation data (*UAf_decrease_outputs.csv*) is used as a test dataset. Let's take a quick look at the heat exchanger fouling fault. Reduced heat transfer to the feed leads the controller TC1 to open valve V1 more to increase the fuel flow so as to maintain T2 at the setpoint. Moreover, less heat going into preheating the feed implies that the flue gas temperature (T3) in the furnace increases. The diagram below summarizes the scenario. Variables F5 and T3 have been shown here and drifts in their values are evident; however, the 'faulty values' are not very far away from the normal values obtained under fault-free operations with varying feed flow.



Our objective is to use AE model to flag faulty samples by using only NOC samples for model training.

Fault detection indices

For a data sample x , statistics Q and T^2 are computed in the familiar way.

$$\begin{aligned} Q &= e^T e \\ &= (x - \hat{x})^T (x - \hat{x}) \end{aligned}$$

$$T = h^T \Lambda^{-1} h$$

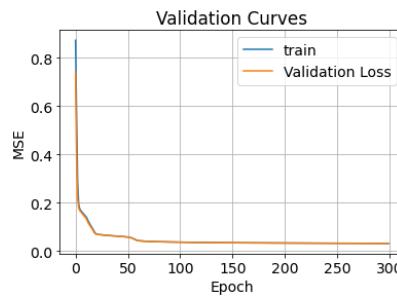
↑ Covariance matrix of the latent variables

Let's generate the monitoring charts for the training samples. A stacked AE model is used for building the fault detection solution.

```
# define and compile model
input_layer = Input(shape=(X_fit_scaled.shape[1],))
encoded = Dense(23, kernel_regularizer=regularizers.L1(0.0001), activation='relu',
                kernel_initializer='he_normal')(input_layer)
encoded = Dense(6, kernel_regularizer=regularizers.L1(0.0001), activation='relu',
                kernel_initializer='he_normal')(encoded)
decoded = Dense(23, kernel_regularizer=regularizers.L1(0.0001), activation='relu',
                kernel_initializer='he_normal')(encoded)
decoded = Dense(X_fit_scaled.shape[1], kernel_regularizer=regularizers.L1(0.0001),
                activation='linear')(decoded)

autoencoder = Model(inputs=input_layer, outputs=decoded)
encoder = Model(inputs=input_layer, outputs=encoded)
autoencoder.compile(optimizer='adam', loss='mse')

# fit model
from tensorflow.keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='val_loss', patience=10)
history = autoencoder.fit(X_fit_scaled, X_fit_scaled, epochs=300, batch_size=256,
                           validation_data=(X_val_scaled, X_val_scaled), callbacks=es)
```



```
#####
#####
```

Monitoring statistics for training samples

```
#####
#####
```

```
h_train = encoder.predict(X_train_scaled)
```

```
# Q metric for training samples
```

```
error_train = X_train_scaled - X_train_scaled_pred
```

```
Q_train = np.sum(error_train*error_train, axis = 1)
```

```
Q_CL = np.percentile(Q_train, 99)
```

```
# T2 metric for training samples
```

```
h_cov = np.cov(h_train, rowvar=False)
```

```
h_cov_inv = np.linalg.inv(h_cov)
```

```
T2_train = np.zeros((X_train.shape[0],))
```

```
for i in range(X_train.shape[0]):
```

```
    T2_train[i] = np.dot(np.dot(h_train[i,:], h_cov_inv), h_train[i,:].T)
```

```
T2_CL = np.percentile(T2_train, 99)
```

```
# Q_train control chart
```

```
plt.figure(), plt.plot(Q_train, color='black')
```

```
plt.plot([1,len(Q_train)],[Q_CL,Q_CL], linestyle='--',color='red')
```

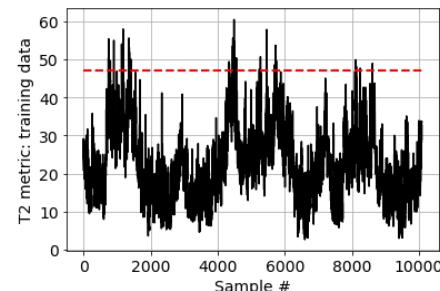
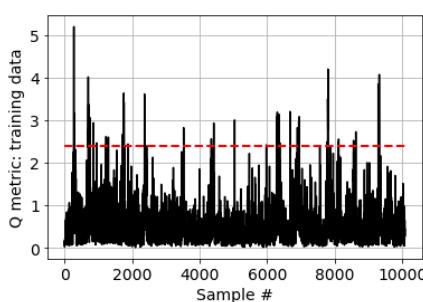
```
plt.xlabel('Sample #'), plt.ylabel('Q metric: training data')
```

```
# T2_train control chart
```

```
plt.figure(), plt.plot(T2_train, color='black')
```

```
plt.plot([1,len(T2_train)],[T2_CL,T2_CL], linestyle='--',color='red')
```

```
plt.xlabel('Sample #'), plt.ylabel('T2 metric: training data')
```



Fault detection for test data

Let's now generate the monitoring charts for the heat exchanger fouling dataset.

```

# read test data, scale, and reconstruct
X_test = pd.read_csv('UAf_decrease_outputs.csv', header=None).values
X_test = X_test[:,1:]

X_test_scaled = scaler.transform(X_test) # using scaling parameters from training data
X_test_scaled_pred = autoencoder.predict(X_test_scaled)
X_test_pred = scaler.inverse_transform(X_test_scaled_pred)

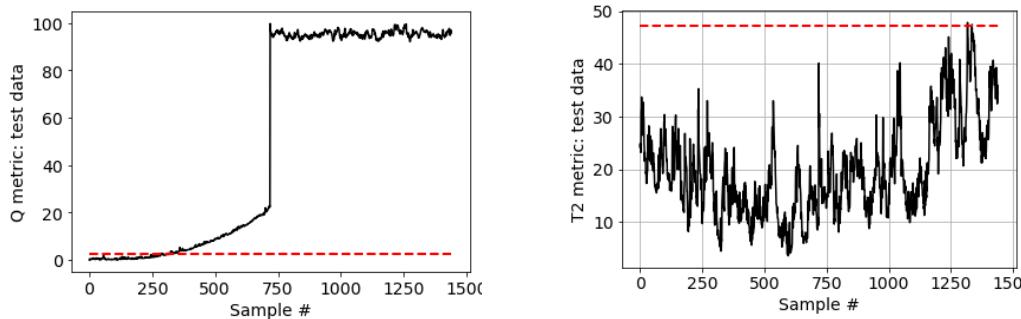
# Q metric for test samples
error_test = X_test_scaled - X_test_scaled_pred
Q_test = np.sum(error_test*error_test, axis = 1)

# T2 metric for test samples
T2_test = np.zeros((X_test.shape[0],))
for i in range(X_test.shape[0]):
    T2_test[i] = np.dot(np.dot(h_test[i,:], h_cov_inv), h_test[i,:].T)

# Q_test control chart
plt.figure(), plt.plot(Q_test, color='black')
plt.plot([1,len(Q_test)],[Q_CL,Q_CL], linestyle='--',color='red')
plt.xlabel('Sample #'), plt.ylabel('Q metric: test data')

# T2_test control chart
plt.figure(), plt.plot(T2_test, color='black')
plt.plot([1,len(h_test)],[T2_CL,T2_CL], linestyle='--',color='red')
plt.xlabel('Sample #'), plt.ylabel('T2 metric: test data')

```



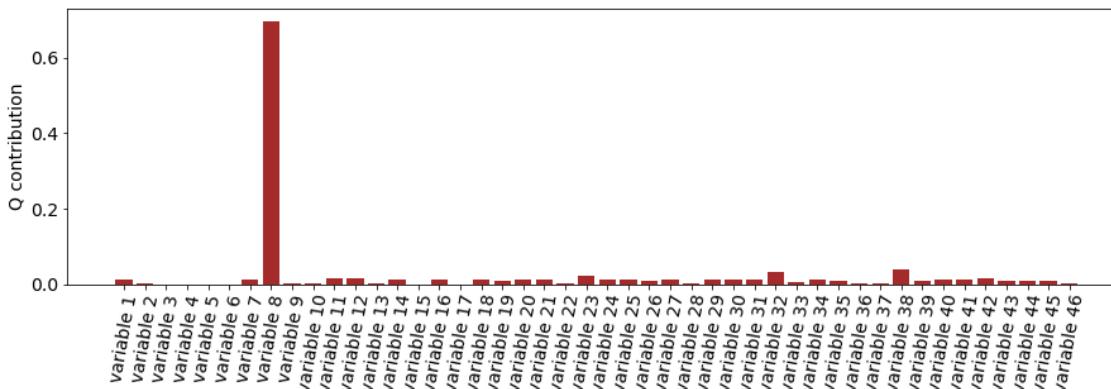
Above monitoring charts indicate good performance of the model: no false alerts before the onset of fault and fault flagged within 2 hours after fault onset. You are encouraged to build monitoring charts using the AE model from Section 16.1 that had 1-D latent variable; you will notice unsatisfactory model performance due to large delay in fault detection.

Fault diagnosis via contribution plot

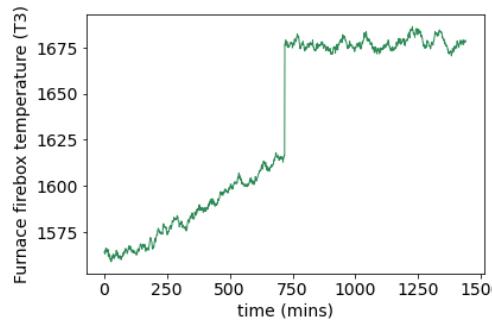
AE-based Fault diagnosis is commonly performed via contribution analysis¹⁰⁴ for Q statistic using the same methodology that we saw in Chapter 7.

```
# Q contribution
sample = 250
error_test_sample = error_test[sample-1,]
Q_contri = error_test_sample*error_test_sample # vector of contributions

plt.figure(figsize=[15,4]), plt.bar(['variable ' + str((i+1)) for i in range(len(Q_contri))], Q_contri)
plt.xticks(rotation = 80), plt.ylabel('Q contribution')
```



```
# plot top diagnosed variables
plt.figure(figsize=[6,4]), plt.plot(X_test[:,7],'cornflowerblue')
plt.xlabel('time (mins)'), plt.ylabel('T3')
```



The contribution plot has correctly flagged the furnace firebox temperature as the faulty variable most responsible to the deviation of the test sample from the NOC behavior.

¹⁰⁴ Due to nonlinear transformation involved in generating AE-based latent variables, the contribution analysis for T^2 statistic is not as convenient as it was for PCA.

16.3 Self-Organizing Maps: An Introduction

Self-organizing maps (SOMs)¹⁰⁵ are unsupervised neural networks that aim to map a high-dimensional dataset onto a 2D grid of neurons as shown in Figure 16.3. The mapping is done in such a way that the topology of data is preserved, i.e., data samples close to each other in the original input space are mapped to nearby neurons on the SOM grid. Accordingly, SOMs render themselves very useful for visualization and clustering of complex high-dimensional datasets. SOMs are named as such as no supervision/guidance is needed to determine the mapping; this makes SOM an excellent EDA tool. Additionally, like autoencoders, no restriction is imposed on the input dataset regarding the data distribution, linearity, and independence among variables, etc.

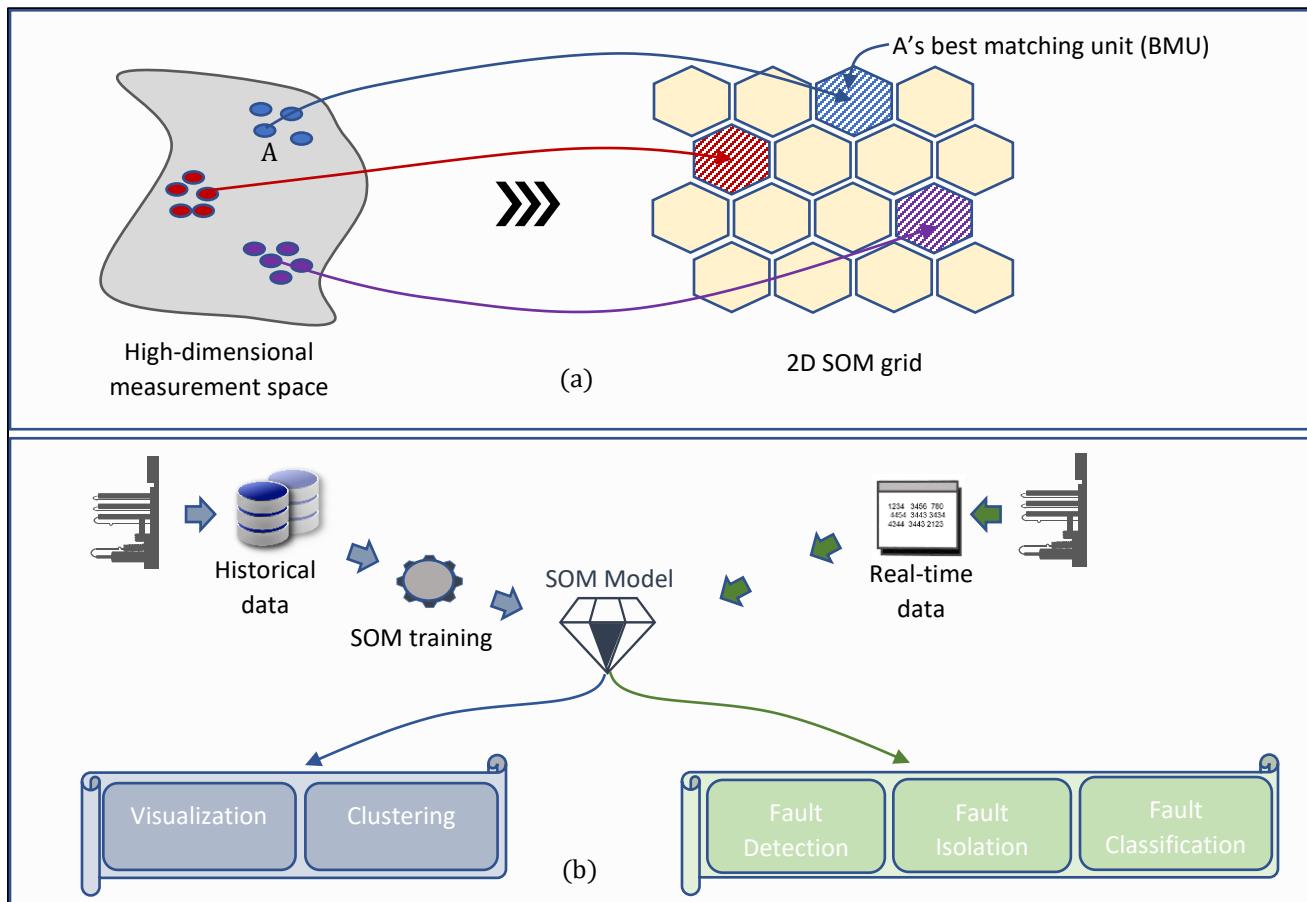


Figure 16.3: (a) Concept and (b) uses of self-organizing maps

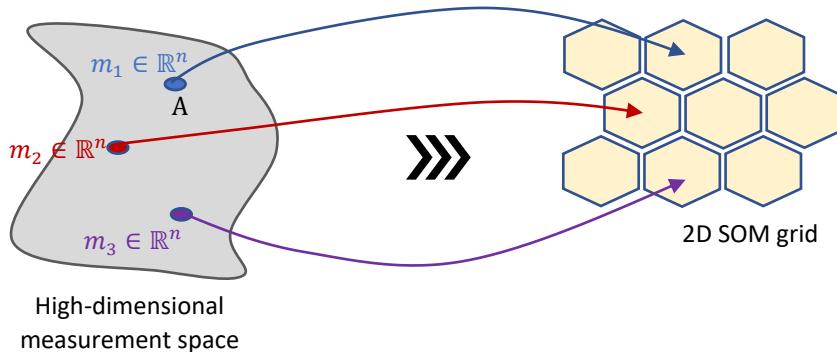
The neuron that a data sample gets mapped to is called the sample's BMU (best matching unit). During training, multiple data samples may get assigned the same BMU. In a way, the BMUs discretize the input space during model training into several local sub-regions and the samples lying in a sub-region are mapped to the same BMU. For a test data sample, if there

¹⁰⁵ SOM is also called as Kohonen map in honor of Prof. Teuvo Kohonen who introduced SOM.

is an unusually large distance between the sample and the sub-region represented by the BMU, then an anomaly alert may be raised. Seems interesting, right? To understand how exactly this is achieved, let's delve into the mathematical details of SOM training.

Mathematical background

A SOM is not fitted in the same manner as a traditional ANN fitting, i.e., the back-propagation algorithm is not employed. To understand the model fitting procedure, first consider the end result of fitting. Let the fitting dataset consists of N samples, where each sample is a n -dimensional vector. After model fitting, each SOM node (or neuron) is assigned a n -dimensional reference vector (or weight vector), say $m_j \in \mathbb{R}^n$ for the j^{th} node, as shown in the illustration below. In a way, the j^{th} node represents the local region around the vector m_j in the input space. Once the reference vectors have been generated, a sample's BMU can be found readily. To see how this is done, let's go through each step of SOM model fitting.



➤ Normalize input dataset and initialize reference vectors

Let there be J neurons in the SOM grid. Each reference vector, $m_j \forall j \in [1, J]$, is assigned randomly or alternatively, assigned a random vector from the dataset. The iteration index is set to $t = 1$.

➤ Sample and assign BMU

A sample x_i is randomly selected from the fitting dataset. The neuron whose reference vector has the smallest (Euclidean¹⁰⁶) distance from x_i is defined as x_i 's BMU, i.e.,

$$b_i = \arg \min_j \|x_i(t) - m_j(t)\| \quad \forall j \in [1, J]$$

\nwarrow
 x_i 's BMU

¹⁰⁶ Other metrics of distance can also be used

➤ *Update reference vectors*

The following formula is used to update $m_j \forall j$

$$m_j(t+1) = m_j(t) + \alpha(t) h_{b_i j} \|x_i(t) - m_j(t)\| \quad \forall j \in [1, J]$$

 The term $h_{b_i j}$ is the neighborhood function centered on BMU b_i and ensures that only the BMU b_i and its neighboring neurons on the SOM grid are moved closer to the sample x_i . It is defined as follows

Location of BMU on SOM grid *Location of j^{th} neuron*

$$h_{b_i j} = \exp\left(\frac{\|r_{b_i} - r_j\|}{2\sigma^2(t)}\right) \quad \forall j \in [1, J]$$

neighborhood width

 $\alpha(t)$ is the learning rate which controls the rate at which the weight vectors get updated

 $\alpha(t)$ and $\sigma(t)$ are assigned large values initially and are decreased monotonically with iterations.

➤ *Increment iteration ($t \rightarrow t + 1$) and go to step 2 if training has not converged and maximum value of t has not been reached.*

Evaluating SOM fit

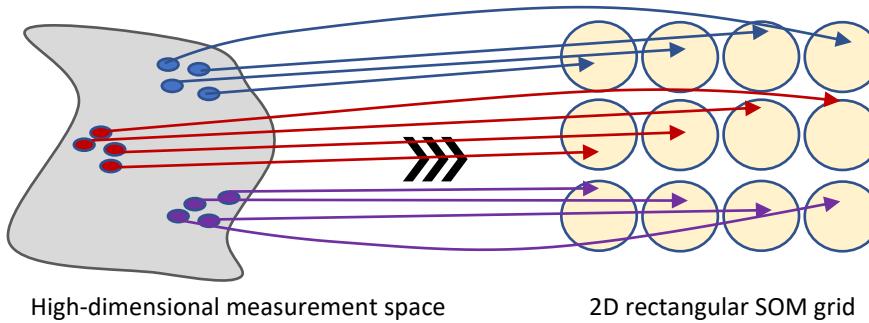
The final map quality depends heavily on the initialization of weight vectors, size of grid, the learning rate, etc. Unfortunately, for high-dimensional datasets, it is difficult to determine exactly if SOM has managed to generate a good map. A metric called quantization error is commonly used which, for a sample x_i , is defined as

$$E_{Q,i} = \|x_i - m_{b_i}\| \quad \text{eq. 1}$$



$E_{Q,i}$ high $\Rightarrow x_i$ has large difference from its mapped BMU b_i

The average quantization error over all the fitting samples can provide a measure of goodness of fit but it can be misleading as illustrated below



Number of neurons are same as number of fitting samples



$m_{b_i} = x_i$, i.e., reference vector of BMU of x_i is x_i itself



The quantization error is 0 here, but it's obvious that the topology has not been preserved

Therefore, different metrics are used to judge the fitted map quality. A couple of commonly used metrics are described below.

Quantization error

As described before, the overall quantization error (E_Q) for fitting samples is given as

$$E_Q = \frac{\sum_i \|x_i - m_{b_i}\|^2}{N}$$

Although E_Q does not give a good measure of topological preservation, it can indicate how good the map fits to data. E_Q can also be used to infer overfitting and underfitting. Adding more neurons lead to lower E_Q and therefore very low E_Q may imply overfitted model. High values of E_Q can indicate insufficient number of neurons or un converged network learning.

Topographic error

This metric measures how well the original shape of data has been preserved on SOM grid. It is computed by simply counting the number of samples for which the first and second BMUs are not adjacent neurons on the SOM grid.

$$E_T = \frac{1}{N} \sum_{i=1}^N u(x_i)$$

$u(x_i) = \begin{cases} 1, & \text{if 1st and 2nd BMUs are non-adjacent} \\ 0, & \text{otherwise} \end{cases}$

Size of SOM grid

The shape and size of SOM grid can strongly influence the quality of map. The hexagonal grid is very commonly employed and a generic guidance is to use $5\sqrt{N}$ neurons in the grid.

16.4 Visualization of Semiconductor Dataset via SOM

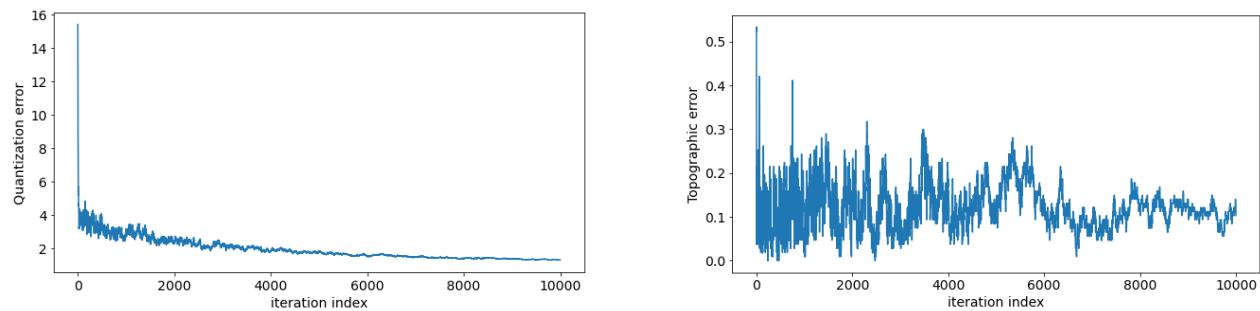
To see SOM in action, let's apply it to our semiconductor metal-etch dataset. We will again work with the first three PC scores of the original unfolded data as done in Chapters 11 and 14 for better visualization of SOM's input dataset. Let's see if SOM can indicate presence of 3 clusters. We will employ *Minisom*¹⁰⁷ package for building our SOM model.

```
# fit SOM model
from minisom import MiniSom

N = score_train.shape[0]
N_neurons = 5*np.sqrt(N)

som = MiniSom(np.floor(np.sqrt(N_neurons)).astype(int), np.floor(np.sqrt(N_neurons)).astype(int),
              score_train.shape[1], sigma=1.5, learning_rate=.7, activation_distance='euclidean',
              topology='hexagonal', neighborhood_function='gaussian', random_seed=10)
som.train(score_train, num_iteration=10000, verbose=True)
```

The plots of the evolution of quantization and topographic errors during training are shown below. It is apparent that both the errors show sharp decrease in the first few iterations. Overall 10000 seem to be a good value for the number of iterations.



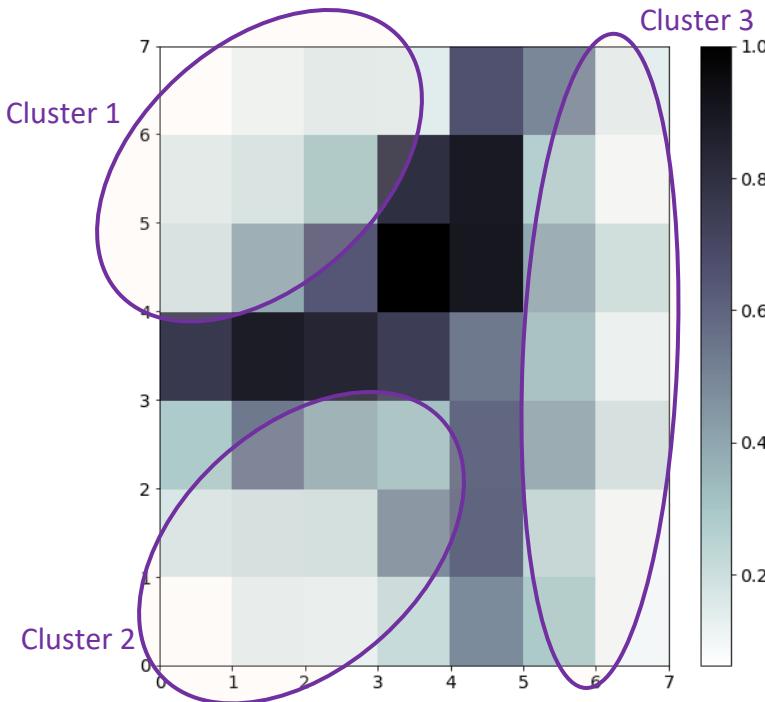
¹⁰⁷ <https://github.com/JustGlowing/minisom>

The most common approach to visualize a SOM and look for clusters is to generate a U-matrix¹⁰⁸ (unified distance matrix) as shown below for our fitted SOM model. Here, each neuron is colored based on the average distance between the weight vectors of the neuron and its neighbors. A lighter color imply that the neuron and its neighbors are mapped to the same region of the input space. From the shown U-matrix, three clusters are clearly evident.

```
# plot U-matrix
```

```
plt.figure(figsize=(9, 9))
```

```
plt.pcolor(som.distance_map().T, cmap='bone_r') # plotting the distance map as background
plt.colorbar()
```

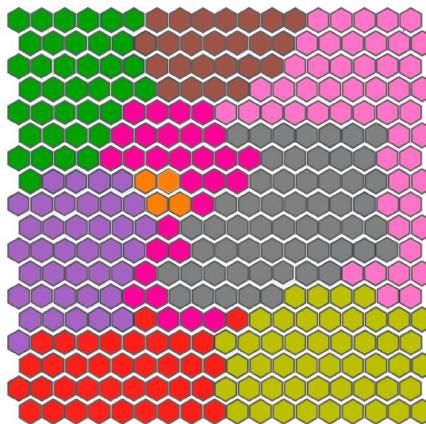


¹⁰⁸ Other common visualization tools include the component planes, frequency maps, class representation maps, etc.

SOM for supervised learning



If training dataset contains fault samples from different classes of faults (including fault-free samples), then one can use class representation map to see where different classes lie on the SOM grid. An example is shown below. Here, the color of a neuron is based on the most frequently occurring class of the samples mapped to the neuron. Such a map also allows SOM model to be used for fault diagnosis wherein the fault status of a test sample can be determined based on the fault label of its BMU in the class representation map.



Class representation map. Each color corresponds to a fault class.

[Map shared under Creative Commons Attribution (CC BY) license (creativecommons.org/licenses/by/4.0/) by Concetti et al. in paper titled 'An Unsupervised Anomaly Detection Based on Self-Organizing Map for the Oil and Gas Sector']

16.5 Process Monitoring using SOM: Semiconductor Case Study

In SOM-based process monitoring, the statistic that is monitored is the quantization error defined in Eq. 1. Large value of this metric indicates that the test sample is away from the normal operation data and is faulty. Let's generate the control chart for the training samples of the metal-etch dataset.

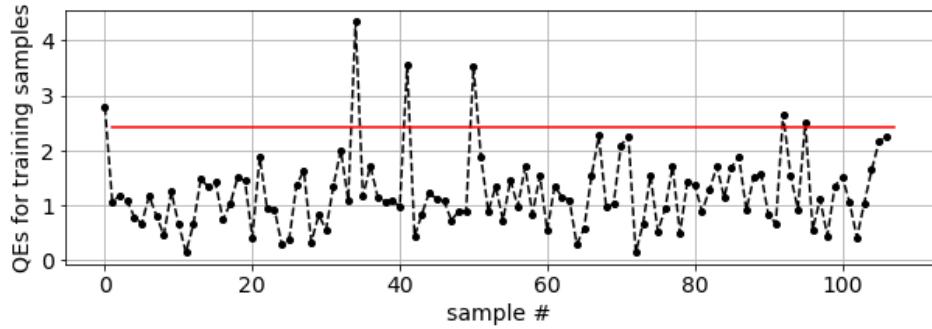
```
# QEs for training samples
```

```
QEs_train = np.linalg.norm(som.quantization(score_train) - score_train, axis=1)
```

```
QE_CL = np.percentile(QEs_train, 95)
```

```
plt.figure(), plt.plot(QEs_train, '--', marker='o', color='black')
```

```
plt.plot([1,len(QEs_train)],[QE_CL, QE_CL], color='red')
plt.xlabel('sample #'), plt.ylabel('QEs for training samples')
```

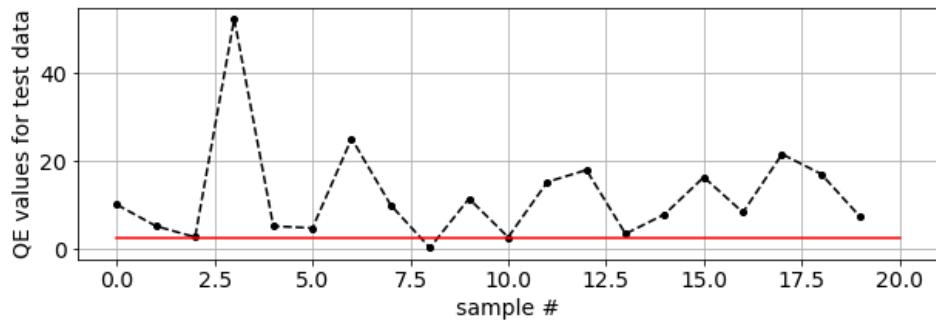


Fault detection for test data

Let's now see if the SOM model can detect the faults in the test samples.

```
# QEs for test samples
score_test = pipe.transform(unfolded_TestdataMatrix)
QEs_test = np.linalg.norm(som.quantization(score_test) - score_test, axis=1)

print('Number of flagged faults (using control chart): ', np.sum(QEs_test > QE_CL))
>>> Number of flagged faults (using control chart): 19
```



Fault diagnosis via contribution plot

Upon the detection of a fault, the culprit variable(s) can be identified by finding the variable(s) contributing the most to the quantization error. E_Q can be broken down component-wise as follows (in the same way as we did for PCA's Q metric)

$$E_{Q,i} = \sum_{j=1}^m \|x_{ij} - m_{b_{ij}}\|^2$$

where x_{ij} is the value of the j^{th} variable for the i^{th} test sample and $m_{b_{ij}}$ is the value of the j^{th} variable in the weight vector of the corresponding BMU.

Summary

In this chapter, we looked at two popular unsupervised neural network models, viz, autoencoders and self-organizing maps. We studied how to employ these models for building process monitoring solutions. Case-studies using industrial-scale systems (fluid catalytic cracking unit from oil refineries and metal-etch semiconductor process) were shown to illustrate the step-by-step procedures.

Part 6

Vibration-based Condition Monitoring

Chapter 17

Vibration-based Condition Monitoring: Signal Processing and Feature Extraction

Rotating machinery, which includes motors, compressors, pumps, turbines, fans, etc., form the backbone of industrial operations. Unsurprisingly, a large fraction of operation downtime can be attributed to the failures of these machines. Over the last decade, the process industry has adopted predictive maintenance as the means to proactively handle these failures and the technique that has largely become synonymous with predictive maintenance is vibration-based condition monitoring (VCM). All rotating machines exhibit vibratory motions and different kind of faults produce characteristic vibratory signatures. This makes VCM a reliable and effective tool for health management of rotating equipment. Considering the importance of VCM in process industry, its different aspects are covered in this part of the book.

Vibrations are usually measured at very high frequency and the large volume of data makes analysis of raw data difficult. Correspondingly, processing vibration data and extracting meaningful features that can provide early signs of failures become very crucial. Traditionally, these features have been analyzed by vibration experts. However, in recent times, several successful applications of ML-based VCM have been reported. All the techniques that we have studied in the previous parts of the book can be used for VCM. While we will look at ML-based VCM in the next chapter, this chapter sets the foundations for VCM and covers vibration data processing and feature extraction.

Over the years, VCM practitioners and researchers have fine-tuned the art of vibration monitoring and have come up with several specialized and advanced techniques. Arguably, it is easy for a beginner to feel ‘lost’ in the world of VCM. The current and the following chapters will help provide some order to this seemingly chaotic world. Specifically, the following topics are covered

- Basics of vibrations
- VCM workflow
- Spectral analysis of vibration signal
- Time domain, frequency domain, and time-frequency domain feature extraction

17.1 Vibration: A Gentle Introduction

Vibrations are simply back and forth motion of machines around their position of rest. All rotating machines (motors, blowers, chillers, compressors, turbines, etc.) exhibit vibratory motion under normal and faulty conditions. Figure 17.1 shows a representative setup for vibration sensing of an industrial machine. The sensors (transducers) convert vibratory motion (of displacement, velocity, or acceleration) into analogue electrical signals which are digitized and stored. The figure below shows how the recorded signal looks like on a time-axis for a machine with gradually degrading condition. The increasing vibration levels indicate underlying machine issues.

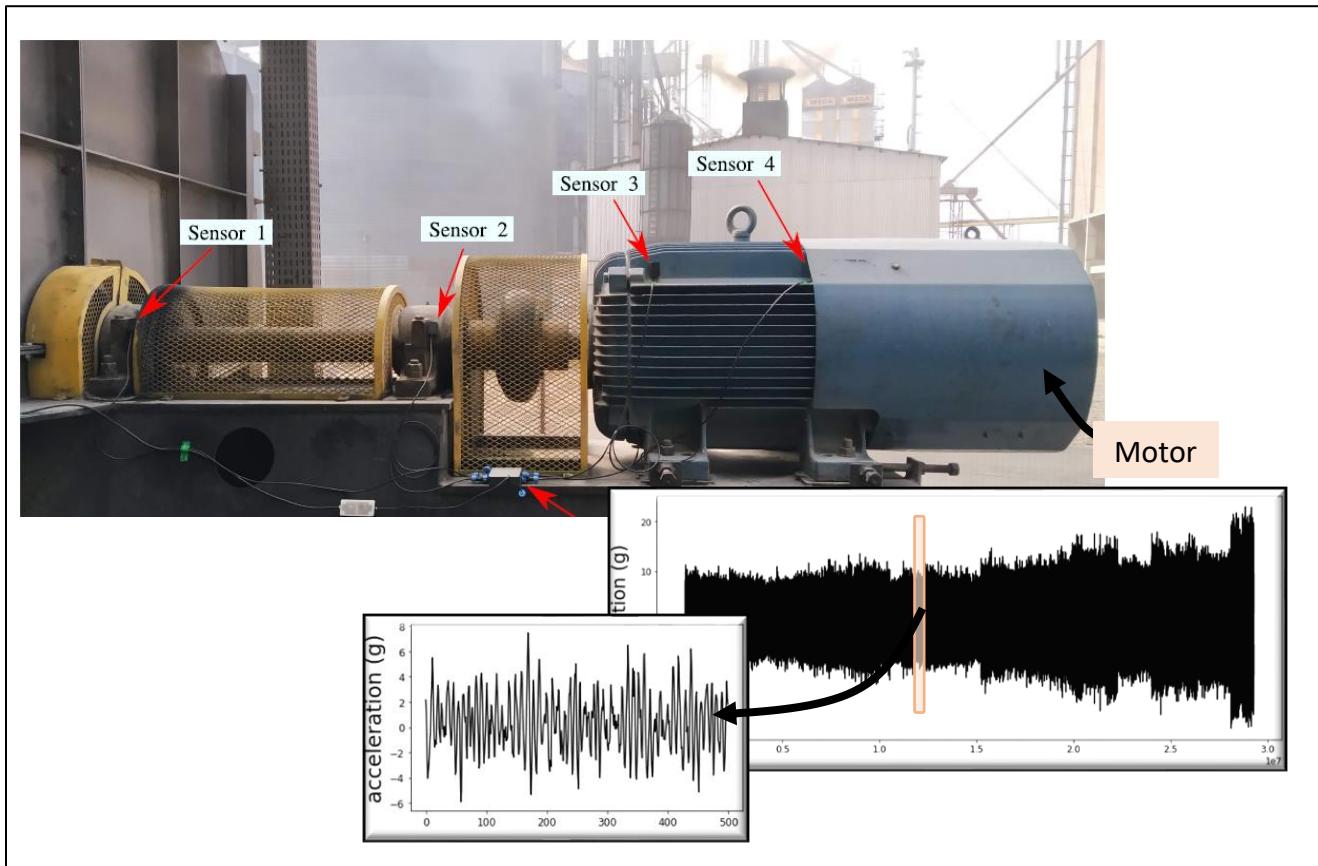


Figure 17.1: Representative vibration monitoring system¹⁰⁹

The components of rotating machines (rotors, bearings, gears) undergo different types of failures due to well-studied causes such as mechanical looseness, misalignment, cracks, etc. These faults produce characteristic vibration patterns. However, it is difficult to 'see' or extract

¹⁰⁹ Romassini et al., A Review on Vibration Monitoring Techniques for Predictive Maintenance of Rotating Machinery. Eng, 2023. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

these specific signatures in the raw time domain vibration signal. Therefore, the standard approach is to convert time domain signal into frequency domain to extract features that provide early signs of failures and aid diagnosis of underlying faults.



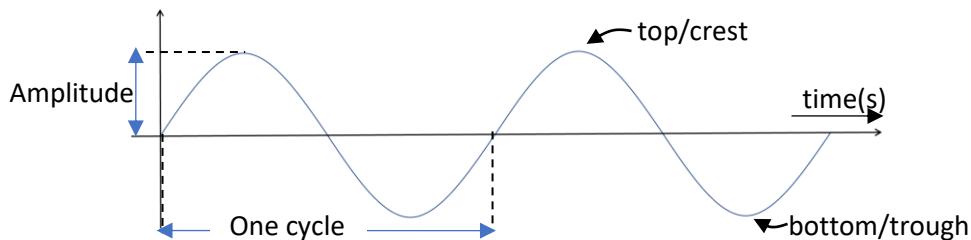
In general, while vibration levels indicate the severity of faults, the dominant frequency components of the vibration signal can indicate the source of faults.

The sequence of steps that are undertaken to convert raw vibration signal into useful features and to deploy ML models is presented in the next section. The focus of this chapter is to understand how to extract these features that can eventually be used as inputs to ML models.

Basic vibration terminology



The time domain plot of vibration data is called waveform. Let's look at a simple sinusoidal waveform to understand some terms that are used to characterize vibration.



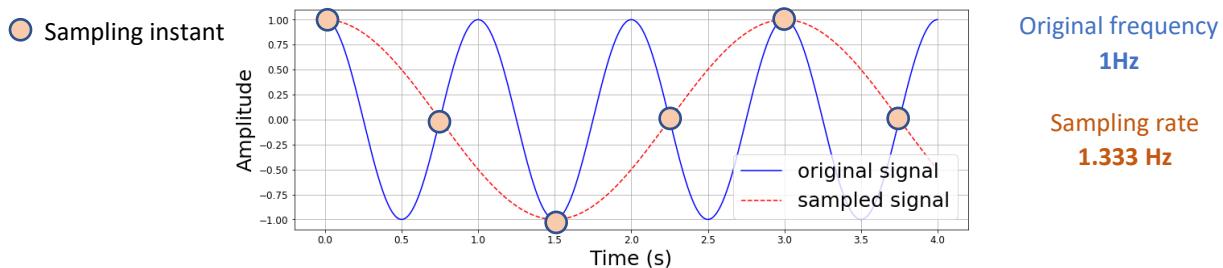
Frequency (f), measured in cycles/s or Hz (Hertz), denotes the number of cycles completed per unit time. The time taken to complete one cycle is referred to as time-period (T). T and f are inversely related

$$f = \frac{1}{T}$$

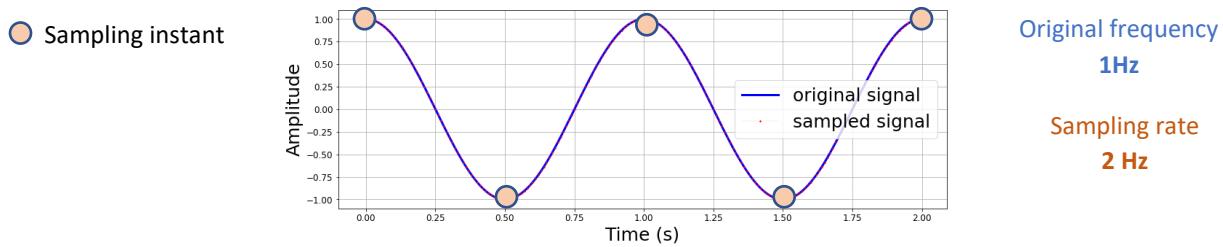
The height of the crest above the zero/reference line is called amplitude. Waveforms from real equipment are more complex than a simple sinusoid. However, you will see later that waveforms can be broken down into several sinusoidal components.

Vibration signal acquisition

The most commonly employed sensor is accelerometer which measures vibration acceleration (usually measured in g). The other sensor options are proximity sensors and velocity transducers which record vibration displacement and velocity, respectively. The output from the vibration sensor is a continuous analog signal which is digitized for digital storage in a computer. Therefore, a pertinent aspect during vibration signal acquisition relates to the sampling frequency (f_s) which is the frequency at which the vibration signal is sampled. For example, let $f_s = 100$ samples/s or 100 Hz; this implies that data is collected at equal intervals of 0.01 seconds. Judicious selection of f_s is very critical. To see why, consider the illustration below.



In the above illustration, the sampled signal does not resemble the original signal! There is a theorem called Nyquist sampling theorem which states that for a sampling frequency f_s , original signals will be collected correctly only up to frequency $f_s/2$. Correspondingly, the frequency $f_s/2$ is known as Nyquist frequency (f_q). For the above illustration, to be able to correctly capture 1 Hz vibration, f_s must be 2 Hz (2 samples per second) which as shown below does work nicely.



The phenomenon of an original high frequency signal appearing as a low frequency signal due to undersampling is called aliasing. This is undesirable and therefore, during vibration signal acquisition, anti-aliasing filters are frequently employed to filter out components with frequencies higher than f_q .

17.2 Vibration-based Condition Monitoring: Workflow

We have so far talked about the different aspects of VCM in only bits and pieces. Figure 17.2 summarizes the commonly employed steps involved in VCM which are briefly described below.

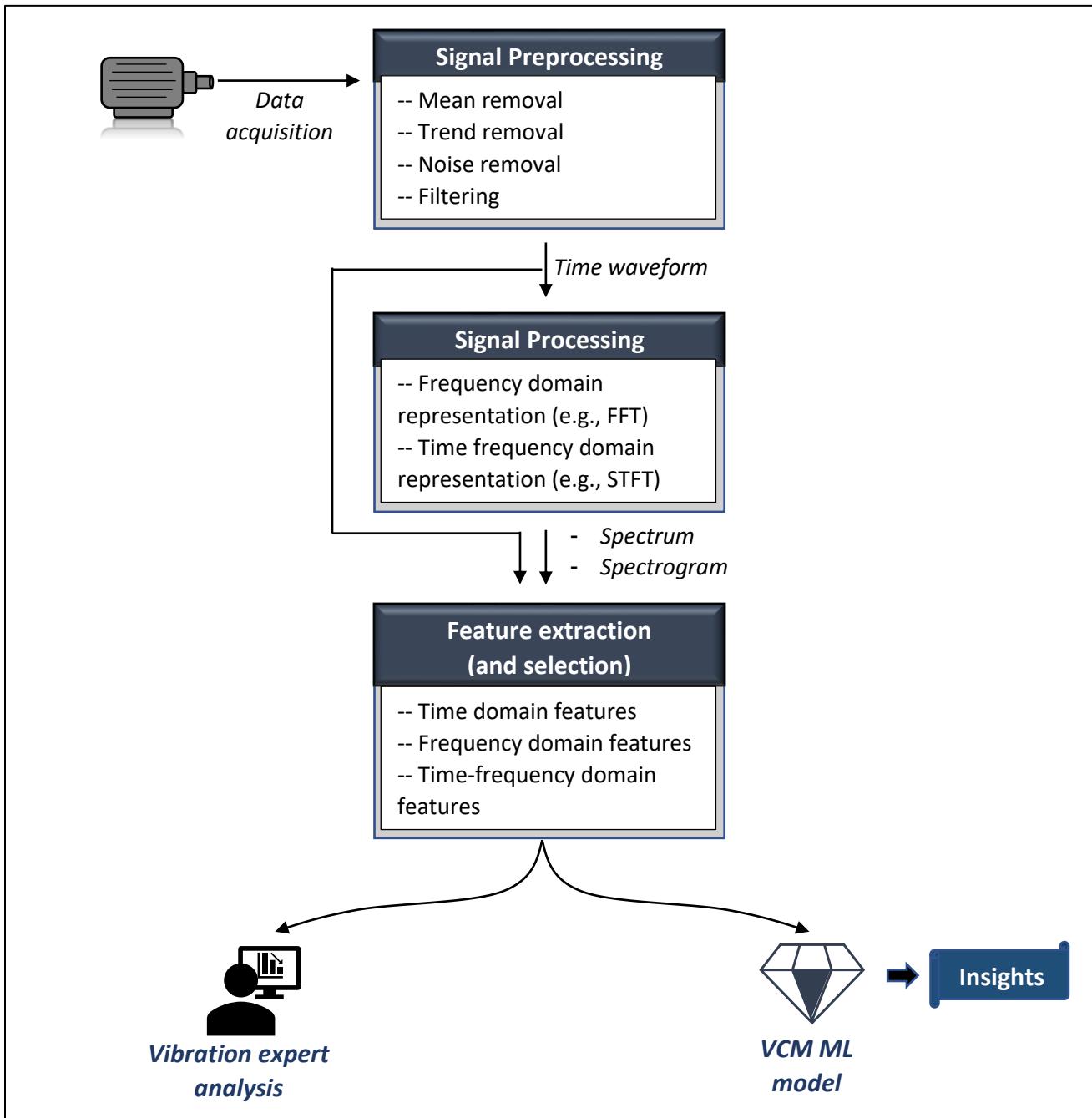
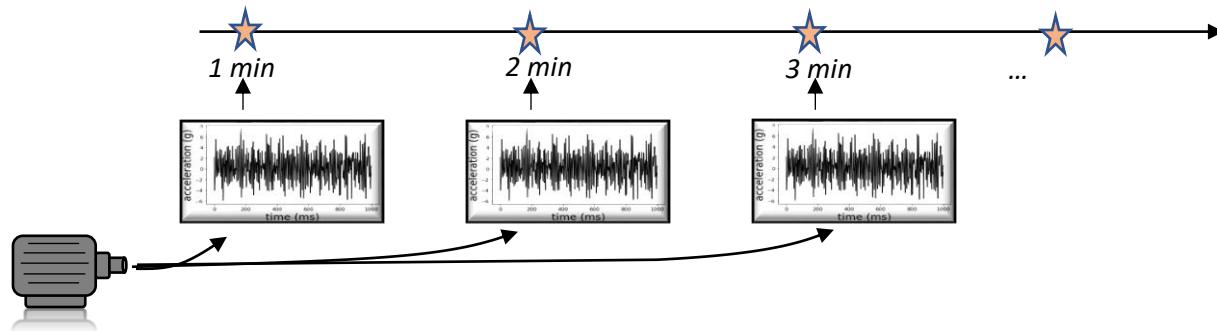


Figure 17.2: Vibration condition monitoring workflow

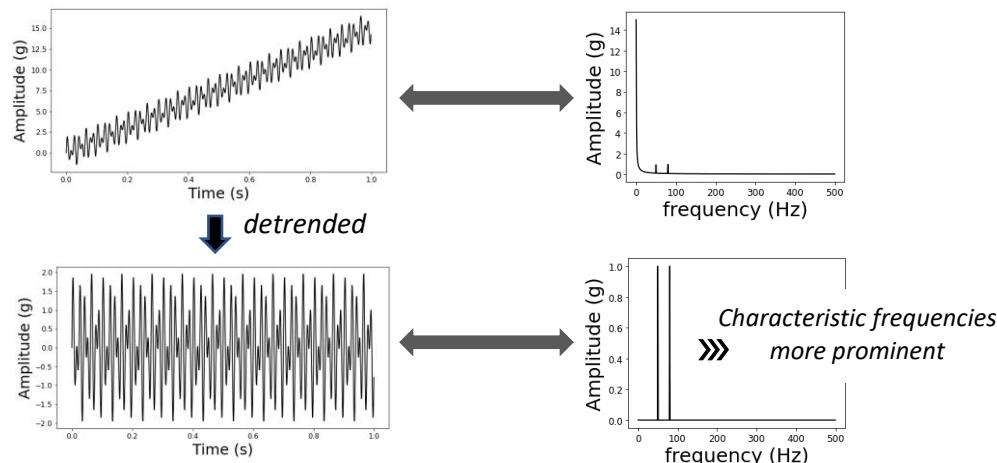
Data acquisition:

As a common practice, the vibration data is analyzed in batches of equal length acquired at regular time intervals as shown in the illustration below. Here, at every minute, 1 second of vibration data is acquired and sent for pre-processing and vibration monitoring. If the sampling frequency is 1024 Hz, then each collected waveform contains 1024 data points.



Signal pre-processing:

Tasks involved in this step attempt to improve the signal-to-noise ratio (SNR) of the acquired vibration signal to prepare it for the subsequent tasks. Unwanted frequency components of the signal due to noise and measurement system errors are removed. During the early stages of failure, the impact on the vibration signal is weak and can be difficult to detect in the absence of adequate signal pre-processing. As an example, the illustration below shows the benefits of trend removal.

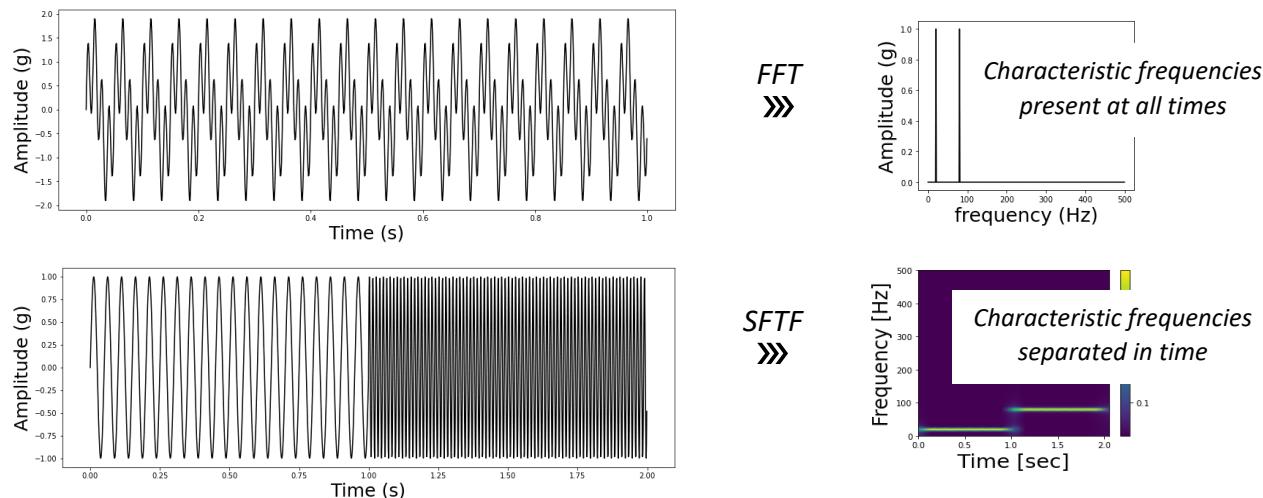


Signal processing:

A quick glimpse of waveform can tell you if a machine is vibrating at abnormally high levels. However, waveforms are notoriously difficult to use to detect failures at incipient stages of faults when fault characteristics are weak. Moreover, increased amplitude levels in vibration waveform does not tell much about the underlying source of failure. To derive more actionable information, spectrum analysis via fast Fourier transform (FFT) is commonly performed to find

the frequency components of the waveform. Presence of significant components at unexpected frequencies indicates faulty conditions and provides clues for fault sources.

A downside of Fourier transform is that it is restricted to stationary signals. It cannot indicate the change of component frequencies over time in a waveform, i.e., a frequency component is assumed to be present throughout the entire duration of a waveform at the same energy level. However, this is not always true, especially in the case of a degrading machine. Therefore, analysis in time-frequency domain is performed to decompose a waveform into its frequency components at different time-instants. The most popular tool for this analysis is short-time Fourier transform (STFT) and a spectrogram is the output from this analysis. We will go into more details on these techniques in the next section.



Feature extraction:

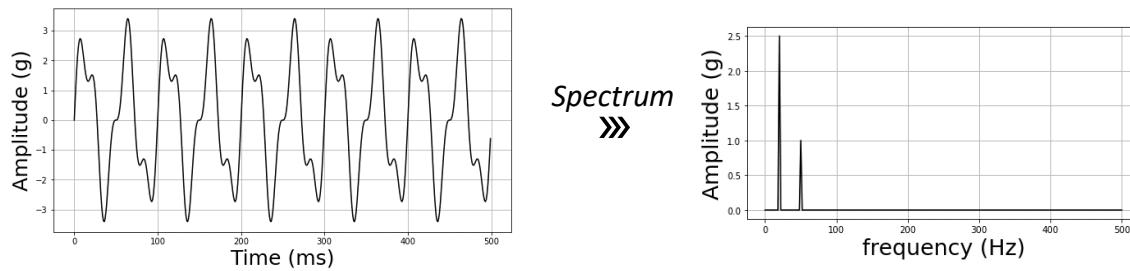
One can directly compare the current waveform, spectrum, and spectrogram with those from NOC periods to detect abnormalities. Alternatively, one can pass the entire waveform and/or spectrum as input to a ML model. However, a more convenient approach is to extract meaningful features that summarize the vibration signals adequately and provide crucial clues regarding the operational state of a machine. Usage of features increases the chances of obtaining effective ML models. The commonly used features include, amongst others, RMS and kurtosis of waveform, frequency center and peak frequency of spectrum, and mean frequency from spectrogram. Once the features have been extracted, they can be used for fault detection and classification. While we will deal with this last step of VCM workflow in the next chapter, let's obtain better familiarization with the vibration signal processing and feature extraction tasks.

17.3 Vibration Signal Processing

As alluded to in the previous section, analyzing the time-domain waveform manually can be very inconvenient and may not provide sufficient leading indicators of incipient faults. A more useful representation of vibration data can be obtained in frequency domain and time-frequency domain.

Frequency domain analysis

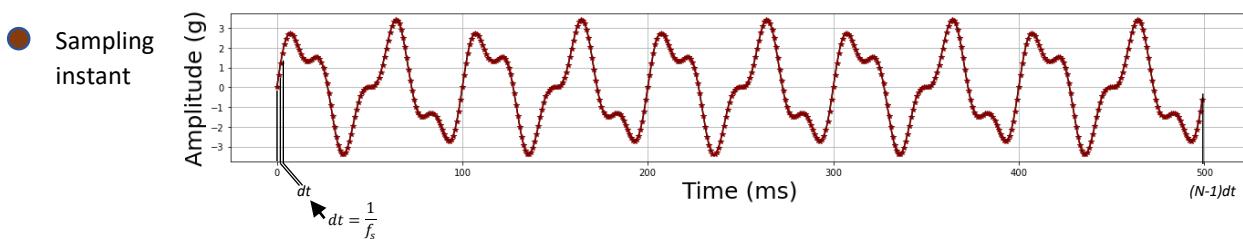
Consider the following time domain and frequency domain vibration data of a machine.



For an untrained eye, the waveform does not provide much useful information about 'how' the machine is vibrating. However, in the frequency domain, one can clearly see two distinct frequency components potentially arising from different rotating components of the machine. Comparison of peak amplitudes and peak frequencies with NOC values can indicate the severity and type of fault, if any exists. This extremely useful plot of frequency versus amplitude is called a spectrum and is commonly obtained via FFT.

Fourier transformation

Frequency domain analysis entails decomposition of a waveform into a sum of several sinusoids of different frequencies. This decomposition is called Fourier transformation. To see how it is calculated, let us define a few terms first. Let the sampling frequency be f_s , the waveform contain N data points, and the frequency resolution, df , be defined as $\frac{f_s}{N}$.



The Fourier transform (FT) is calculated as follows,

$$Y(kdf) = \frac{2}{N} \sum_{i=0}^{N-1} y(t_i) e^{-j\frac{2\pi i k}{N}} \quad k = 0, 1, 2, \dots, \frac{N}{2} - 1$$

FT value at frequency $f_k = kdf$ $y(t_i) = y(t)$ at time $t_i = idt$

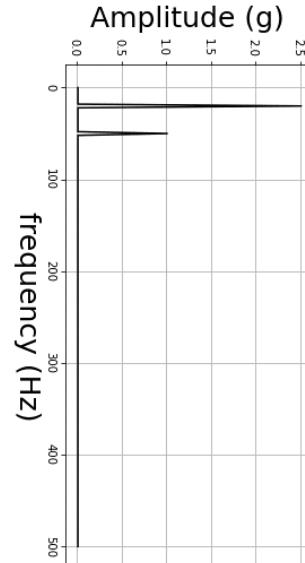
$Y(f_k)$ is a complex number and the amplitude corresponding to f_k that is shown on a spectrum plot is given as the amplitude of $Y(f_k)$, i.e., $|Y(f_k)|$. A spectrum can be very easily generated in Python. Let us do it for our above waveform.

```
# simulate signal
import numpy as np, matplotlib.pyplot as plt
fs = 1000 # 1000 Hz
dt = 1.0/fs
t = np.arange(0,0.5,dt) # sampling instants

y1 = np.sin(2*np.pi*50*t) # 50Hz component
y2 = 2.5*np.sin(2*np.pi*20*t) # 20Hz component
y = y1+y2 # sampled signal

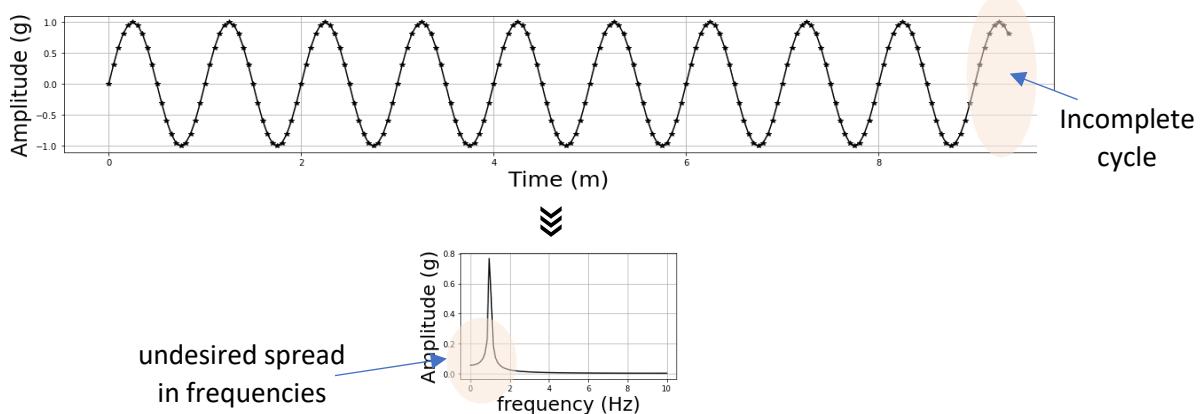
# generate spectrum
from scipy.fft import rfft, rfftfreq
N = len(t)
Y_spectrum = rfft(y)
freq_spectrum = rfftfreq(N, dt)

plt.plot(freq_spectrum, 2.0/N *np.abs(Y_spectrum), 'black')
plt.ylabel('Amplitude (g)'), plt.xlabel('frequency (Hz)')
```

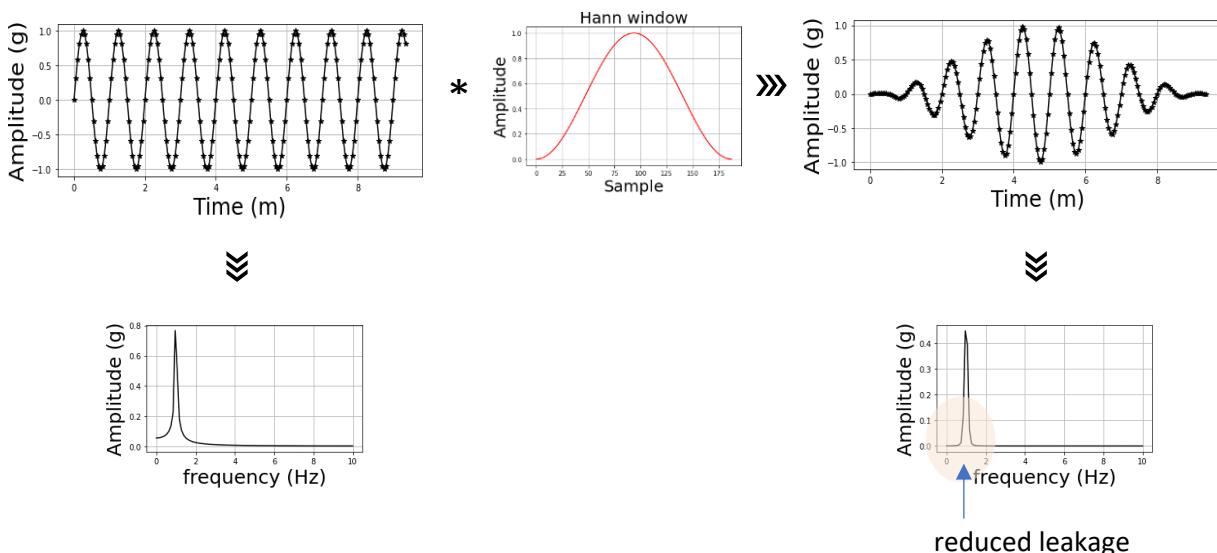


Sources of error during Fourier transformation

There are several sources of error during Fourier transformation which can result in spurious frequencies showing up in your spectrum plot, amplitudes being incorrect, and frequencies incorrectly missing from the spectrum. The errors can arise due to noisy measurements, inadequate frequency resolution, etc. Let's consider a well-known error called 'leakage'. To understand this, assume that a machine is vibrating sinusoidally at 1 Hz. A sequence of length 9.4 seconds is taken for analysis at $f_s = 20$ Hz.

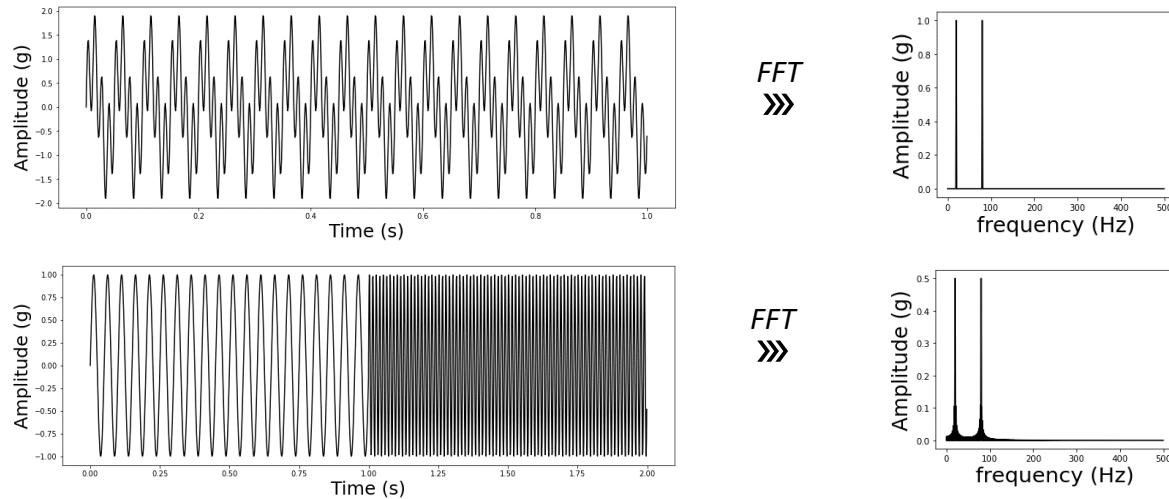


Instead of a sharp spectral line at 1 Hz, we have obtained a range of dominant frequencies. These spurious components result from the non-periodic waveform taken for analysis and the resulting smearing of energy in the frequency domain is called *leakage*. To prevent leakage, the waveform is multiplied with a window of data to force the sampled waveform to be zero at the beginning and the end as shown below. We can see that windowing has significantly reduced the smearing of frequencies. The window shown in the below illustration is called Hann window and is commonly employed for leakage reduction.



Time-frequency domain analysis

Consider the two scenarios below.



It is obvious that the two waveforms are different - two frequency components are present throughout the time in scenario 1 and in scenario 2, the two components are present at different points in time. However, the spectrum for both the scenarios are similar (ignoring the little ripples in the 2nd spectrum and the amplitude differences of peak frequencies). Understandably, the spectrum for scenario 2 can lead to wrong inferences. The solution to handle scenario 2 is to analyze the waveform in both time and frequency domains together and the most popular tool to do this is short-time Fourier transform (STFT) whose procedure is in Figure 17.3¹¹⁰.

Here, you can see that the waveform is divided into overlapped segments, and Fourier transformation is performed, resulting in a 2D matrix, $S_y(f, \tau)$. The STFT is visualized using spectrogram and waterfall plots. A waterfall simply plots each FT spectrum one after the other giving an impression of a waterfall in a 3D plot. A spectrogram is a 2D plot on time-frequency axis with amplitude variations shown using colors. These diagrams help to see how constituent frequencies change within a waveform over time. The code below shows how to obtain time-frequency decomposition of our waveform in scenario 2.

¹¹⁰ The figure is adapted from the open-access article by Kim et al. title ‘Diagnostics 101: A Tutorial for Fault Diagnostics of Rolling Element Bearing Using Envelope Analysis in MATLAB’ and published in *Applied Sciences* (2020). The article is distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).

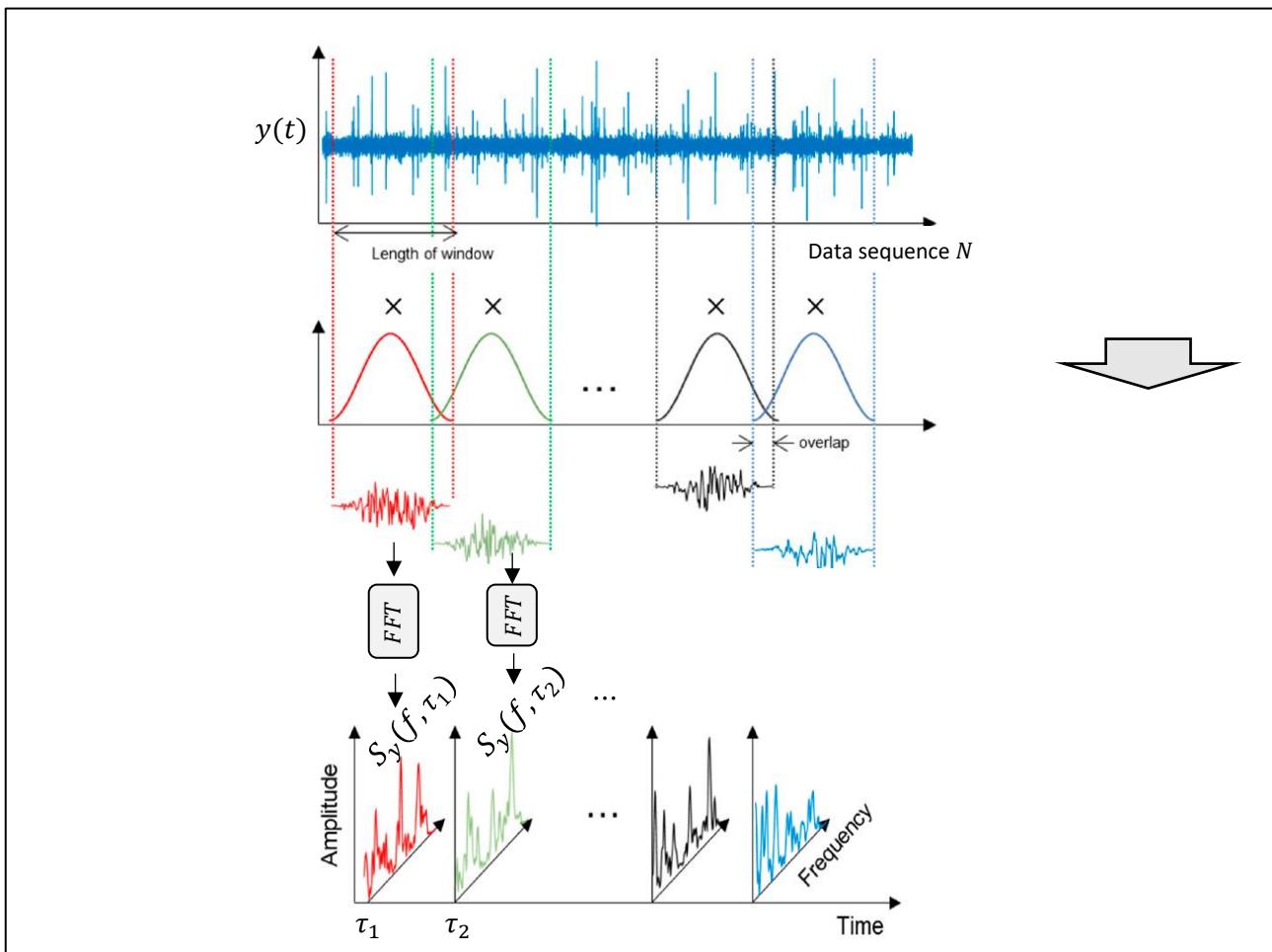


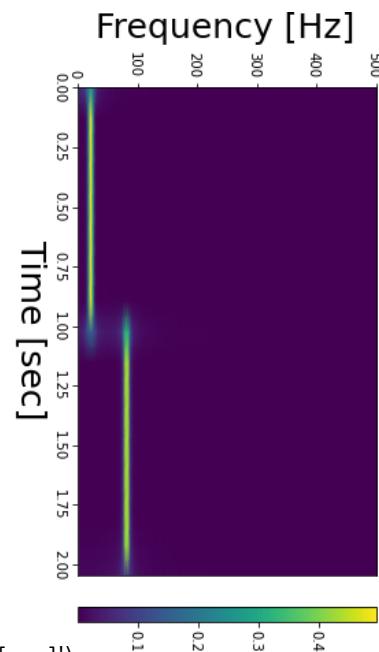
Figure 17.3: STFT procedure

```
# simulate signal
import numpy as np, matplotlib.pyplot as plt
fs = 1000 # 1000 Hz
dt = 1.0/fs
t = np.arange(0,1,dt)

y1 = np.sin(2*np.pi*20*t) # 20Hz component
y2 = np.sin(2*np.pi*80*t) # 80Hz component
y = np.hstack((y1,y2)) # sampled signal
t = np.hstack((t, t[-1]+t))
```

```
# generate spectrogram
from scipy import signal
f, t, Sxx = signal.stft(y, fs)
```

```
plt.pcolormesh(t, f, np.abs(Sxx), shading='gouraud')
plt.colorbar(), plt.ylabel('Frequency [Hz]'), plt.xlabel('Time [sec]')
```



17.4 Feature Extraction from Vibration Signals

By the time you reach this step of your VCM workflow, you will have a waveform, and its spectrum and spectrogram at hand from which you can extract meaningful features. To illustrate feature extraction, we will use data from a real wind turbine. The wind turbine dataset¹¹¹ consists of vibration acceleration recorded from a wind turbine for a period of 50 days. Six seconds of vibration data was recorded each day. A total of 50 files have been provided with each file containing data for a day. A bearing fault leads to increasing vibration levels with failure occurring on the 50th day. Figure 17.4 shows the combined vibration signal for the 50 days. An increasing level of vibration is clearly evident. Let's understand how features can be extracted for a waveform from one day of data. We will reuse this dataset for building a metric called health indicator using only time domain features in Part 7 of the book and therefore, we will look at the code for time domain feature generation.

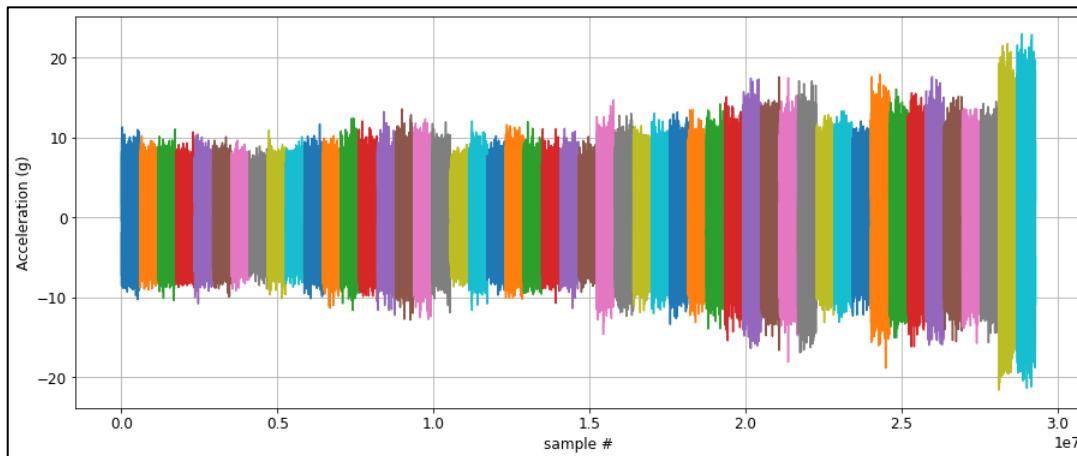


Figure 17.4: Vibration signal from Wind Turbine dataset

Time domain features

When a fault occurs in a rotating machine, the vibration levels and distribution of waveform may change. Waveform features can reflect these changes. Table 1 below shows the commonly used time domain features. We assume that the signal is $y(n)$ for where $n = 1, 2, \dots, N$ where N is the number of data points.

¹¹¹ Available at <https://github.com/mathworks/WindTurbineHighSpeedBearingPrognosis-Data>. Data has been shared by MathWorks under Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) license (<https://creativecommons.org/licenses/by-nc-sa/4.0/>). Permission was granted by the original author of the dataset, Eric Bechhoefer, to use the data in this book.

Mean	Standard deviation	Root mean square (RMS)
$Y_m = \frac{1}{N} \sum_{n=1}^N y(n)$	$Y_{std} = \sqrt{\frac{\sum_{n=1}^N (y(n) - Y_m)^2}{N - 1}}$	$Y_{rms} = \sqrt{\frac{\sum_{n=1}^N (y(n))^2}{N}}$
Peak	Peak-to-Peak	Skewness
$Y_{peak} = \max y(n) $	$Y_{P2P} = \max y(n) - \min y(n)$	$Y_{skewness} = \frac{\sum_{n=1}^N (y(n) - Y_m)^3}{(N - 1) Y_{std}^3}$
Kurtosis	Shape factor	Crest factor
$Y_{skewness} = \frac{\sum_{n=1}^N (y(n) - Y_m)^4}{(N - 1) Y_{std}^4}$	$Y_{SF} = \frac{Y_{rms}}{\frac{1}{N} \sum_{n=1}^N y(n) }$	$Y_{CF} = \frac{Y_{peak}}{Y_{rms}}$
Impulse factor	Margin (or clearance) factor	
$Y_{IF} = \frac{Y_{peak}}{\frac{1}{N} \sum_{n=1}^N y(n) }$	$Y_{MF} = \frac{Y_{peak}}{\left(\frac{1}{N} \sum_{n=1}^N \sqrt{ y(n) }\right)^2}$	

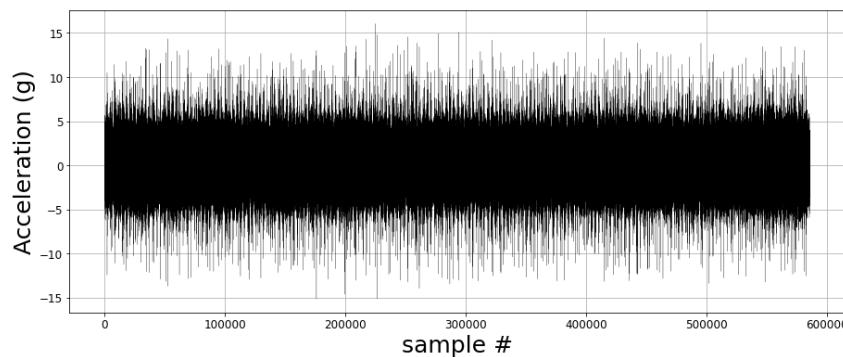
Table 1: Time domain statistical features

Features like RMS (root mean square), STD (standard deviation), peak, and P2P (peak to peak) show increasing trend as a fault becomes more and more severe. RMS quantifies the energy content in the vibration signal and is a popular metric for vibration monitoring. RMS is also more stable and robust to noise compared to other metrics such as peak and P2P. The code below shows how to compute these features.

```
# import packages
import numpy as np, matplotlib.pyplot as plt, pandas as pd
import scipy.io

# fetch data for a waveform (file data-20130418T230803Z.mat randomly selected)
matlab_data = scipy.io.loadmat('data-20130418T230803Z.mat', struct_as_record = False)
vib_data = matlab_data['vibration'][ :, 0]

plt.figure(figsize=(15,6))
plt.plot(vib_data, linewidth=0.2)
plt.xlabel('sample #', fontsize=25), plt.ylabel('Acceleration (g)', fontsize=25)
```



```
# compute RMS, STD, peak, peak2peak
N = len(vib_data)
vibStd = np.std(vib_data, ddof=1, axis=0)
vibRMS = np.sqrt(np.sum(vib_data ** 2)/N)
vibPeak = np.max(np.abs(vib_data))
vibPeak2Peak = np.max(vib_data) - np.min(vib_data)
```

```
>>> Standard deviation: 2.288
>>> RMS: 2.293
>>> Peak: 16.049
>>> Peak-to-Peak: 31.163
```

RMS (and peak, P2P) are not capable of detecting faults during early stages. Kurtosis and skewness are more preferred for incipient faults. Kurtosis is a good indicator of an impulsive/spiky signal. Skewness quantifies the asymmetry of the distribution of the vibration signal amplitude. It is known that for a normally functioning machine, kurtosis and skewness are around 3 and 0, respectively. For faulty machines, kurtosis becomes higher than 3 and skewness deviate from 0 significantly. Note that kurtosis is known to be not suitable for very severe faults as its value decreases when the waveform becomes less impulsive.

```
# compute kurtosis, skewness
from scipy.stats import kurtosis, skew

vibKurtosis = kurtosis(vib_data, fisher=False)
vibSkewness = skew(vib_data, axis=0)

>>> Kurtosis: 4.378
>>> Skewness: 0
```

Code for computation of other time domain features is shown below.

```
# rest of the time domain features
vibMean = np.mean(vib_data)
vibShapeFactor = vibRMS / (np.mean(np.abs(vib_data)))
vibCrestFactor = np.max(np.abs(vib_data)) / vibRMS
vibImpulseFactor = np.max(np.abs(vib_data)) / (np.mean(np.abs(vib_data)))
vibMarginFactor = np.max(np.abs(vib_data)) / (np.mean(np.sqrt(abs(vib_data)))) ** 2

>>> Mean, Shape Factor, Crest Factor, Impulse Factor, Margin Factor: 0.160, 1.296, 6.998, 9.073,
10.854
```

Frequency domain features

Although time domain features are easy to compute and interpret, a major shortcoming with them is that they do not provide clues regarding the underlying source of abnormal vibrations. Frequency domain features enjoy several advantages over time domain features. Abnormal frequencies show up with significant amplitudes in the vibration spectrum in the presence of faults. Frequency domain features are good indicators of both incipient and severe faults. Moreover, presence of specific frequency components provide direct clues regarding which component of a rotating machine has failed.

When provided a vibration spectrum, one of the first thing an expert may look at is the presence of harmonic frequencies corresponding to the rotational speed of the machine. For example, if a machine is rotating at 3600 RPM (rotations per minute), then the amplitudes at 60 Hz (called 1X), 120^{12} Hz (called 2X or second harmonic), and higher harmonics are extracted as features. You will see in the next chapter how these frequency components are useful for FDD. Analogous to Table 1, Table 2 shows the common statistical features derived from vibration spectrum. Here, $Y(k)$ is the amplitude at the k^{th} spectrum line and f_k is the corresponding frequency for $k = 1, 2, \dots, K$.

¹¹² Sub-harmonics (0.5X, 0.25X, ...) are also used as useful features

Mean	Variance	Third moment
$F_1 = \frac{1}{N} \sum_{k=1}^K Y(k)$	$F_2 = \frac{\sum_{k=1}^K (Y(k) - F_1)^2}{K - 1}$	$F_3 = \frac{\sum_{k=1}^K (Y(k) - F_1)^3}{K (\sqrt{F_2})^3}$
Fourth moment	Frequency center	Standard deviation 1
$F_4 = \frac{\sum_{k=1}^K (Y(k) - F_1)^4}{K F_2^2}$	$F_5 = F_{FC} = \frac{\sum_{k=1}^K f_k Y(k)}{\sum_{k=1}^K Y(k)}$	$F_6 = \sqrt{\frac{\sum_{k=1}^K (f_k - F_5)^2 Y(k)}{K}}$
Root mean square frequency (RMSF)	D factor	E factor
$F_7 = F_{RMSF} = \sqrt{\frac{\sum_{k=1}^K f_k^2 Y(k)}{\sum_{k=1}^K Y(k)}}$	$F_8 = \sqrt{\frac{\sum_{k=1}^K f_k^4 Y(k)}{\sum_{k=1}^K f_k^2 Y(k)}}$	$F_9 = \frac{\sum_{k=1}^K f_k^2 Y(k)}{\sqrt{\sum_{k=1}^K Y(k) \sum_{k=1}^K f_k^4 Y(k)}}$
G factor	Third moment 1	Fourth moment 1
$F_{10} = \frac{F_6}{F_5}$	$F_{11} = \frac{\sum_{k=1}^K (f_k - F_5)^3 Y(k)}{K F_6^3}$	$F_{12} = \frac{\sum_{k=1}^K (f_k - F_5)^4 Y(k)}{K F_6^4}$
	H factor	
	$F_{12} = \frac{\sum_{k=1}^K (f_k - F_5)^{1/2} Y(k)}{K \sqrt{F_6}}$	

Table 2: Frequency domain statistical features¹¹³¹¹³ https://github.com/Oybek90/Machine_Learning_from_scratch/blob/main/frequency-domain-feature-extraction-methods.ipynb.

Yaguo Lei, Intelligent Fault Diagnosis and Remaining Useful Life Prediction of Rotating Machinery.

The first feature, mean of the spectrum, is the average of the amplitudes of all the frequencies in the spectrum. As fault severity increases, overall vibration energy goes up and so does F_1 . Frequency center (F_5 or F_c) and root mean square frequency (F_7 or RMSF) are used to track the dominant frequencies in the spectrum. The peak frequencies (say, top 10) and their respective amplitudes are also commonly used for fault detection.

Impact of operating conditions



Variations in machine load and running speed impact machine vibration characteristics such as overall vibration levels, amplitudes of specific frequency components, etc. Therefore, operating condition should be taken into account during interpretation of vibration features. Very often, a reference vibration signal and/or spectrum is recorded for different operational conditions and used later for FDD. Alternatively, operations conditions are used as additional variables for training ML models.

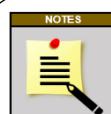
Time-frequency domain features

A commonly extracted time-frequency domain feature is mean peak frequency which is simply average of the peak frequencies at different time instances in the spectrogram. It is computed as shown below where N_τ is the number of segments used in SFTF.

$$f_{peak}(\tau) = \underset{f}{\operatorname{argmax}} S(\tau, f)$$

$$f_{mean, \text{ peak}} = \frac{\sum_\tau f_{peak}(\tau)}{N_\tau}$$

Another common strategy is to extract frequency domain features from each FFT spectrum of the waterfall.



Fault prone components of a rotating machine such as gears and bearings have been well studied and their ‘frequencies of defect’ have been discovered and defined mathematically. These frequencies show up in spectrum when faults occur and therefore work quite well as diagnostic features. Readers are encouraged to see the work by Sim et al. titled ‘A Tutorial for Feature Engineering in the Prognostics and Health Management of Gears and Bearings’ (Applied Sciences, 2020).

With this we have completed our study of the first phase of VCM which is summarizing complex waveforms from rotating machines into a bunch of features. These features are used for fault detection, fault classification, and fault prognosis as we will see in the upcoming chapters. Many more advanced techniques than what we have covered in this chapter have been devised for analysis of vibration signals. These techniques include, amongst others, envelope analysis, wavelet transforms, empirical mode decomposition, and Hilbert-Huang transform. Although we have only scratched the surface of the broad field of vibration signal processing, you now have the fundamentals in place to navigate this world confidently!

Summary

In this chapter, we acquainted ourselves with the basics of processing vibration signals as a precursor to building ML-based fault detection solutions. We studied the concept of vibration spectrum and understood how to generate the spectrum. We looked into some detail how to extract time domain, frequency domain, and time-frequency domain features. We will continue building upon this foundation and see how to utilize these features to detect faults in rotating machines.

Chapter 18

Vibration-based Condition Monitoring: Fault Detection & Diagnosis

Vibration-based condition monitoring was already a widely adopted technique in process industry long before ML craze took over the manufacturing world. The International Organization for Standardization (ISO) has come up with alarm limits for vibration RMS for different classes of rotating machines. Additionally, VCM researchers have worked diligently to discover the characteristics signatures of failures in different components of a rotating machines. Correspondingly, several rules of thumb and heuristics have been devised to pinpoint root causes of faults using vibration features. However, these heuristics do not cover all possible fault scenarios and a vibration expert is still required to conduct analysis and interpretation of vibration signal features. Fortunately, the advent of machine learning has made VCM more accessible to generic process data scientists.

Several different types of ML models have been reported in VCM literature for fault classification, fault detection, and fault diagnosis. For example, fault detection applications have been built by using the whole spectrum (or waveform) as input to an autoencoder or spectrogram image as input to a CNN (convolutional neural network) model. ML models don the cap of a vibration expert to find the patterns in vibration signal, distinguish between NOC and abnormal vibrations, and discriminate between different fault conditions. In this chapter, we will look at one such implementation of ML-based VCM. Specifically, the following topics are covered

- VCM workflow
- Classical approaches for VCM
- SVM-based fault classification of motors

18.1 VCM Workflow: Revisited

Vibration signals contain indicators of machine faults. Previously, we saw the steps commonly taken to ‘amplify’ these indicators through judicious extraction of features. In this chapter, we will focus on how these features are used to make inferences regarding health of rotating machinery. Figure 18.1 shows some of the approaches commonly employed. The classical approaches include, amongst others, simply looking for the presence of harmonics in the spectrum and comparing individual features against ISO-recommended thresholds. In recent times, ML-based VCM is gradually becoming more popular. Any of the ML techniques that we have seen in the previous parts of the book can be employed.

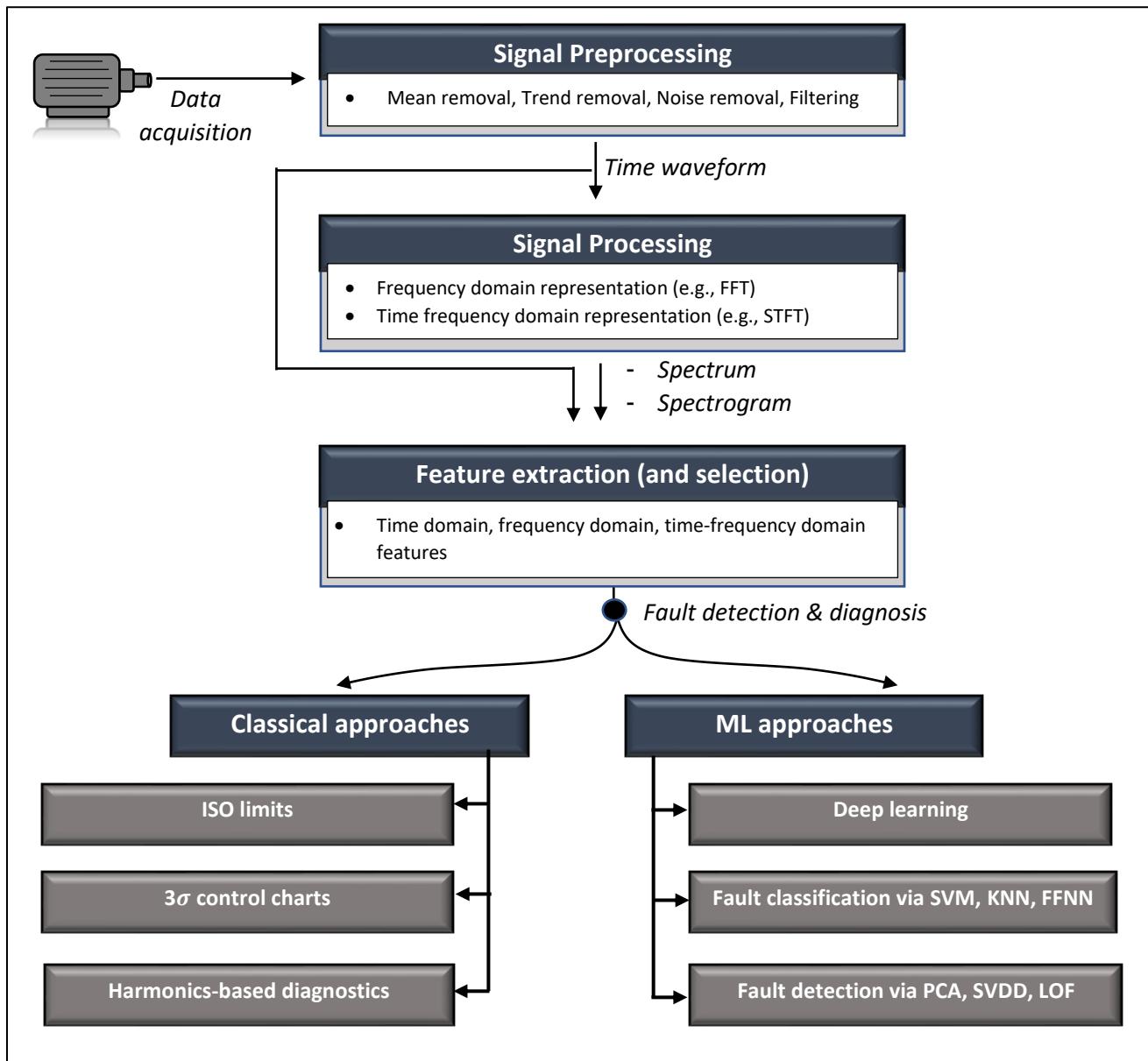
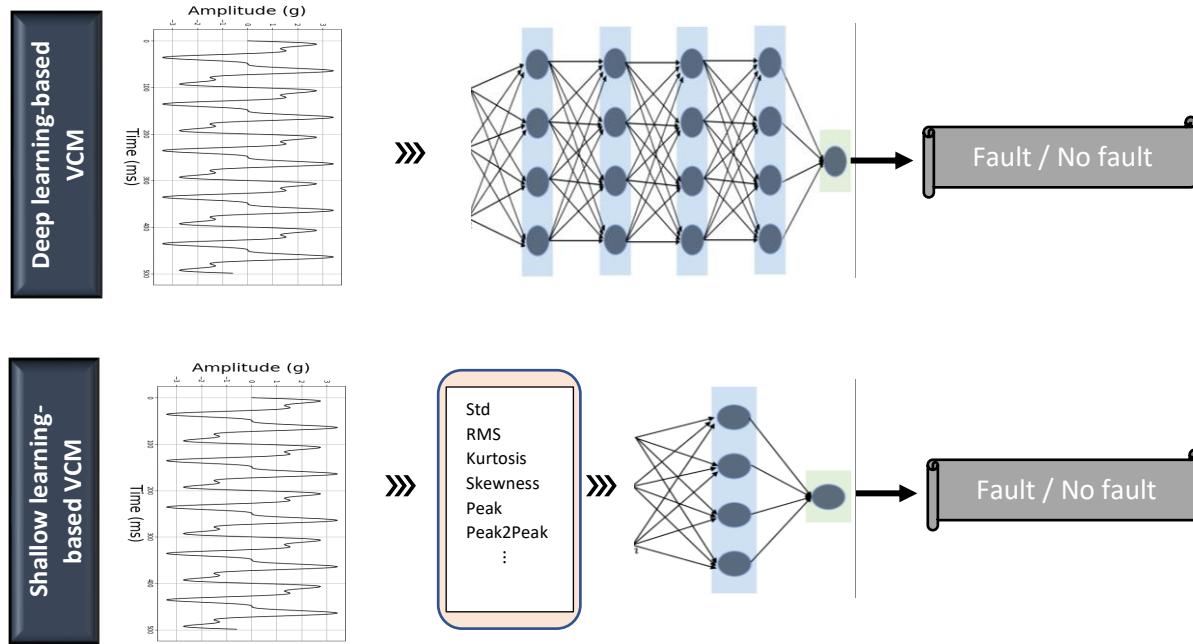


Figure 18.1: Vibration condition monitoring workflow - revisited

The phenomenal success of deep learning in areas of computer vision and natural language processing has encouraged VCM practitioners to bypass explicit feature extraction and directly use the vibration waveform/spectrum/spectrogram as inputs to the deep learning models; for example, passing 2D spectrogram images as inputs to convolutional neural networks models. Deep learning automatically learns the fault signatures in vibration data. Illustrations below compares shallow learning- and deep learning-based VCM using time domain waveforms.



18.2 Classical VCM Approaches: A Quick Primer

Below we discuss a few (easy to implement and understand) classical approaches for VCM.

Control charts for vibration features

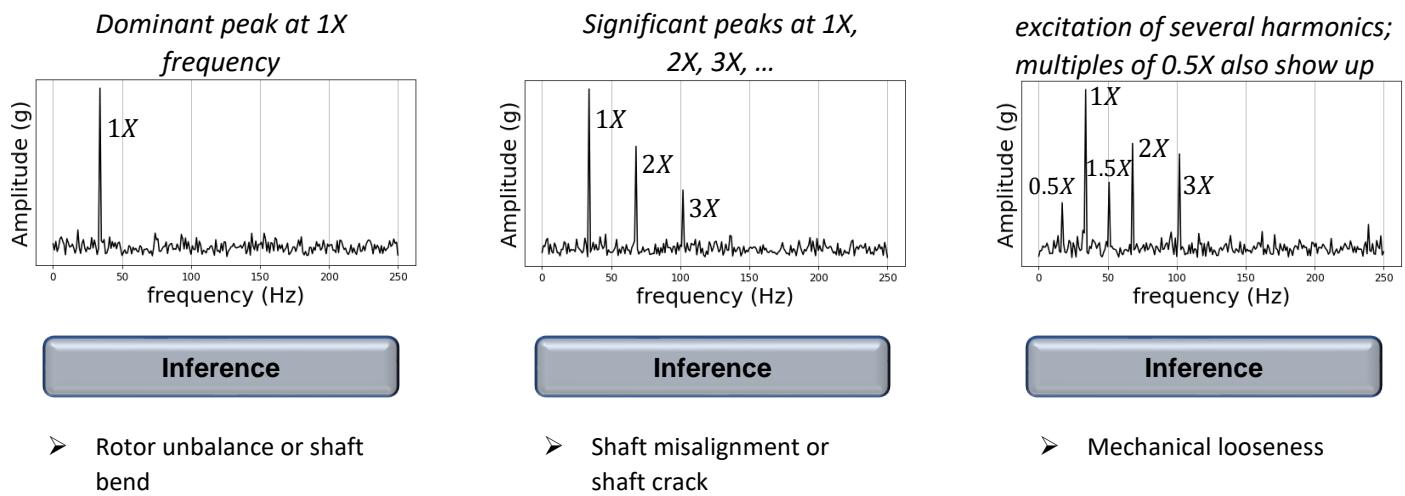
In this approach, some key features (say, RMS, kurtosis, etc.) are selected as the fault indicators to monitor. Each selected indicator is monitored using Shewhart control charts, i.e., a machine is considered healthy if each feature remains within the range $\mu \pm 3\sigma$ where the mean μ and standard deviation σ are computed from NOC waveforms.

ISO Code

An alternative to 3σ control chart is to directly compare your machine's vibration level to industrial standards. One such standard is ISO 10816¹¹⁴ that provide good/satisfactory/unsatisfactory levels of RMS values for different types (steam turbines, hydro turbine, etc.) and classes (small/medium/large) of machines.

Harmonics-based FDD

As alluded to in Chapter 17, harmonics of a machine's rotating speed provide crucial clues regarding specific causes of abnormal vibrations. A few examples of abnormal spectra of a machine rotating at 2040 RPM are shown along with fault inferences.



More details on these heuristics-based inferences can be found in any classical book on VCM¹¹⁵.

¹¹⁴ <https://www.vibsens.com/index.php/knowledge-base/iso10816-iso7919-charts/iso10816-charts>

¹¹⁵ Jyoti K. Sinha, Industrial Approaches in Vibration-based Condition Monitoring. CRC Press, 2020.

18.3 Machine Learning-based VCM: Motor Fault Classification via SVM

To demonstrate ML-based VCM, we will use a popular dataset called CWRU bearing dataset¹¹⁶. The dataset includes vibrations collected from an electric motor under NOC and different faulty conditions, under varying loads, and at 48 kHz and 12 kHz. For the purpose of our implementation, we will consider data collected at 48 kHz from the drive end of the motor with 1 hp load. In total, vibrations are recorded under the following 10 different conditions:

- C1 : Ball defect (0.007 inch)
- C2 : Ball defect (0.014 inch)
- C3 : Ball defect (0.021 inch)
- C4 : Inner race fault (0.007 inch)
- C5 : Inner race fault (0.014 inch)
- C6 : Inner race fault (0.021 inch)
- C7 : Normal operation
- C8 : Outer race fault (0.007 inch, data collected from 6 O'clock position)
- C9 : Outer race fault (0.014 inch, data collected from 6 O'clock)
- C10 : Outer race fault (0.021 inch, data collected from 6 O'clock)

The data from each condition is provided in a separate file (such as *110.mat*) on the CWRU website¹¹⁷. Data from each file is divided into smaller segments of 2048 data points and 230 segments are obtained from each of the 10 files corresponding to C1 to C10. In total 2300 waveforms are available to us. For each segment, nine time domain features (peak, peak-to-peak, mean, standard deviation, RMS, skewness, kurtosis, crest factor, and shape factor) are extracted. This results in a feature matrix of size 2300 X 9. A column indicating the condition class is also added to the end and the final data matrix is provided in the file *feature_time_48k_2048_load_1.csv*¹¹⁸. Our objective is to develop a ML model that can predict the operating condition of the motor using the time domain features. For this, we will build a kernel SVM-based classifier as shown below.

```
# import packages
import numpy as np, pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
```

¹¹⁶ <https://engineering.case.edu/bearingdatacenter>

¹¹⁷ <https://engineering.case.edu/bearingdatacenter/48k-drive-end-bearing-fault-data>

¹¹⁸ The file and the shown SVM implementation have been adapted from the work (Copyright (c) 2022 Biswajit Sahoo) of Biswajit Sahoo (https://github.com/biswajitsahoo1111/cbm_codes_open) which is shared under MIT License (https://github.com/biswajitsahoo1111/cbm_codes_open/blob/master/LICENSE.md)

```

from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix
import seaborn as sns

# read feature data
featureData = pd.read_csv('feature_timeDomain_48k_2048_load_1HP.csv')

```



Index	peak2peak	peak	mean	Std	RMS	skewness	kurtosis	CrestFactor	ShapeFactor	faultType
0	0.78	0.36	0.018	0.12	0.12	-0.12	-0.042	2.9	7	Ball_007_1
1	0.83	0.47	0.022	0.13	0.13	0.17	-0.082	3.5	6	Ball_007_1
2	0.91	0.47	0.02	0.15	0.15	0.04	-0.27	3.1	7.4	Ball_007_1
3	1.1	0.58	0.021	0.16	0.16	-0.023	0.13	3.7	7.6	Ball_007_1
4	1	0.45	0.022	0.14	0.14	-0.082	0.4	3.2	6.3	Ball_007_1

```

featureData['faultType'] = pd.Categorical(featureData['faultType']) # designates last column as
# categorical variable

```

```

# generate training and test datasets
train_data, test_data = train_test_split(featureData, test_size = 750, stratify =
                                         featureData['faultType'], random_state = 1234)
print(train_data['faultType'].value_counts())

```

```

>>> Ball_007_1 155
Ball_014_1 155
Ball_021_1 155
IR_007_1 155
IR_014_1 155
IR_021_1 155
Normal_1 155
OR_007_6_1 155
OR_014_6_1 155
OR_021_6_1 155

```

```

# scale data
scaler = StandardScaler()
train_data_scaled = scaler.fit_transform(train_data.iloc[:, :-1])
test_data_scaled = scaler.transform(test_data.iloc[:, :-1])

```

```

# find best SVM hyperparameters via grid-search
hyperParameters = {'C':[0.1, 1, 10, 50, 100, 300], 'gamma':[0.01, 0.05, 0.1, 0.5, 1, 10]}

```

```

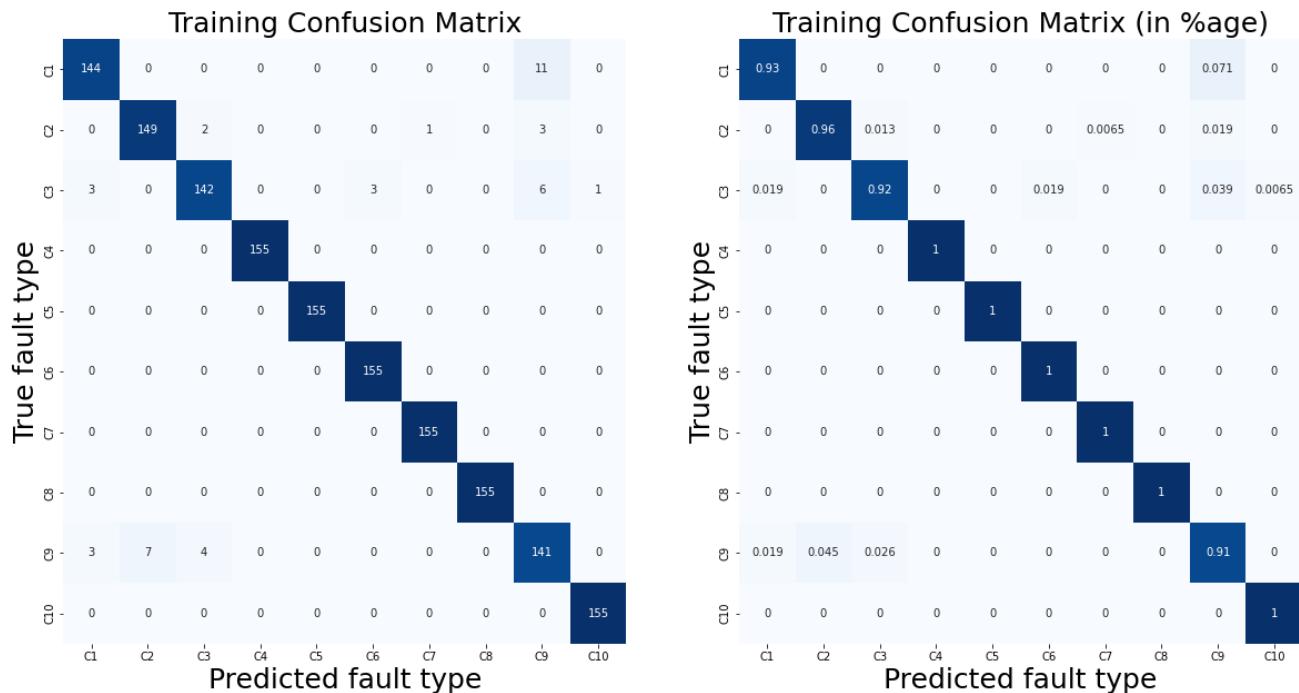
svm_clf = GridSearchCV(SVC(), hyperParameters, cv= 5)
svm_clf.fit(train_data_scaled, train_data['faultType'])
print(svm_clf.best_params_)

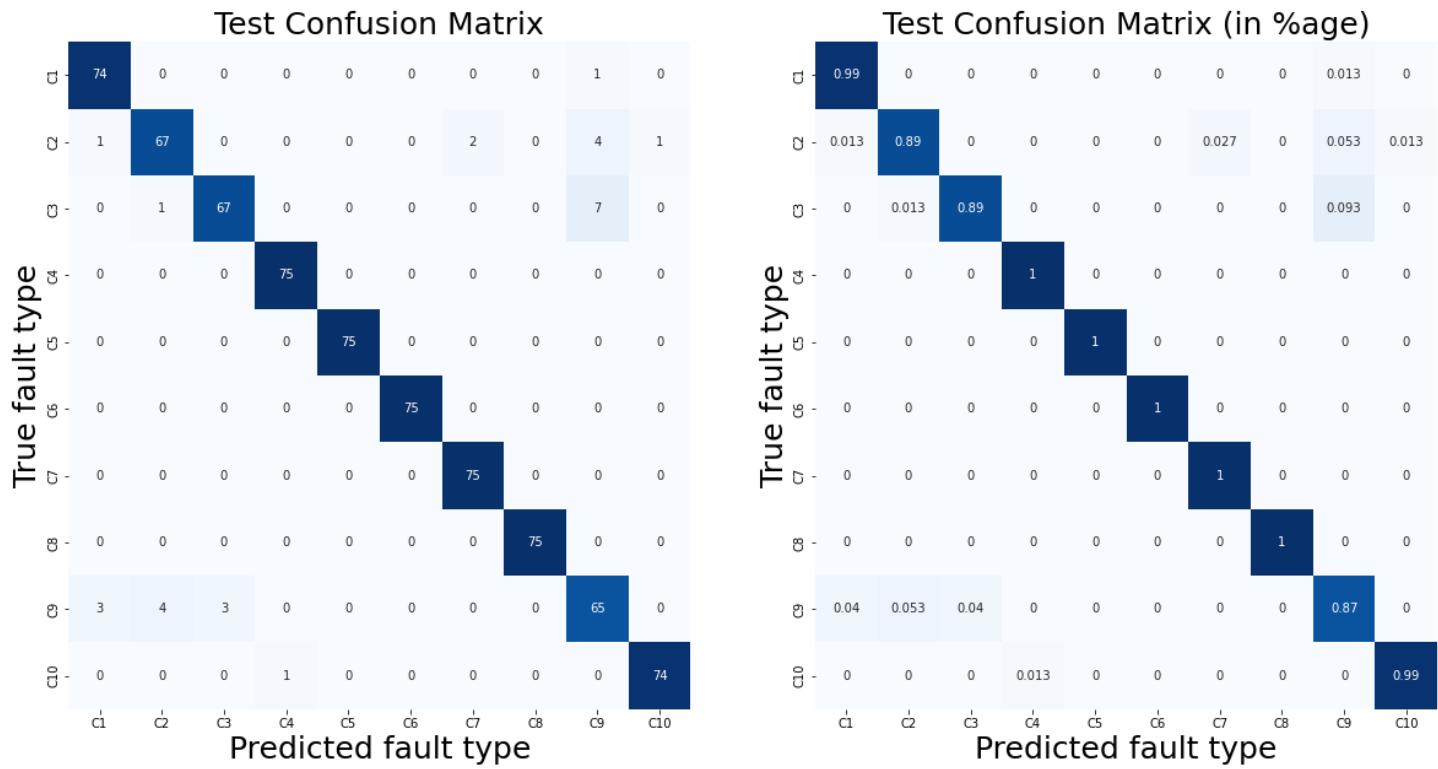
>>> {'C': 50, 'gamma': 0.1}

# predict fault classes and generate confusion matrices
train_pred = svm_clf.predict(train_data_scaled)
test_pred = svm_clf.predict(test_data_scaled)
CM_train = confusion_matrix(train_data['faultType'], train_pred)
CM_test = confusion_matrix(test_data['faultType'], test_pred)

fault_type = ['C1','C2','C3','C4','C5','C6','C7','C8','C9','C10']
sns.heatmap(CM_train, annot= True, fmt = "d", xticklabels=fault_type, yticklabels=fault_type,
             cmap = "Blues", cbar = False)
sns.heatmap(CM_test, annot = True, xticklabels=fault_type, yticklabels=fault_type, cmap =
             "Blues", cbar = False)

```





The confusion matrices indicate good performance by the SVM classifier. With this, we end our quick look at vibration-based monitoring of rotating equipment. VCM is a huge field of study by itself, and it was impossible to fit this vast area in a single part of this book. Nonetheless, we hope that the current and the previous chapters have imparted you a working-level knowledge of deploying ML for VCM.

Summary

In this chapter, we focused on the fault detection and diagnosis phase of VCM workflow. We looked at the classical and ML-based approaches for making inferences from processed vibration signals. Finally, we implemented a SVM-based classifier for fault classification of a motor using the popular CWRU dataset.

Part 7

Predictive Maintenance

Chapter 19

Fault Prognosis: Concepts & Methodologies

All machines eventually break and plant operators have traditionally relied upon regular time-based (preventive) maintenance to avoid costly downtimes due to machinery failures. Although economically inefficient, preventive maintenance remained the default approach in process industry for a long time. Only in recent times, condition-based maintenance approach has gained widespread acceptance wherein a machine's real-time data is used to assess the machine's health, detect failures, and trigger (on-demand) maintenance. However, the recent advancement in data mining has brought another step change in the mindset of plant reliability personnel: mere detection of machine faults is no longer good enough; accurate forecast of the fault's progression leading to predictive maintenance (PdM) is the new vogue. The lure of PdM is obvious – it facilitates advance planning of maintenance, better management of spare part's inventory, etc. Correspondingly, PdM is the holy grail that industrial executives are striving for to remain competitive.

PdM, in essence, involves fault prognosis or the prediction of a machine's health degradation over time after detection of incipient faults. Different PdM methodologies are employed depending on the availability of fundamental knowledge of fault's mechanism, historical run-to-failure data, etc. The dominant PdM approach involves computation of a health indicator (HI) that summarizes the state of a machine health and shows a clear degradation trend as an incipient fault progresses from incipience to high severity. HI allows computation of RUL (remaining useful life) which is the remaining time until fault severity crosses failure threshold necessitating the machine being taken out of service.

Several different strategies have been devised for computation of HIs and the subsequent RUL estimation. While the RUL estimation strategies are covered in detail in the next chapter, this chapter focusses on the data-driven methods for HI computations. Specifically, the following topics are covered

- Concepts and methodologies for PdM
- Fault prognosis: introduction and workflow
- Approaches for health indicator computation
- Fault prognosis case study for wind turbines

19.1 Fault Prognosis: Introduction & Workflow

Fault prognosis simply refers to the task of estimating the progression of health degradation of a machine¹¹⁹. Fault prognosis kick in after a fault has been detected. The end objective of fault prognosis is to estimate the time remaining until fault severity hits failure threshold. A machine or an operation unit may be kept in operation (even with faults) until it reaches failure conditions. Therefore, estimation of the time remaining or RUL can help plant operators maximize an equipment lifetime and plan maintenance judiciously. Figure 19.1 presents the different prognostic methodologies that can be employed depending on the level of available information about fault mechanism and past fault data.

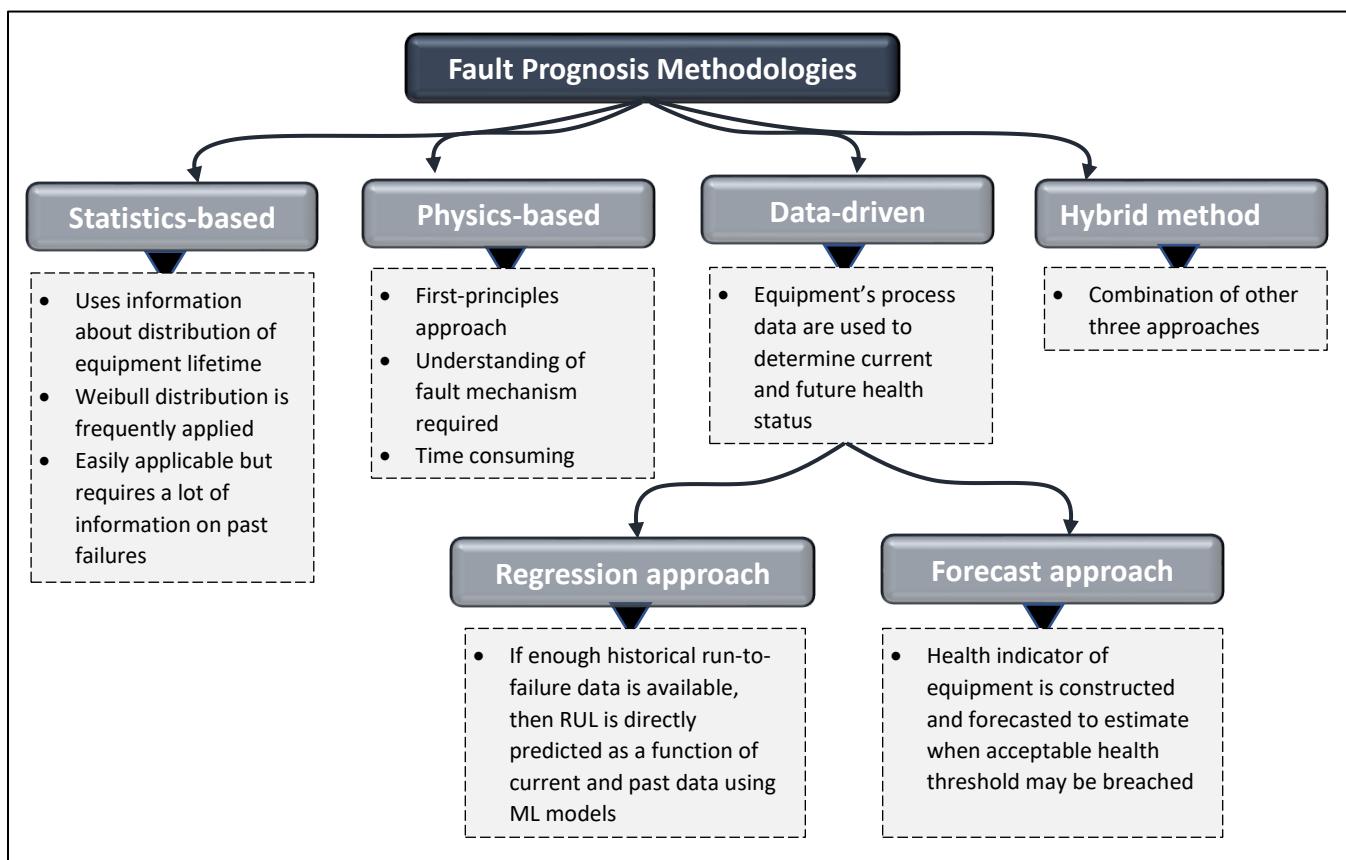


Figure 19.1: Prognostics Methodologies

Among the shown approaches, HI-based approach is very popular. As shown in Figure 19.2, a curve showing the current trend of fault severity or health condition is computed. Thereafter, the future progression of the curve is predicted to estimate the RUL. In this chapter, we will look at how such curves can be generated in a data-driven way. The strategy for HI forecast is covered in detail in the next chapter.

¹¹⁹ Fault prognosis is not limited to health prediction of machines only. It is applicable to a subprocess of a plant and the whole plant as well.

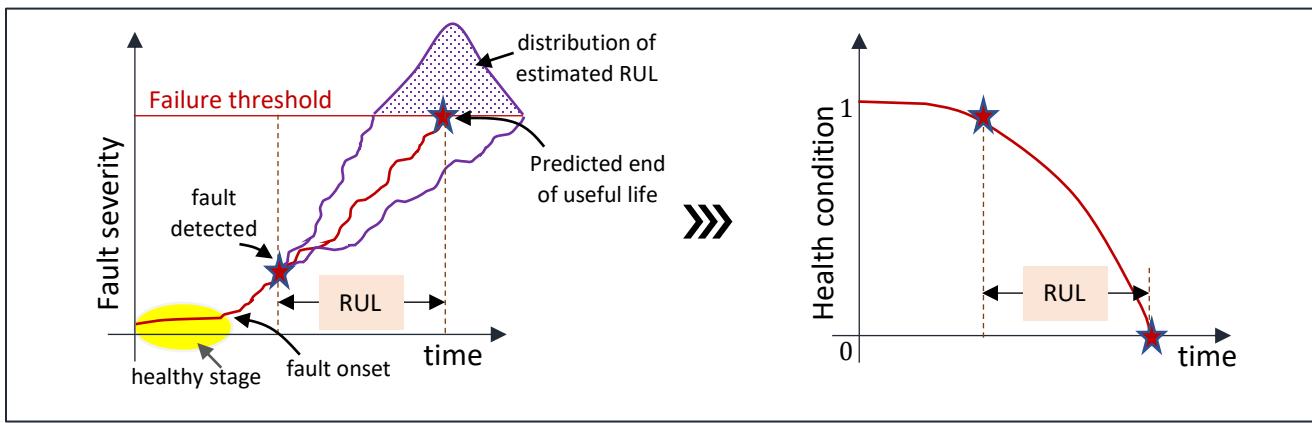


Figure 19.2: Fault severity and health condition progression with time

Figure 19.3 shows the typical workflow for data-driven fault prognosis. Although fault prognosis is preceded by fault detection, technically the models used for the two tasks can be different; therefore, fault prognosis has been presented as a standalone modeling exercise. It is not uncommon to have the same HI used first for fault detection and then subsequently for fault prognosis. Figure 19.3 also shows two non-HI-based approaches where RULs are estimated directly, i.e., pre-processed raw data or features are used as model inputs and RUL is the predicted output. The requirement of abundant historical run-to-failure data (which are rarely available in abundance) for training the DL and ML models make these regression-based approaches less common.

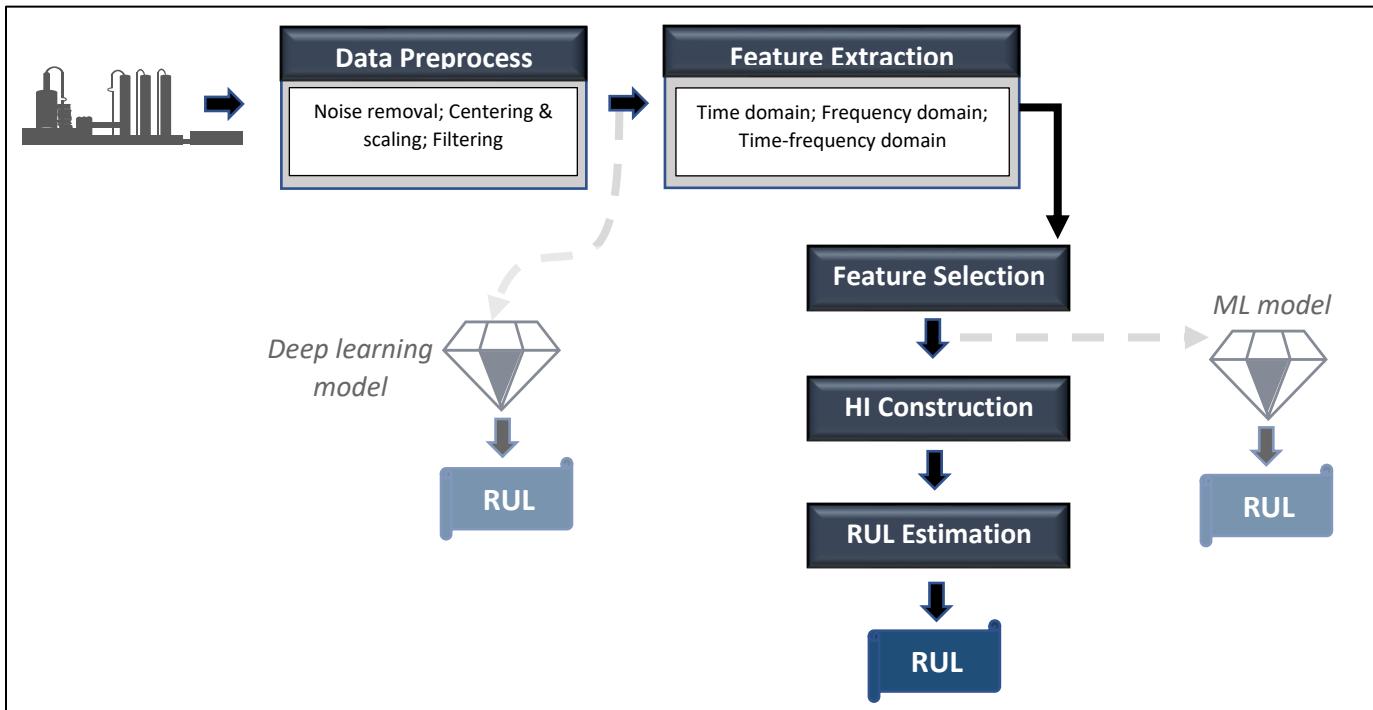


Figure 19.3: Typical workflow for data-driven fault prognosis

Let's now move on and learn how HIs are actually computed.

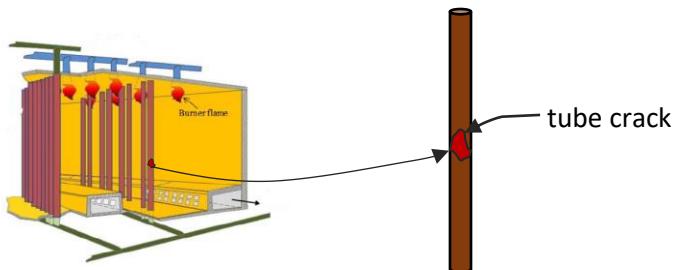
Maintenance Strategies

The maintenance strategies in process industry have been strongly influenced by the advances in ML and sensor technologies. It has evolved from time-based preventive maintenance to proactive monitoring-based condition-based maintenance (CBM), and to advanced prediction-based predictive maintenance (PdM).

Preventive Maintenance	Condition-based Maintenance	Predictive Maintenance
<ul style="list-style-type: none"> Equipment maintenance is done at pre-set schedule Non-optimal strategy as maintenance may happen too early or too late E.g., a compressor may be scheduled to be replaced every 5 years 	<ul style="list-style-type: none"> Equipment maintenance is done when signs of faults are detected Manual expert analysis is carried out to determine exact maintenance plan E.g., a compressor may be monitored using PCA to detect the need to repair 	<ul style="list-style-type: none"> Degradation of machine's health over time is estimated and the time remaining until failure is computed E.g., a compressor may be monitored using RNN to estimate when vibration RMS will go above failure level

19.2 Machinery Health Indicators: Introduction and Approaches

By now you understand that a health indicator is simply a metric that (directly or indirectly) quantifies the fault status of an equipment or process. For example, consider a catalyst-filled reformer tube in a steam-methane reformer that develops a crack on its outer surface. Specialized tools exist that can measure the depth of crack and thus directly tell us the fault severity. Alternatively, one can measure the surface and furnace gas temperature around the crack which will show abnormal values due to the crack. The value of abnormal deviation (difference between observed and expected value) would indirectly give us the level of fault severity.



HI can simply be one of the measured signals or a combination of them. It may also be derived from a model. Many ML techniques that we worked with in the previous chapters already provide metrics that can be used as a health indicator. For example, in SVDD, the distance from the center in the feature space can be used as a HI as it indicates how far a test system has moved away from the NOC behavior. Figure 19.4 below summarizes the different approaches for HI construction. Let's take a brief look at them.

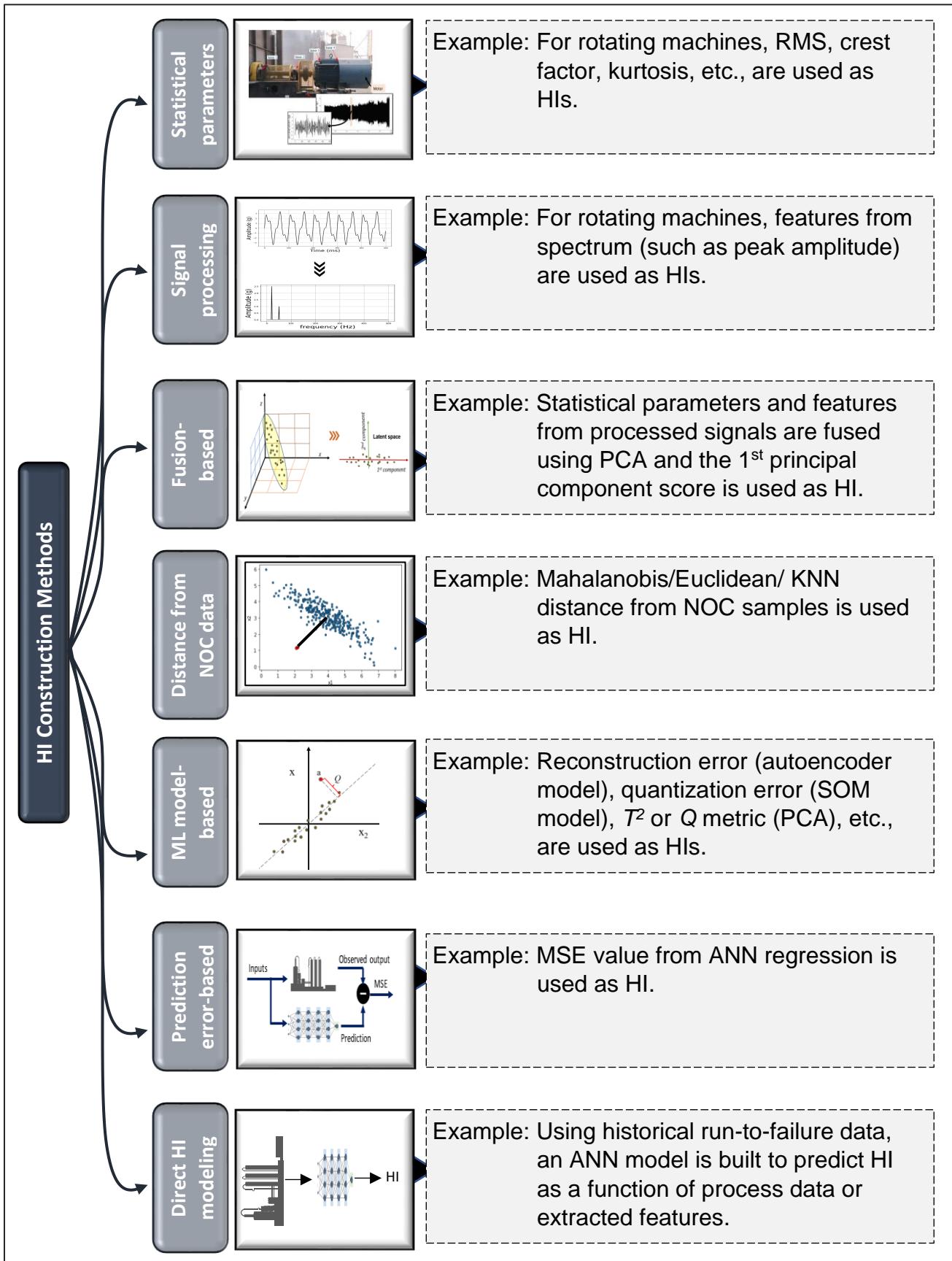


Figure 19.4: Approaches for health indicator construction

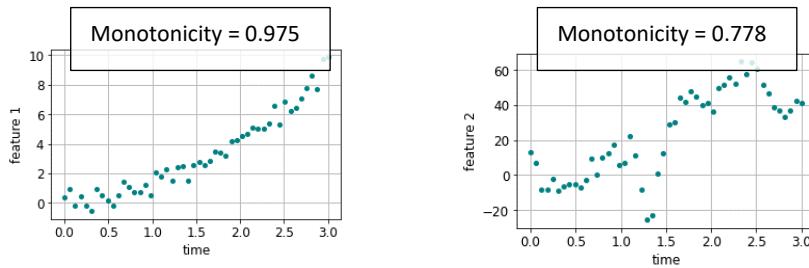
While Figure 19.4 is self-explanatory, a few aspects are worth mentioning.

Dimensional vs dimensionless features

Statistical parameters are commonly used for fault prognosis in rotating machinery. However, as we alluded in Chapter 17, dimensional parameters like RMS depend on operating condition such as machine rotating speed and therefore are not suitable for use as a HI. Dimensionless parameters such as skewness, kurtosis, crest factor, and impulse factor are more suitable.

Feature selection

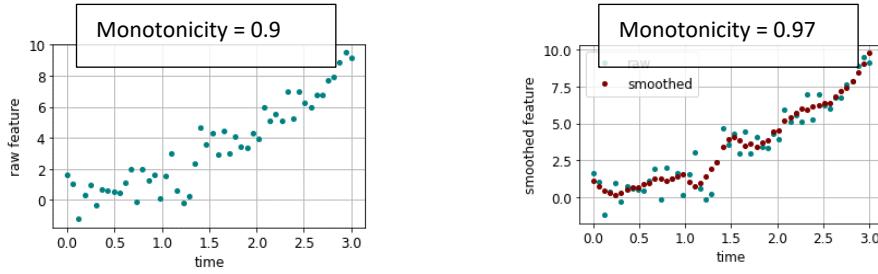
Another aspect is the selection of features for creating fusion-based HI or intermediate model. A common practice is to include only those features that show clear monotonic trend as health condition deteriorates.



Monotonicity is usually computed as the Spearman correlation between the feature values and time which indicates whether the feature continuously increases (or decreases) with time or not. Monotonicity ranges from 1- to 1 where a large absolute value is favorable for prognosis.

Feature smoothing

Extracted features are almost always noisy as shown in the illustration below. Noise can impact the accuracy of RUL estimation and monotonicity value (and therefore feature selection result). Therefore, a common practice is to smooth out the noise in features. Smoothing is commonly conducted via (causal) moving averaging, wherein, the smoothed value at any time is given by the average of the recent past data.

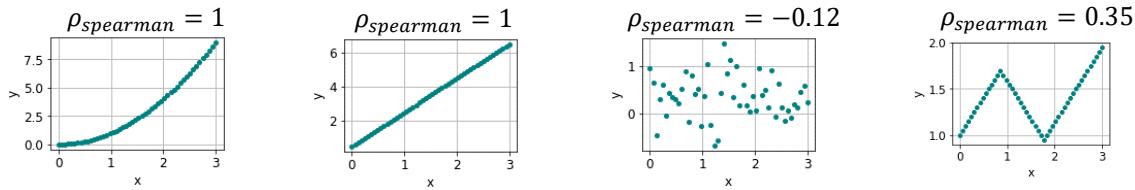


Spearman Correlation

Spearman correlation between two variables x and y is simply the Pearson correlation between the rank of the variables and is often used to quantify nonlinear relationship between a pair of variables.

$$\rho_{spearman} = \frac{\text{cov}(r_x, r_y)}{\sigma_{r_x} \sigma_{r_y}}$$

σ_{r_x} = rank of variable x ; σ_{r_y} = rank of variable y



The plots above make it apparent that $\rho_{spearman}$ can capture the monotonic relationship between two variables very well.

Historical run-to-failure data

The last approach in Figure 9.4 differs from the rest in that it is a supervised learning of HI. Here, historical run-to-failure¹²⁰ data is used to create a map between machine/process data to HI. For any single run-to-failure data sequence, the HI labels are estimated using the heuristics that HI is 1 for healthy machine and 0 for a failed machine. HI values are interpolated for intermediate states. Illustration below shows the procedure.

¹²⁰ Complete sequence of a machine's measured signals from healthy state to failure

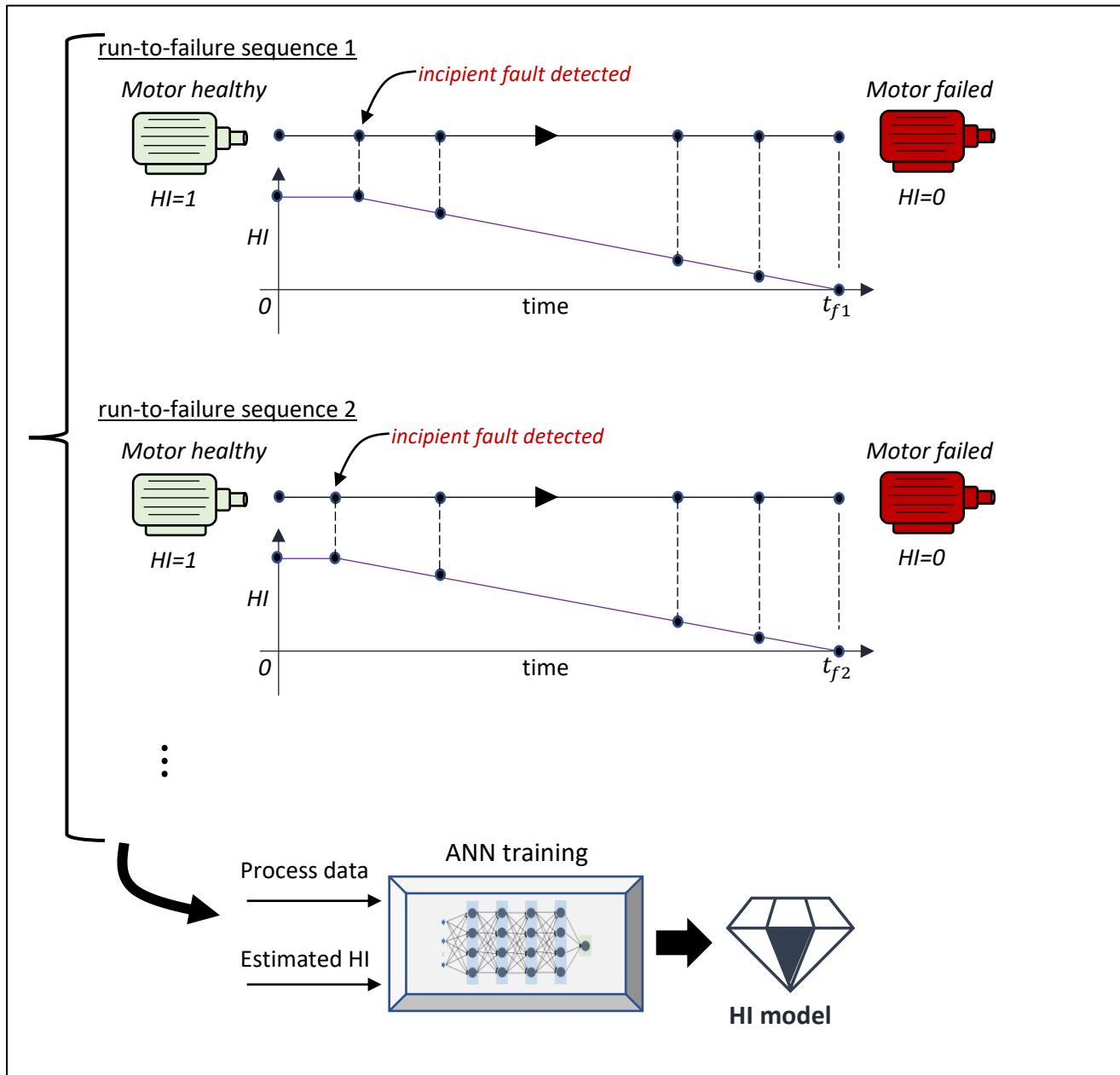


Figure 19.5: Regression-based HI modeling using historical run-to-failure data

Let us bring all the covered concepts together and see an application of approach three from Figure 19.4 for HI construction for a (wind) turbine using vibration signals.

19.3 Health Indicator Construction Using Vibration Signals for a Wind Turbine

As alluded to before in Chapter 17, the wind turbine dataset¹²¹ consists of vibration acceleration recorded from a wind turbine for a period of 50 days. Six seconds of vibration data was recorded each day. A total of 50 files have been provided with each file containing data for a day. A fault in bearing leads to increasing vibration levels with failure occurring on the 50th day. Figure 19.6 shows the combined vibration signal for the 50 days. An increasing level of vibration is clearly evident. We will use this dataset to demonstrate how a HI can be constructed from extracted time domain features that exhibits a clear health degradation trend over time¹²².

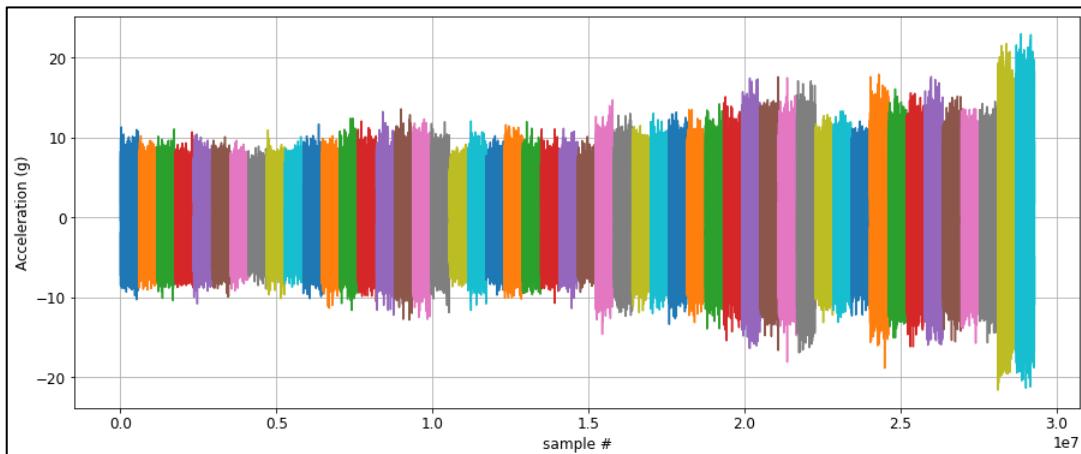


Figure 19.6: Vibration signal from Wind Turbine dataset

Let's begin by importing required packages and defining a utility function that extracts time domain features from a vibration waveform.

```
# import packages
import numpy as np, matplotlib.pyplot as plt, pandas as pd
import scipy.io
import glob
from scipy.stats import kurtosis, skew, spearmanr
```

¹²¹ Available at <https://github.com/mathworks/WindTurbineHighSpeedBearingPrognosis-Data>. Data has been shared by MathWorks under Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) license (<https://creativecommons.org/licenses/by-nc-sa/4.0/>). Permission was granted by the original author of the dataset, Eric Bechhoefer, to use the data in this book.

¹²² MathWorks has provided a case study (<https://www.mathworks.com/help/predmaint/ug/wind-turbine-high-speed-bearing-prognosis.html>) using this dataset wherein they demonstrate how to predict RUL using fusion-based HI. We adopt a similar approach in our case study.

```
# function to compute time domain features
def timeDomainFeatures(VibData):
    N = len(VibData)
    vibMean = np.mean(VibData)
    vibStd = np.std(VibData, ddof=1, axis=0)
    vibRMS = np.sqrt(np.sum(VibData ** 2)/N)
    vibPeak = np.max(np.abs(VibData))
    vibPeak2Peak = np.max(VibData) - np.min(VibData)
    vibSkewness = skew(VibData, axis=0)
    vibKurtosis = kurtosis(VibData, fisher=False)
    vibShapeFactor = vibRMS / (np.mean(np.abs(VibData)))
    vibCrestFactor = np.max(np.abs(VibData)) / vibRMS
    vibImpulseFactor = np.max(np.abs(VibData)) / (np.mean(np.abs(VibData)))
    vibMarginFactor = np.max(np.abs(VibData)) / (np.mean(np.sqrt(abs(VibData)))) ** 2)

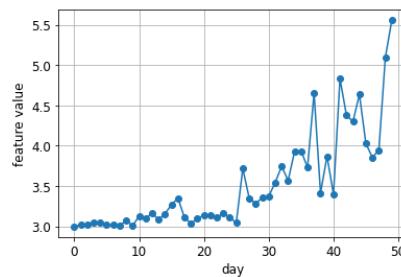
    features = np.array([vibMean, vibStd, vibRMS, vibPeak, vibPeak2Peak, vibSkewness,
                        vibKurtosis, vibShapeFactor, vibCrestFactor, vibImpulseFactor, vibMarginFactor])
    return features
```

Let's now read data from all the 50 files and extract features.

```
# collect feature values for each day
Filenames = glob.glob('data-2013*.mat') # fetches names of all relevant files

Nfeatures = 11
features50days = np.zeros((50, Nfeatures))
for i in range(len(Filenames)):
    matlab_data = scipy.io.loadmat(Filenames[i], struct_as_record = False) # reads data from file123
    vib_data = matlab_data['vibration'][:,0]
    features = timeDomainFeatures(vib_data)
    features50days[i,:] = features

plt.figure(), plt.plot(features50days[:,6], '-o'), plt.xlabel('day'), plt.ylabel('feature value') # kurtosis
```

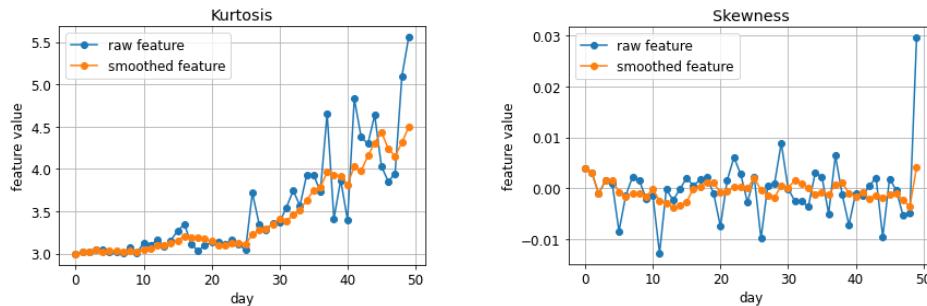


¹²³ Files are in '.mat' format

As we had alluded to in the text, the extracted features are noisy. We will smooth them using moving average.

```
# smooth features using moving average
windowSize = 5
features50days_smoothed = pd.DataFrame(features50days).rolling(windowSize).mean().values
features50days_smoothed[:windowSize-1, :] = features50days[:windowSize-1, :] # replace nan in
# first 4 rows with original values

plt.figure(), plt.plot(features50days[:, 6], label='raw feature') # kurtosis
plt.plot(features50days_smoothed[:, 6], label='smoothed feature')
plt.xlabel('day'), plt.ylabel('feature value'), plt.legend()
```



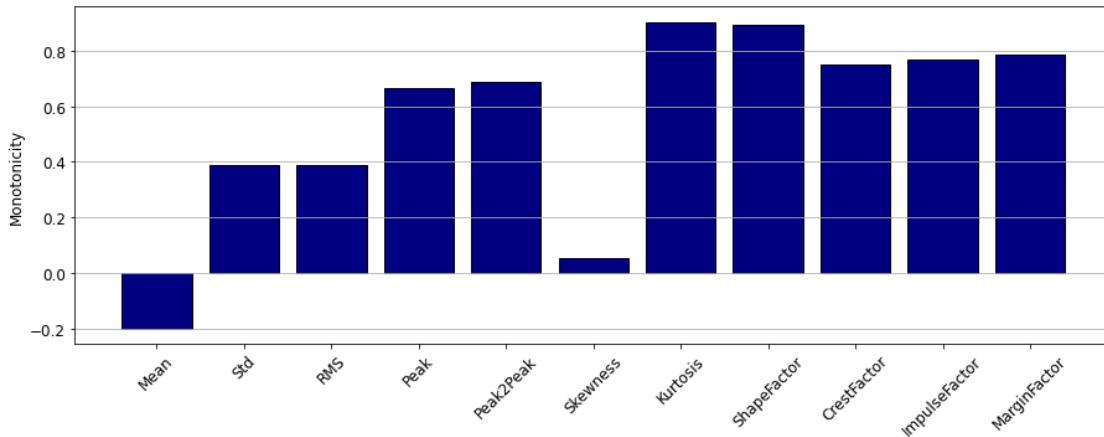
The plots above show that not all features show clear degradation trend. We will use monotonicity to select HI-relevant features for further processing; specifically, we will use only features with monotonicity greater than 0.7. Note that our purpose of computing HI is to eventually predict RUL. For this, we will assume that 32 days of data are available to guide the construction of HI and make a prediction for RUL. The specific strategy used to predict the RUL is shown in the next chapter.

```
# separate training data
Ndays_train = 32 # ~ 2/3rd
features_train = features50days_smoothed[:Ndays_train, :]
features_all = features50days_smoothed

# monotonicity of features
feature_monotonicity = np.zeros((Nfeatures,))
for feature in range(Nfeatures):
    result = spearmanr(range(Ndays_train), features_train[:, feature])
    feature_monotonicity[feature] = result.statistic

# bar plot
featureNames = ['Mean', 'Std', 'RMS', 'Peak', 'Peak2Peak', 'Skewness', 'Kurtosis', 'ShapeFactor',
                 'CrestFactor', 'ImpulseFactor', 'MarginFactor']
```

```
plt.figure(figsize=(15,5))
plt.bar(range(Nfeatures), feature_monotonicity, tick_label=featureNames)
plt.xticks(rotation=45), plt.ylabel('Monotonicity'), plt.grid(axis='y')
```



```
# pick features with monotonicity >= 0.7
featuresSelected = np.where(np.abs(feature_monotonicity) >= 0.7)[0]
selectFeatures_train = features_train[:,featuresSelected]
selectFeatures_all = features_all[:,featuresSelected]
```

Let's now fuse the selected features into a single metric. For this purpose, we will utilize the PCA technique as shown below. Notice that we again use only the training data to generate the PCA model.

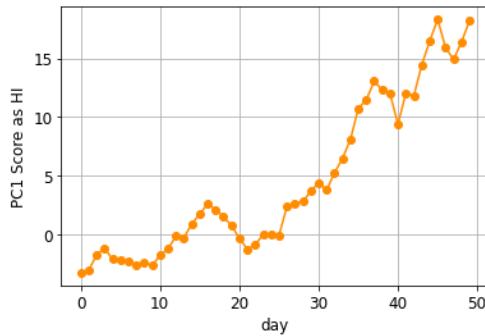
```
# perform PCA and extract scores along the first principal component
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler().fit(selectFeatures_train)
selectFeatures_train_normal = scaler.transform(selectFeatures_train)
selectFeatures_all_normal = scaler.transform(selectFeatures_all)

pca = PCA().fit(selectFeatures_train_normal)
PCA_all_scores = pca.transform(selectFeatures_all_normal)

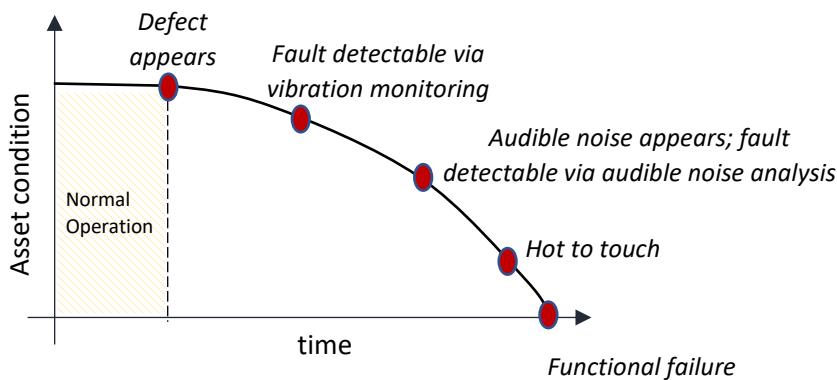
plt.figure(), plt.plot(PCA_all_scores[:,1],'-o', color='darkorange')
plt.xlabel('day'), plt.ylabel('PC1 Score as HI')
```

The first component explains almost 95% of the data variance and the plot below shows that the first principal component exhibits a nice monotonic trend as bearing fault progresses; therefore it is a good metric to be used as a health indicator.



P-F Curve

In the machinery reliability world, the P-F (potential failure) curve is a frequently used concept. The curve represents the progression of an equipment's health condition over time from a healthy state to initiation of defect to failure. The changes in machine's characteristics or failure symptoms such as high temperature, high vibration, high acoustic noise, etc., are specified on the curve which indicates the technique that may be effective in fault detection at any given stage of fault.



Hopefully, now you have a better understanding of what fault prognosis entails and the different methodologies available to implement predictive maintenance solutions.

Summary

In this chapter, we acquainted ourselves with the methodologies employed for predictive maintenance and undertook a detailed conceptual study of health indicators (HIs). Generation of HIs is one of the primary means for the estimation of RUL and therefore, we looked at the methods commonly used to compute health indicators. We consolidated the concepts covered by working through a real case study of HI generation for a wind turbine.

Chapter 20

Fault Prognosis: RUL Estimation

In the previous chapter, we introduced the concept of remaining useful life which is simply the time remaining until failure of an equipment. Three broad data-based techniques were mentioned that are: 1) reliability data-based approach wherein lifespan distribution of similar equipment is utilized to find the expected RUL 2) direct computation of RUL via regression-based ML modeling 3) computation of health indicator as an intermediate step. The first two approaches require information about the past lifespan of equipment and complete run-to-failure histories. However, it is difficult to get these data in process industry as very often machines get repaired before they reach failure stages (remember preventive maintenance!). This makes HI-based approach more suitable and, unsurprisingly, more popular. In the previous chapter, we saw how to compute HI for a wind turbine. We will take this case study to completion and show how to estimate the RUL.

Within the HI-based approach, two strategies are widely adopted. If decent amount of past run-to-failure data are available, then one can simply pick up the historical HI trend that matches the most with the current equipment's HI trajectory and use the historical lifespan to compute the required RUL. This is called similarity-based approach. A popular alternative is to simply use the existing HI values of current equipment and fit a curve to it to extrapolate it in the future and find when the failure threshold is breached. This is called degradation-based approach. We will go into more details into these two strategies in this chapter. Overall, the following topics are covered

- Introduction to RUL
- Health indicator-based RUL estimation strategies
- Health indicator degradation modeling for RUL estimation of a wind turbine
- Deep learning-based direct RUL estimation for a gas turbine

20.1 RUL: Revisited

In the previous chapter, we looked at some broad classes of strategies for RUL estimation. We also looked at how a health indicator can be calculated. Figure 20.1 reproduces Figure 19.1 and adds more details regarding HI-based approaches for RUL computation. The figure also highlights the four commonly employed strategies. As alluded to earlier, the choice of model depends on the type and amount of information available on past failures. If large amount of past run-to-failure data are available, then one can build a deep learning model to directly predict the RUL. We will see one such application in this chapter.

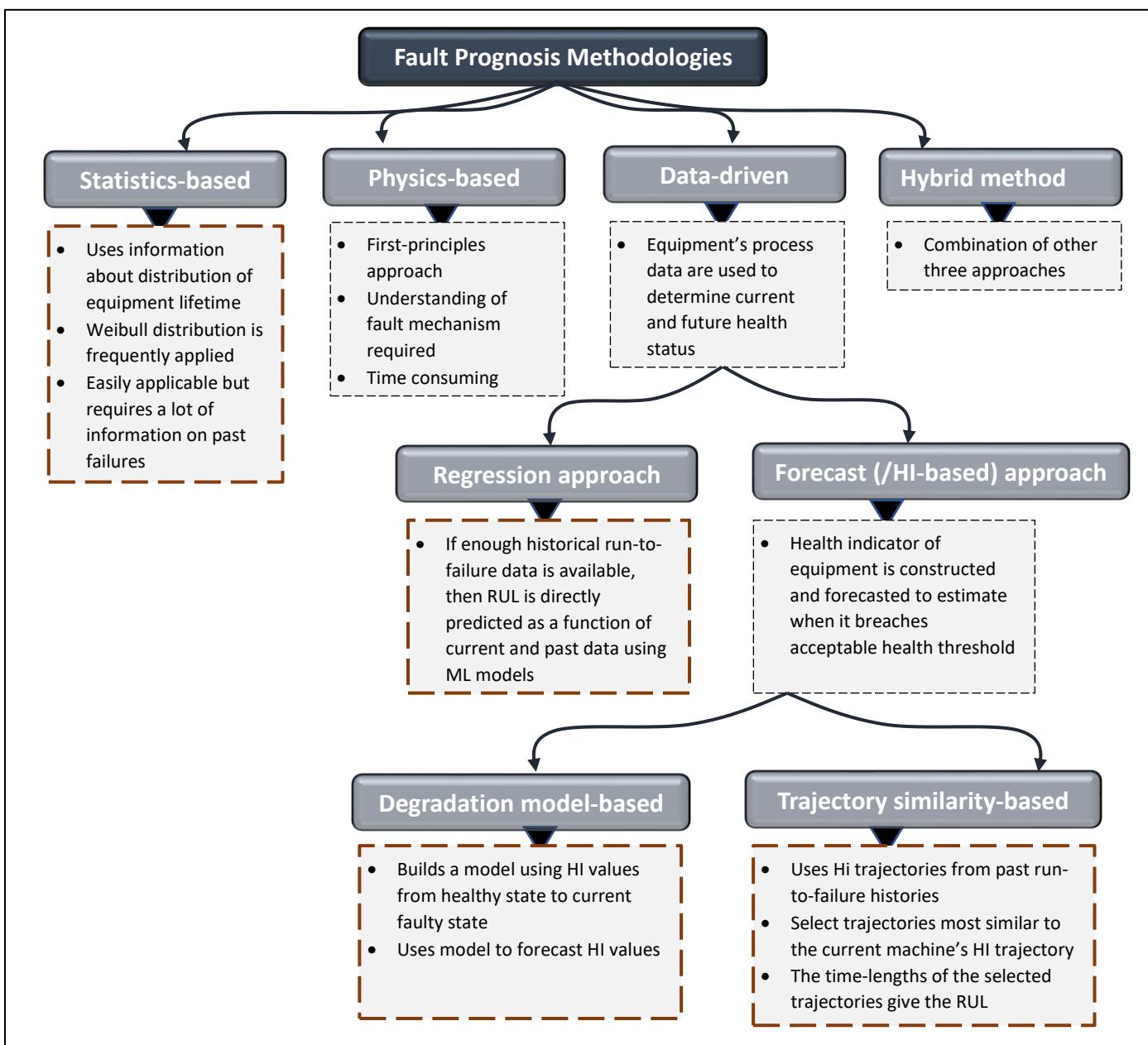


Figure 20.1: Prognostics Methodologies



While HI-based strategies are popular, a few points of caution need to be mentioned. For degradation-based modeling, you should ensure that either the impact of future operating conditions is taken into account or the operating conditions do not change much. For similarity-based approach, care should be taken that the historical HI trajectories are selected for the same faulty conditions as the fault that the current equipment is experiencing. Therefore, correct fault classification is crucial prior to deployment of fault prognosis.

RUL estimation performance assessment

Consider the figure below. At time t_k , degradation curve is extrapolated to predict RUL_{pred} while the actual RUL is $\text{RUL}_{\text{actual}}$. A trivial way to quantify error is to use mean squared error (MSE). However, the impact of positive and negative RUL errors on plant operations can be quite different. Underprediction ($\text{RUL}_{\text{pred}} < \text{RUL}_{\text{actual}}$) \Rightarrow maintenance takes place earlier than it should have been. While this is bad, the cost of overprediction ($\text{RUL}_{\text{pred}} > \text{RUL}_{\text{actual}}$) is worse as it can lead to delayed maintenance and, therefore, catastrophic equipment failure.

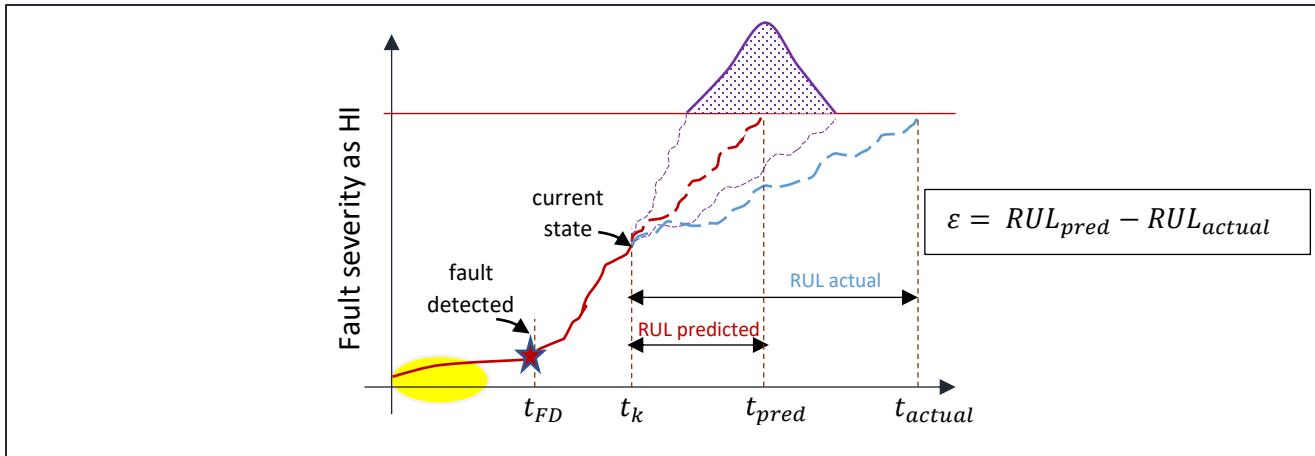


Figure 20.2: RUL: actual vs predicted

The preference for negative ε over positive ε is often specified by putting greater penalty for $\varepsilon > 0$ when computing the overall performance (for example, during model fitting).

20.2 Health Indicator-based RUL Estimation Strategies

In Figure 20.2, we saw that by the time instance t_k , HI already shows a degradation trend. For RUL estimation, the trick lies in figuring out how the HI evolves further until it reaches the failure threshold. Let's look at two popular ways of approaching this problem.

Health degradation modeling

In the degradation model approach, a HI model is fitted, as shown in the figure below, using the available HI data for the equipment under study. As shown, the model can take different forms. At time t_k , the model is used to predict t_F when the fitted HI curve breaches the failure threshold; this gives RUL prediction at time t_k as $t_F - t_k$. The model training and RUL estimation is redone at time t_{k+1} when becomes h_{k+1} available. We will apply this strategy to the wind turbine case-study introduced in the previous chapter.

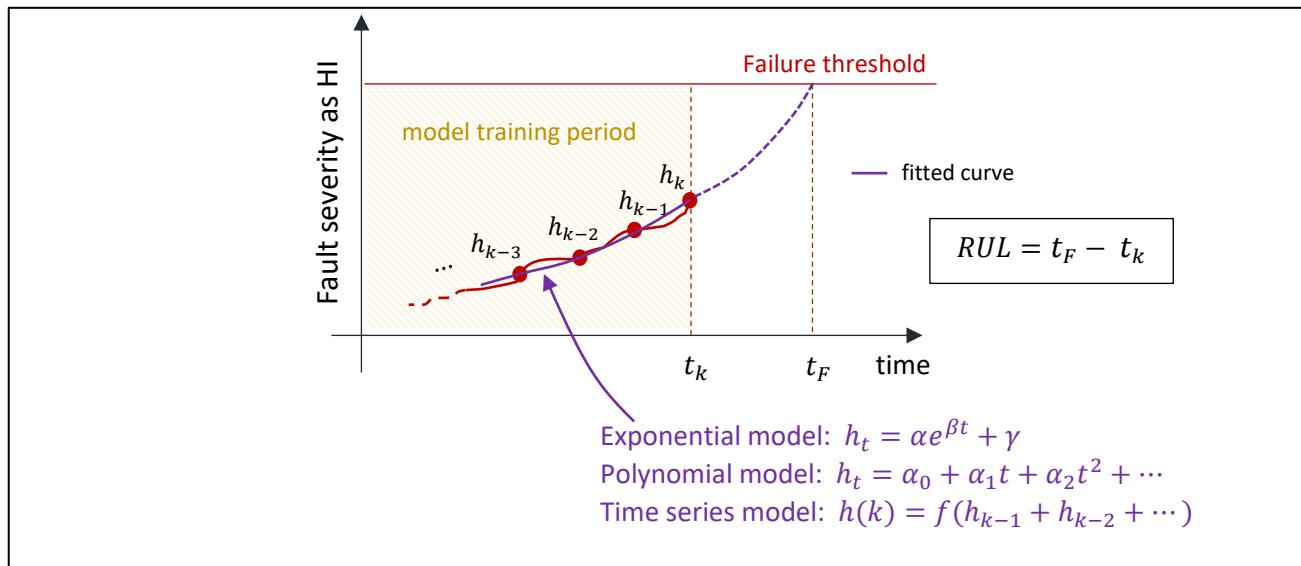


Figure 20.3: Degradation modeling-based RUL estimation

The selection of failure threshold is non-trivial. The threshold may be easy to specify if HI is based out of a single measurement or feature such as RMS of vibration signal or temperature of equipment. However, for a generic case, domain specific knowledge or historical failure records would be needed.

Trajectory similarity-based modeling

In similarity-based approach, known HI trends from past equipment failures are gathered as shown in Figure 20.4 below. At time t_k , the trajectories that are similar¹²⁴ to the current equipment's HI trajectory until time t_k are selected. The lifespans of these selected records are combined (simple average or weighted combination) to estimate the RUL of the current equipment.

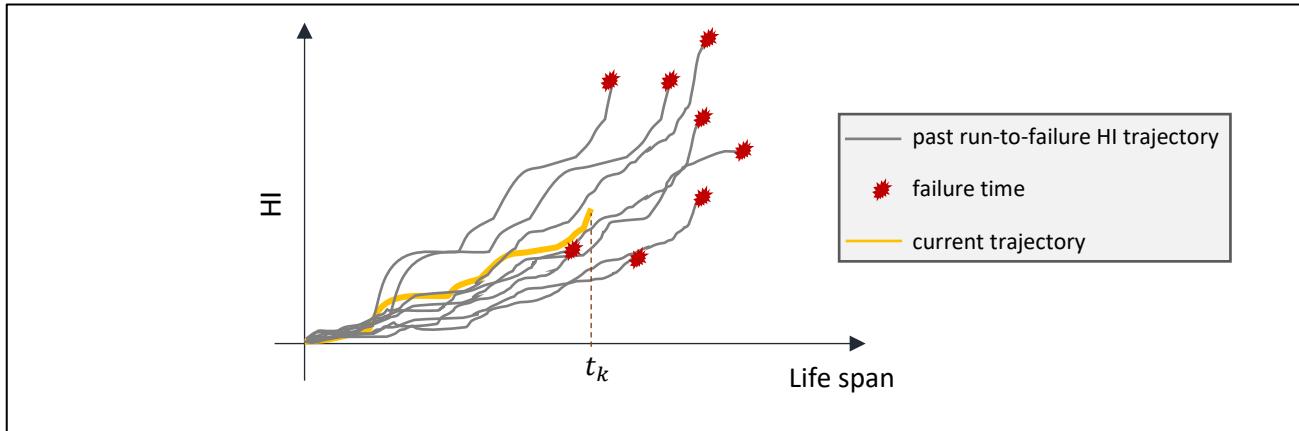


Figure 20.4: Trajectory similarity-based RUL estimation

20.3 RUL Estimation via Degradation Modeling for a Wind Turbine

To demonstrate degradation modeling-based approach, we will continue the case study of wind turbine from the previous chapter. We will attempt to fit an exponential model to the HI values obtained for the training period of 32 days.

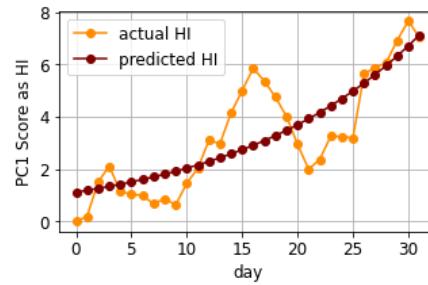
```
# fit exponential model (HI = a*exp(b*t) + c); t is in days
PCA_train_scores = pca.transform(selectFeatures_train_normal)
HI_train = PCA_train_scores[:, 0]
HI_train = HI_train - HI_train[0] # forcing HI to start from 0 for better fit

def func(t, a, b):
    return a*np.exp(b*t)
```

¹²⁴ Similarity between two trajectories are often quantified via Euclidean distance. Alternatively, dynamic time warping (DTW) has also been employed.

```
param_opt, param_cov = scipy.optimize.curve_fit(func, range(32), HI_train)

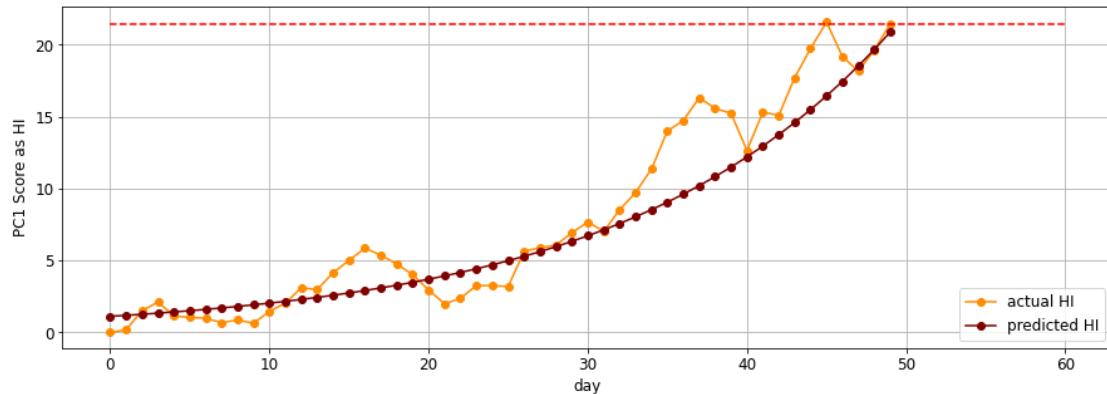
plt.figure(), plt.plot(range(32), HI_train, '-o', color='darkorange', label='actual HI')
plt.plot(range(32), func(np.arange(32), *param_opt), '-o', color='maroon', label='predicted HI')
plt.xlabel('day'), plt.ylabel('PC1 Score as HI'), plt.legend()
```



The actual HI trajectory seems to have significant stochastic drift characteristic in the training period and therefore, it is not surprising our deterministic exponential fit cannot closely follow the actual trajectory. Nonetheless, let's check out the error in RUL prediction. We will use the last value of the computed HI (day 50) as the failure threshold and estimate when the fitted trajectory crosses this threshold.

```
# forecast using the degradation model
HI_all = PCA_all_scores[:,0]
HI_all = HI_all - HI_all[0]
```

```
plt.figure(), plt.plot(range(50), HI_all, '-o', color='darkorange', label='actual HI')
plt.plot(range(50), func(np.arange(50), *param_opt), '-o', color='maroon', label='predicted HI')
plt.plot([0,60], [HI_all[-1],HI_all[-1]], '--', color='red')
plt.xlabel('day'), plt.ylabel('PC1 Score as HI'), plt.legend()
```



The above plot indicates that the simple exponential degradation model predicts the end of useful life perfectly!

20.4 RUL Estimation via ANN-based Regression Modeling for a Gas Turbine

For illustration of regression-based direct prediction of RUL using past run-to-failure histories, we will use simulated aircraft gas turbine engine dataset¹²⁵ which consists of operational and dynamic data (such as temperature, pressure) from multiple sensors from several engine operation simulations. Each simulation starts with an engine (with different degrees of initial wear) operating within normal limits. Engine degradation starts at some point during the simulation and continues until engine failure (engine health margin, a function of efficiency and flow, falling below a threshold). Training datasets contain complete data until engine failures, while the test dataset contains data until some point prior to failure. Actual RULs for the engines have been provided for the test dataset. Our objective is to develop a PdM model to predict engine failure using simulation data in the test dataset.

Aircraft engine dataset

We will use data in the following three files from the repository dataset.

- train_FD001.txt: Along with (3) operational settings, it contains timeseries data from 21 sensors (s1 to s21) from 100 different simulations or engines. The last cycle of each simulation is the point of failure of the corresponding engine. In Figure 20.5, sensor drift due to engine degradation is apparent.
- test_FD001.txt: It also contains simulated timeseries data for 100 engines, however, here only partial timeseries is provided for each engine. Figure 20.5 shows the provided timeseries for engine ID 90. The RUL of each engine needs to be estimated.
- RUL_FD001.txt: It contains the actual RULs of the 100 engines in the test dataset measured in cycles. We will use this information to measure our model predictions' accuracy.

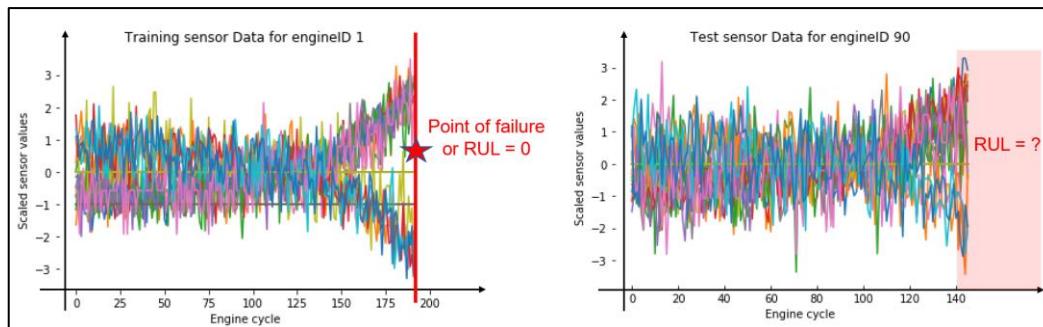


Figure 20.5: Sensor reading from training dataset (left) and test dataset (right). The actual RUL for the shown engine ID 90 is 28 as provided in the RUL_FD001.txt.

¹²⁵ This dataset (NASA Turbofan Jet Engine Data Set) is in public domain and is provided by NASA (https://data.nasa.gov/Aerospace/CMAPSS-Jet-Engine-Simulated-Data/ff5v-kuh6/about_data).

RUL prediction using LSTM

In this illustration, we will build a LSTM network to predict the RULs for the test engines. As done by others¹²⁶, we will use a sequence of 50 past cycles to make the failure prediction. Let's start with reading data from the provided text files. Note that, unlike previous illustrations, we will use Pandas library more extensively here due to the ability to reference columns by descriptive names which makes the code easier to understand.

```
# training data
train_df = pd.read_csv('PM_train.txt', sep=" ", header=None)
train_df.drop(train_df.columns[[26, 27]], axis=1, inplace=True) # last two columns are blank
train_df.columns = ['EngineID', 'cycle', 'OPsetting1', 'OPsetting2', 'OPsetting3', 's1', 's2', 's3',
's4', 's5', 's6', 's7', 's8', 's9', 's10', 's11', 's12', 's13', 's14', 's15', 's16', 's17', 's18', 's19', 's20', 's21']

# test data
test_df = pd.read_csv('PM_test.txt', sep=" ", header=None)
test_df.drop(test_df.columns[[26, 27]], axis=1, inplace=True)
test_df.columns = ['EngineID', 'cycle', 'OPsetting1', 'OPsetting2', 'OPsetting3', 's1', 's2', 's3',
's4', 's5', 's6', 's7', 's8', 's9', 's10', 's11', 's12', 's13', 's14', 's15', 's16', 's17', 's18', 's19', 's20', 's21']

# actual RUL for each engine-id in the test data
truth_df = pd.read_csv('PM_truth.txt', sep=" ", header=None)
truth_df.drop(truth_df.columns[[1]], axis=1, inplace=True) # 2nd column is blank
truth_df.columns = ['finalRUL'] # assigning column name as finalRUL
truth_df['EngineID'] = truth_df.index + 1 # adding new column EngineID
```

While most of the code above is self-explanatory, the last part reads the true RUL data, removes the redundant columns with 'nan' data, and adds an additional column 'EngineID' as shown below

Index	0	1
0	112	nan
1	98	nan
2	69	nan
3	82	nan



Index	finalRUL	EngineID
0	112	1
1	98	2
2	69	3
3	82	4

Next, we will do some dataframe manipulations to compute RUL at any given cycle for an engine. The code below basically adds the highlighted column shown in the snippet below to

¹²⁶ <https://github.com/umbertogriffo/Predictive-Maintenance-using-LSTM>

the training and test dataframes. We will also clip the RUL values in training dataset at the threshold 150.

Index	EngineID	cycle	OPsetting1	OPsetting2	OPsetting3	s1	s2	s21	engineRUL
159	1	160	-0.0006	-0.0004	100	518.67	643.45	23.2817	32
160	1	161	0.0008	0.0001	100	518.67	643	23.2962	31
161	1	162	-0.0005	0.0004	100	518.67	643.15	23.1538	30
162	1	163	0.0003	-0.0004	100	518.67	642.85	23.1419	29
163	1	164	0.0005	-0.0002	100	518.67	643.17	23.1761	28

```
# training dataset
maxCycle_df = pd.DataFrame(train_df.groupby('EngineID')['cycle'].max()).reset_index()
maxCycle_df.columns = ['EngineID', 'maxEngineCycle']

train_df = train_df.merge(maxCycle_df, on=['EngineID'], how='left')
train_df['engineRUL'] = train_df['maxEngineCycle'] - train_df['cycle']
train_df.drop('maxEngineCycle', axis=1, inplace=True) # maxEngineCycle is not needed anymore

# compute maxEngineCycle for test data using data from test_df and truth_df
maxCycle_df = pd.DataFrame(test_df.groupby('EngineID')['cycle'].max()).reset_index()
maxCycle_df.columns = ['EngineID', 'maxEngineCycle']
truth_df['maxEngineCycle'] = maxCycle_df['maxEngineCycle'] + truth_df['finalRUL']
truth_df.drop('finalRUL', axis=1, inplace=True)

# generate engineRUL for test data
test_df = test_df.merge(truth_df, on=['EngineID'], how='left')
test_df['engineRUL'] = test_df['maxEngineCycle'] - test_df['cycle']
test_df.drop('maxEngineCycle', axis=1, inplace=True)

# clip RUL in training data at the threshold 150
RULthreshold = 150
train_df['engineRUL'] = np.where(train_df['engineRUL'] > RULthreshold, 150, train_df['engineRUL'])
```

Next, we will scale the sensor readings.

```
# training dataset: create temporary dataframe with columns to be scaled
all_cols = train_df.columns # get columns names
cols_to_scale = train_df.columns.difference(['EngineID', 'cycle', 'engineRUL'])
train_df_with_cols_to_scale = train_df[cols_to_scale]
```

```

# scale and rejoin with columns that were not scaled
scaler = StandardScaler()
scaled_train_df_with_cols_to_scale = pd.DataFrame(scaler.fit_transform(train_df_with_cols_to_scale),
                                                    columns=cols_to_scale) # scalar transform returns a numpy array
train_df_scaled = train_df[['EngineID','cycle','engineRUL']].join(scaled_train_df_with_cols_to_scale)
train_df_scaled = train_df_scaled.reindex(columns = all_cols) # same columns order as before

# test dataset: repeat above steps
all_cols = test_df.columns
test_df_with_cols_to_scale = test_df[cols_to_scale]
scaled_test_df_with_cols_to_scale = pd.DataFrame(scaler.transform(test_df_with_cols_to_scale),
                                                 columns=cols_to_scale)
test_df_scaled = test_df[['EngineID','cycle','engineRUL']].join(scaled_test_df_with_cols_to_scale)
test_df_scaled = test_df_scaled.reindex(columns = all_cols) # same columns order as before

```

Next, we will create the sequence samples. For each engine, any continuous block of 50 cycles forms a sequence. To accomplish this, we will define a utility function as shown below

```

nSequenceSteps = 50 # number of cycles in a sequence
X_train_sequence = []
y_train_sequence = []

# define utility function
def generate_LSTM_samples(engine_df, nSequenceSteps):
    """
    This function generates list of LSTM samples (numpy arrays of size (nSequenceSteps, 24)
    each) for LSTM input and list of output labels for LSTM
    """

    engine_X_train_sequence = []
    engine_y_train_sequence = []
    engine_data = engine_df.values # converting to numpy

    for sample in range(nSequenceSteps, engine_data.shape[0]):
        engine_X_train_sequence.append(engine_data[sample-nSequenceSteps:sample, :-1]) # last
                                         # column is output label
        engine_y_train_sequence.append(engine_data[sample,-1])

    return engine_X_train_sequence, engine_y_train_sequence

# generate sequence samples
for enginID in train_df_scaled['EngineID'].unique():
    engine_df = train_df_scaled[train_df_scaled['EngineID'] == enginID]

```

```

engine_df = engine_df[['OPsetting1', 'OPsetting2', 'OPsetting3', 's1', 's2', 's3', 's4', 's5', 's6', 's7',
                      's8', 's9', 's10', 's11', 's12', 's13', 's14', 's15', 's16', 's17', 's18', 's19', 's20',
                      's21', 'engineRUL']]
engine_X_train_sequence, engine_y_train_sequence = generate_LSTM_samples(engine_df,
nSequenceSteps)
X_train_sequence = X_train_sequence + engine_X_train_sequence # adding samples to the
common list
y_train_sequence = y_train_sequence + engine_y_train_sequence

X_train_sequence, y_train_sequence = np.array(X_train_sequence), np.array(y_train_sequence)

```

We are now ready to build and compile our RNN. The topology includes a single neuron output layer with linear activation function. For regularization, we utilize the dropout technique.

```

# import ANN packages
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout

# custom metric
import tensorflow.keras.backend as K
def r2_custom(y_true, y_pred):
    """Coefficient of determination
    """
    SS_res = K.sum(K.square(y_true - y_pred))
    SS_tot = K.sum(K.square(y_true - K.mean(y_true)))
    return (1 - SS_res/(SS_tot + K.epsilon()))

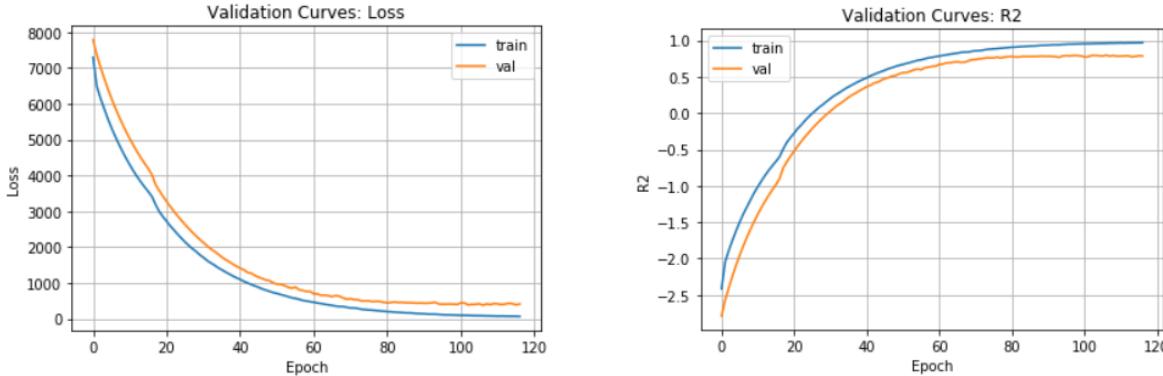
# define model
model = Sequential()
model.add(LSTM(units=100, return_sequences=True, input_shape=(nSequenceSteps, 24)))
model.add(Dropout(0.2))
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(1))

# compile model
model.compile(loss='mse', optimizer='Adam', metrics= r2_custom)

```

We can now fit the model. Validation curves indicate that we can obtain pretty good predictions on training dataset.

```
from tensorflow.keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
history = model.fit(X_train_sequence, y_train_sequence, epochs=200, batch_size=250,
                     validation_split=0.3, callbacks=[es])
```



For test data, we will create one sequence per engine using the last 50 cycles. Figure 20.6 compares the actual versus predicted RUL values for the test engine dataset. It is apparent that the model performs satisfactorily well. Such accurate estimation of remaining useful life of process equipment can be a great assistance in maintenance planning and avoidance of costs due to unexpected equipment failures.

```
# input/output test sequences (only the last sequence is used to predict failure)
X_test_sequence = []
y_test_sequence = []

for enginID in test_df_scaled['EnginID'].unique():
    engine_df = test_df_scaled[test_df_scaled['EnginID'] == enginID]

    if engine_df.shape[0] >= nSequenceSteps:
        engine_df = engine_df[['OPsetting1', 'OPsetting2', 'OPsetting3', 's1', 's2', 's3', 's4', 's5', 's6', 's7',
                               's8', 's9', 's10', 's11', 's12', 's13', 's14', 's15', 's16', 's17', 's18', 's19', 's20',
                               's21', 'engineRUL']].values
        X_test_sequence.append(engine_df[-nSequenceSteps:, :-1])
        y_test_sequence.append(engine_df[-1,-1])

X_test_sequence, y_test_sequence = np.array(X_test_sequence), np.array(y_test_sequence)

# predict RULs
y_test_sequence_pred = model.predict(X_test_sequence)
test_performance = model.evaluate(X_test_sequence, y_test_sequence)
print('R2_test: {}'.format(test_performance[1]))
```

>>> R2_test: 0.834

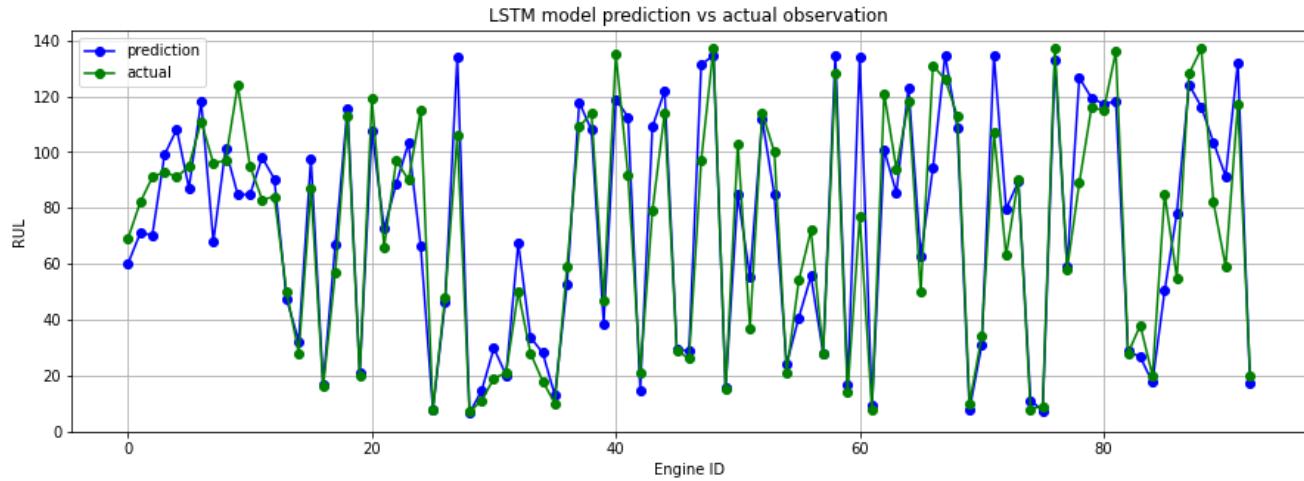


Figure 20.6: Predicted vs observed engine RULs for test aircraft engine dataset

This completes our brief coverage of predictive maintenance methodologies. You now have all the modern tools in your arsenal to tackle plant health management problems in your data production plants. We bid adieu to you and wish you all the best in your process data science career.

Summary

In this chapter, we built upon the fault prognosis foundations laid in the previous chapter. We looked at techniques for RUL prediction using health indicator trajectory and direct regression. We consolidated our conceptual understanding through a couple of case studies on RUL predictions for wind turbine and gas turbine.

End of the book



Machine Learning in Python for Process and Equipment Condition Monitoring, and Predictive Maintenance

This book is designed to help readers quickly gain a working-level knowledge of machine learning-based techniques that are widely employed for building equipment condition monitoring, plantwide monitoring, and predictive maintenance solutions in process industry. The book covers a broad spectrum of techniques ranging from univariate control charts to deep-learning-based prediction of remaining useful life. Consequently, the readers can leverage the concepts learned to build advanced solutions for fault detection, fault diagnosis, and fault prognostics. The application-focused approach of the book is reader friendly and easily digestible to the practicing and aspiring process engineers, and data scientists. Upon completion, readers will be able to confidently navigate the Prognostics and Health Management literature and make judicious selection of modeling approaches suitable for their problems.

The following topics are broadly covered:

- *Exploratory analysis of process data*
- *Best practices for process monitoring and predictive maintenance solutions*
- *Univariate monitoring via control charts and time-series data mining*
- *Multivariate statistical process monitoring techniques (PCA, PLS, FDA, etc.)*
- *Machine learning and deep learning techniques to handle dynamic, nonlinear, and multimodal processes*
- *Fault detection and diagnosis of rotating machinery using vibration data*
- *Remaining useful life predictions for predictive maintenance*