



Machine Learning in Python for Process Systems Engineering

Achieve operational excellence using process data

2024 Edition

Ankur Kumar, Jesus Flores-Cerrillo

Machine Learning in Python for Process Systems Engineering

Achieve operational excellence using process data

**Ankur Kumar
Jesus Flores-Cerrillo**

Dedicated to our spouses, family, friends, motherland, and all the data-science enthusiasts

Machine Learning in Python for Process Systems Engineering

www.MLforPSE.com



Copyright © 2022 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.

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

First published: February 2022
Revision 1 published: June 2023
Revision 2 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.

Most recently, 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_PSE. Code updates, when necessary, will be made and updated on the GitHub repository. Updates to the book's text material will be made 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.

Other book(s) from the series
(<https://MLforPSE.com/books/>)

Book 2



Machine Learning in Python for Dynamic Process Systems

A practitioner's guide for building process modeling, predictive, and monitoring solutions using dynamic data



First Edition

Ankur Kumar, Jesus Flores-Cerrillo

Book 3

ML for Process Industry Series

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

Table of Contents

• Preface	
• Part 1 Introduction and Fundamentals	
• Chapter 1 Machine Learning for Process Systems Engineering	1
○ 1.1 What are Process Systems	
■ 1.1.1 Characteristics of process data	
○ 1.2 What is Machine Learning	
■ 1.2.1 Machine learning workflow	
■ 1.2.2 Type of machine learning systems	
○ 1.3 Machine Learning Applications in Process Industry	
■ 1.3.1 Decision hierarchy levels in a process plant	
■ 1.3.2 Application areas	
○ 1.4 ML Solution Deployment	
○ 1.5 The Future of Process Data Science	
• Chapter 2 The Scripting Environment	20
○ 2.1 Introduction to Python	
○ 2.2 Introduction to Spyder and Jupyter	
○ 2.3 Python Language: Basics	
○ 2.4 Scientific Computing Packages: Basics	
■ 2.4.1 Numpy	
■ 2.4.2 Pandas	
○ 2.5 Typical ML Script	
• Chapter 3 Machine Learning Model Development: Workflow and Best Practices	39
○ 3.1 ML Model Development Workflow	
○ 3.2 Data Pre-processing: Data Transformation	
■ 3.2.1 (Robust) Data centering & scaling	
■ 3.2.2 Feature extraction	
■ 3.2.3 Feature engineering	
■ 3.2.4 Workflow automation via pipelines	
○ 3.3 Model Evaluation	
■ 3.3.1 Regression metrics	
■ 3.3.2 Classification metrics	
■ 3.3.3 Holdout method / cross-validation	
■ 3.3.4 Residual analysis	
○ 3.4 Model Tuning	
■ 3.4.1 Overfitting & underfitting	
■ 3.4.2 Train/validation/test split	
■ 3.3.3 K-fold cross-validation	
■ 3.4.4 Regularization	
■ 3.4.5 Hyperparameter optimization via GridSearchCV	
• Chapter 4 Data Pre-processing: Cleaning Process Data	67
○ 4.1 Signal De-noising	
■ 4.1.1 Moving window average filter	
■ 4.1.2 SG filter	

○	4.2 Variable Selection/Feature Selection <ul style="list-style-type: none"> ▪ 4.2.1 Filter methods ▪ 4.2.2 Wrapper methods ▪ 4.2.3 Embedded methods 	
○	4.3 Outlier Handling <ul style="list-style-type: none"> ▪ 4.3.1 Univariate methods ▪ 4.3.2 Multivariate methods ▪ 4.3.3 Data-mining methods 	
○	4.4 Handling Missing Data	
•	Part 2 Classical Machine Learning Methods	
•	Chapter 5 Dimension Reduction and Latent Variable Methods (Part 1)	90
○	5.1 PCA: An Introduction <ul style="list-style-type: none"> ▪ 5.1.1 Mathematical background ▪ 5.1.2 Dimensionality reduction for polymer manufacturing process 	
○	5.2 Process Monitoring via PCA for Polymer Manufacturing Process <ul style="list-style-type: none"> ▪ 5.2.1 Process monitoring/fault detection indices ▪ 5.2.2 Fault detection ▪ 5.2.3 Fault diagnosis 	
○	5.3 Variants of Classical PCA <ul style="list-style-type: none"> ▪ 5.3.1 Dynamic PCA ▪ 5.3.2 Multiway PCA ▪ 5.3.3 Kernel PCA 	
○	5.4 PLS: An Introduction <ul style="list-style-type: none"> ▪ 5.4.1 Mathematical background 	
○	5.5 Soft Sensing via PLS for Pulp & Paper Manufacturing Process	
○	5.6 Process monitoring via PLS for Polyethylene Manufacturing Process <ul style="list-style-type: none"> ▪ 5.6.1 Fault detection indices ▪ 5.6.2 Fault detection 	
○	5.7 Variants of Classical PLS	
•	Chapter 6 Dimension Reduction and Latent Variable Methods (Part 2)	120
○	6.1 ICA: An Introduction <ul style="list-style-type: none"> ▪ 6.1.1 Mathematical background ▪ 6.1.2 Complex chemical process: Tennessee Eastman Process ▪ 6.1.3 Deciding number of ICs 	
○	6.2 Process Monitoring via ICA for Tennessee Eastman Process <ul style="list-style-type: none"> ▪ 6.2.1 Fault detection indices ▪ 6.2.2 Fault detection 	
○	6.3 FDA: An Introduction <ul style="list-style-type: none"> ▪ 6.3.1 Mathematical background ▪ 6.3.2 Dimensionality reduction for Tennessee Eastman Process 	
○	6.4 Fault Classification via FDA for Tennessee Eastman Process	
•	Chapter 7 Support Vector Machines & Kernel-based Learning	142
○	7.1 SVMs: An Introduction <ul style="list-style-type: none"> ▪ 7.1.1 Mathematical background ▪ 7.1.2 Hard margin vs soft margin classification 	
○	7.2 The Kernel Trick for Nonlinear Data <ul style="list-style-type: none"> ▪ 7.2.1 Mathematical background 	
○	7.3 SVDD: An Introduction	

<ul style="list-style-type: none"> ▪ 7.3.1 Mathematical background ▪ 7.3.2 OC-SVM vs SVDD ▪ 7.3.3 Bandwidth parameter and SVDD illustration ○ 7.4 Process Fault Detection via SVDD ○ 7.5 SVR: An Introduction <ul style="list-style-type: none"> ▪ 7.5.1 Mathematical background ○ 7.6 Soft Sensing via SVR in a Polymer Processing Plant ○ 7.7 Soft Sensing via SVR for Debutanizer Column in a Petroleum Refinery 	
<ul style="list-style-type: none"> • Chapter 8 Finding Groups in Process Data: Clustering & Mixture Modeling 	170
<ul style="list-style-type: none"> ○ 8.1 Clustering: An Introduction <ul style="list-style-type: none"> ▪ 8.1.1 Multimode semiconductor manufacturing process ○ 8.2 Centroid-based Clustering: K-Means <ul style="list-style-type: none"> ▪ 8.2.1 Determining the number of clusters via elbow method ▪ 8.2.2 Silhouette analysis for quantifying clusters quality ▪ 8.2.3 Pros and cons ○ 8.3 Density-based Clustering: DBSCAN <ul style="list-style-type: none"> ▪ 8.3.3 Pros and cons ○ 8.4 Probabilistic Clustering: Gaussian Mixtures <ul style="list-style-type: none"> ▪ 8.4.1 Mathematical background ▪ 8.4.2 Determining the number of clusters ○ 8.5 Multimode Process Monitoring via GMM for Semiconductor Manufacturing Process <ul style="list-style-type: none"> ▪ 8.5.1 Fault detection indices ▪ 8.5.2 Fault detection 	
<ul style="list-style-type: none"> • Chapter 9 Decision Trees & Ensemble Learning 	192
<ul style="list-style-type: none"> ○ 9.1 Decision Trees: An Introduction <ul style="list-style-type: none"> ▪ 9.1.1 Mathematical background ○ 9.2 Random Forests: An Introduction <ul style="list-style-type: none"> ▪ 9.2.1 Mathematical background ○ 9.3 Soft Sensing via Random Forest in Concrete Construction Industry <ul style="list-style-type: none"> ▪ 9.3.1 Feature importances ○ 9.4 Introduction to Ensemble Learning <ul style="list-style-type: none"> ▪ 9.4.1 Bagging ▪ 9.4.2 Boosting ○ 9.5 Effluent Quality Prediction in Wastewater Treatment Plant via XGBoost 	
<ul style="list-style-type: none"> • Chapter 10 Other Useful Classical ML Techniques 	214
<ul style="list-style-type: none"> ○ 10.1 KDE: An Introduction <ul style="list-style-type: none"> ▪ 10.1.1 Mathematical background ▪ 10.1.2 Deciding KDE hyperparameters ○ 10.2 Determining Monitoring Metric Control Limit via KDE ○ 10.3 KNN: An Introduction <ul style="list-style-type: none"> ▪ 10.3.1 Mathematical background ▪ 10.3.2 Deciding k-NN hyperparameters ▪ 10.3.3 Applications of k-NN for process systems ○ 10.4 Process Fault Detection via k-NN for semiconductor Manufacturing Process ○ 10.5 Combining ML Techniques 	

• Part 3 Artificial Neural Networks & Deep Learning	
• Chapter 11 Feedforward Neural Networks	230
○ 11.1 ANN: An Introduction	
■ 11.1.1 Deep learning	
■ 11.1.2 TensorFlow	
○ 11.2 Process Modeling via FFNN for Combined Cycle Power Plant	
○ 11.3 Mathematical Background	
■ 11.3.1 Activation functions	
■ 11.3.2 Loss functions & cost functions	
■ 11.3.3 Gradient descent optimization	
■ 11.3.4 Epochs & batch-size	
■ 11.3.5 Backpropagation	
■ 11.3.6 Vanishing/Exploding gradients	
○ 11.4 Nonlinearity in Neural Nets (Width vs Depth)	
○ 11.5 Neural Net Hyperparameter Optimization	
○ 11.6 Strategies for Improved Network Training	
■ 11.6.1 Early stopping	
■ 11.6.2 Regularization	
■ 11.6.3 Initialization	
■ 11.6.4 Batch normalization	
○ 11.7 Soft Sensing via FFNN for Debutanizer Column in a Petroleum Refinery	
○ FFNN Modeling Guidelines	
• Chapter 12 Recurrent Neural Networks	256
○ 12.1 RNN: An Introduction	
■ 12.1.1 RNN outputs	
■ 12.1.2 LSTM networks	
○ 12.2 System Identification via LSTM RNN for SISO Heater System	
○ 12.3 Mathematical Background	
○ 12.4 Stacked/Deep RNNs	
○ 12.5 Fault Classification vis LSTM for Tennessee Eastman Process	
○ 12.6 Predictive Maintenance using LSTM Networks	
■ 12.6.1 Failure prediction using LSTM	
■ 12.6.2 Remaining useful life (RUL) prediction using LSTM	
• Chapter 13 Reinforcement Learning	285
○ 13.1 Reinforcement Learning: An Introduction	
■ 13.1.1 RL for process control	
○ 13.2 RL Terminology & Mathematical Concepts	
■ 13.2.1 Environment and Markov decision process	
■ 13.2.2 Reward and return	
■ 13.2.3 Policy	
■ 13.2.4 Value function	
■ 13.2.5 Bellman equation	
○ 13.3 Fundamentals of Q-learning	
○ 13.4 Deep RL & Actor-Critic Framework	
■ 13.4.1 Deep Q-learning	
■ 13.4.2 Policy gradient methods	
■ 13.4.3 Actor-Critic framework	
○ 13.5 Deep Deterministic Policy Gradient (DDPG)	
■ 13.5.1 Replay memory	

- 13.5.2 Target networks
 - 13.5.3 OU process as exploration noise
 - 13.6 DDPG RL Agent as Level Controller
- **Part 4 Deploying ML Solutions Over Web**
 - **Chapter 14 Process Monitoring Web Application**

318

Appendix

- **Dataset Descriptions**

330

- Polymer Manufacturing Process Data
- Pulp & Paper Manufacturing Process Data
- Low-Density Polyethylene (LDPE) Process Data
- Tennessee Eastman Process Data
- Semiconductor Manufacturing Process Data
- Polymer Pilot Plant Data
- Debutanizer Column Data from a Petroleum Refinery
- Concrete Compressive Strength Data
- Wastewater Treatment Plant Data
- Combined Cycle Power Plant data
- SISO Heater System Data
- Gas Turbine Data

Preface

Everyday we hear stories about new feats being achieved by machine learning (ML) and artificial intelligence (AI) researchers that have the potential to revolutionize our world. Through humanoid robots, speech recognition, computer vision, driverless cars, fraud detection, personalized recommendations, and automated health diagnosis, machine learning has already become an integral part of our daily life. Moreover, away from the glitz of these ‘visible’ high-tech products, machine learning has also been making silent advances in process industries (chemical industry, biopharma industry, steel industry, etc.) where ML-based solutions are being increasingly used for predictive equipment maintenance, product quality assurance, process monitoring, fault diagnosis, process control, and process optimization. With increasing global competition and stricter product quality standards, industrial plants are relying upon machine learning tools (such as reinforcement-learning-based auto-adaptive process controller) to provide them the winning edge over competitors.

Perhaps you are reading this book because you too have been inspired by the capabilities of machine learning and would like to use it to solve problems being faced by your organization. However, you might be struggling to find a definite guide that can help you decide which specific methodology to choose among the myriad of available methodologies. You may have come across a nice research article that showcases an interesting process systems application of a ML method. However, you might be facing difficulties trying to understand the intricate details of the algorithm. We won’t be surprised if you have struggled to find a data-science book that caters to the needs of a process systems engineer, considers unique characteristics of industrial process systems, and uses industrial-scale process systems for illustrations. We, the authors, have been in that phase. A process engineer will arguably find it more relevant and useful to learn principal component analysis (PCA) by working through a process monitoring application (the most popular application area of PCA in process industry) and learning how to compute the monitoring metrics. Similar arguments could be made for several other popular ML methods. There is a gap in available machine learning resources for industrial practitioners and this book attempts to cover this gap.

In one sense, we wrote this book for our younger selves; a book that we wish had existed when we started experimenting with machine learning techniques. Drawing from our years of experience in developing data-driven industrial solutions, this book has been written with the focus on de-cluttering the world of machine learning, giving a comprehensive exposition of ML tools that have proven useful in process industry, providing step-by-step elucidation of implementation details, cautioning against the

pitfalls and listing various tips & tricks that we have encountered over the years, and using dataset from industrial-scale process systems for illustrations. We strongly believe in 'learning by doing' and therefore we encourage the readers to work through in-chapter illustrations as they follow along the text. For reader's assistance, Jupyter notebooks with complete code implementations are available for download. We have chosen Python as the coding language for the book as it convenient to use, has large collection of ML libraries, and is the de facto standard language for ML. No prior experience with Python is assumed. The book has been designed to teach machine learning from scratch and upon completion, the reader will feel comfortable at using ML techniques.

Machine learning will continue to play significant role in unleashing the next wave of productivity improvements in process industry. We hope that this book will inspire its readers to develop novel ML solutions to challenging problems at work. We wish all the best to the budding process systems data scientist.

Who should read this book

This book provides a comprehensive step-by-step exposition of several popular machine learning techniques that have proven useful in process industry. Industry-relevant illustrations have been included in a 'learn-by-doing' format. Appropriate references to advanced treatment of specific topics are provided in each chapter. Therefore, the book will be useful to process technologists from industry, undergraduate and graduate students, ML researchers, as well as a general data-science enthusiast. If you belong to any of the following categories, you will enjoy reading this book.

- 1) Process systems engineers or data scientists seeking to hone their ML skills
- 2) Process systems engineers looking for ML solutions to specific problems (such as monitoring a high-dimensional multimode process)
- 3) Budding process systems data scientists taking their first step into the ML world
- 4) Budding data scientists (with no Python experience) looking to learn Python by working with real industrial datasets
- 5) Process systems engineers who frequently use commercial data-science software (IBM SPSS, Aspen Mtell) and are interested in learning the technical details for an improved understanding of the methodologies.

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. Every methodology is demonstrated by using simple process system or numerical example to illustrate major characteristics of the method and then by implementing on industrial-scale processes.

The book has been divided into four parts. **Part 1** provides a perspective on the importance of ML in process systems engineering and lays down the basic foundations of ML. **Part 2** provides in-detail presentation of classical ML techniques and has been written keeping in mind the various characteristics of industrial process systems such as high-dimensionality, non-linearity, multimode operations, etc. **Part 3** is focused on artificial neural networks and deep learning. While deep learning is the current buzzword in ML community, we would like to caution the reader against the temptation to deploy a deep learning model for every problem at hand. Often, simpler classical models can provide as good, if not better, results as those from neural net models. For example, partial least squares (PLS) are still the most popular models for soft sensor development in process industry due to its simplicity and powerful capabilities of handling noisy and correlated data. **Part 4** covers the important topic of deploying an ML solution over web.

It was a deliberate decision to not divide the book in terms of supervised / unsupervised / reinforcement-learning categories or application areas (process modeling, monitoring, etc.). This is because several methods overlap these categories which make it difficult to put them under a specific category. For example, SVM and SVR methods fall under supervised category while the related SVDD method falls under unsupervised category. Similar situation holds for PCA/PCR/PLS methods. A reader who is interested in a specific application area may use the table of contents as a guide to relevant sections in Parts 2 and 3. Care has been taken to title the subsections in different chapters appropriately.

Symbol notation

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

- lower-case, bold-face letters refer to vectors ($\mathbf{x} \in \mathbb{R}^{m \times 1}$) and upper-case, bold-face letters denote matrices ($\mathbf{X} \in \mathbb{R}^{n \times m}$)
- 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, bold-faced letter ($\mathbf{x}_i \in \mathbb{R}^{m \times 1}$)

Part 1

Introduction & Fundamentals

Chapter 1

Machine Learning for Process Systems Engineering

Imagine yourself in the shoes of an operator or engineer responsible for uninterrupted and optimal operation of an oil refinery. Keeping an eye on each of the 1000s of process measurements being made every second to look for process abnormalities or opportunities for plant performance improvement is akin to finding a needle in a haystack. The task is undoubtedly overwhelming and is the primary reason why plant managers often complain about having '*too much data but little knowledge and insight*'.

However, unlike humans, computers can be programmed to parse through large amounts of data in real-time, extract patterns, trends, and assist plant personnel in making informed business and operational decisions. This practice of learning about systems from data or machine learning has become an indispensable tool in process operations in the age of increasing global competition and stricter product quality standards.

This chapter provides an overview of how the power of machine learning is harnessed for process systems engineering. Specifically, the following topics are covered

- Unique characteristics of process data
- Types of ML systems and typical workflow to convert data into insights
- Classical applications of ML techniques in process industry
- Common ML solution deployment infrastructure employed in industry

Let's now tighten our seat-belts as we embark upon this exciting journey of de-mystifying machine learning for process systems engineering.

1.1 What are Process Systems

Process systems refer to a collection of physical structures that convert raw materials (wood, natural gas, crude oil, coal, etc.) into final consumer products (paper, fertilizers, diesel, energy, etc.) or intermediate products which are then used to manufacture other commodity materials. These process systems can range from a simple water-heating system to complex oil refineries. Figure 1.1 shows an example (petrochemical) plant comprising several processing units such as distillation columns, heat exchangers, pumps. Process industry is a broad category that encompasses, amongst others, chemical industry, bioprocess industry, power industry, pharmaceuticals industry, steel industry, semiconductor industry, and waste management industry.



Figure 1.1: Petrochemical plant image (obtained from pixabay.com)

In process industry, the task of optimizing production efficiency, controlling product quality, monitoring the process are categorized as process systems engineering (PSE) activities. These tasks often require a mathematical model of the plant. The traditional practice has been to use first principles mathematical description of physio-chemical phenomena occurring inside each process unit to mathematically characterize the whole plant. However, as you may already know, building such fundamental models are time-consuming and difficult for complex systems. Machine learning (ML)-based methods provide a convenient alternative where process data are used to build empirical plant model which can be used for optimization, control, and monitoring purposes. Availability of large amount of sensor data has further boosted the trend of incorporating ML techniques for PSE and demand for process data scientists.

Let's take a look at the kind of data generated in process industry. Majority of the process data include process stream flowrate, temperature, pressure, level, power, and composition measurements as shown in figure 1.2. Additionally, obtaining vibration signals from rotating equipment (motors, compressors), infrared or visual images, and spectra data are also common now-a-days. These indirect data are frequently utilized for predictive equipment maintenance and product quality control.

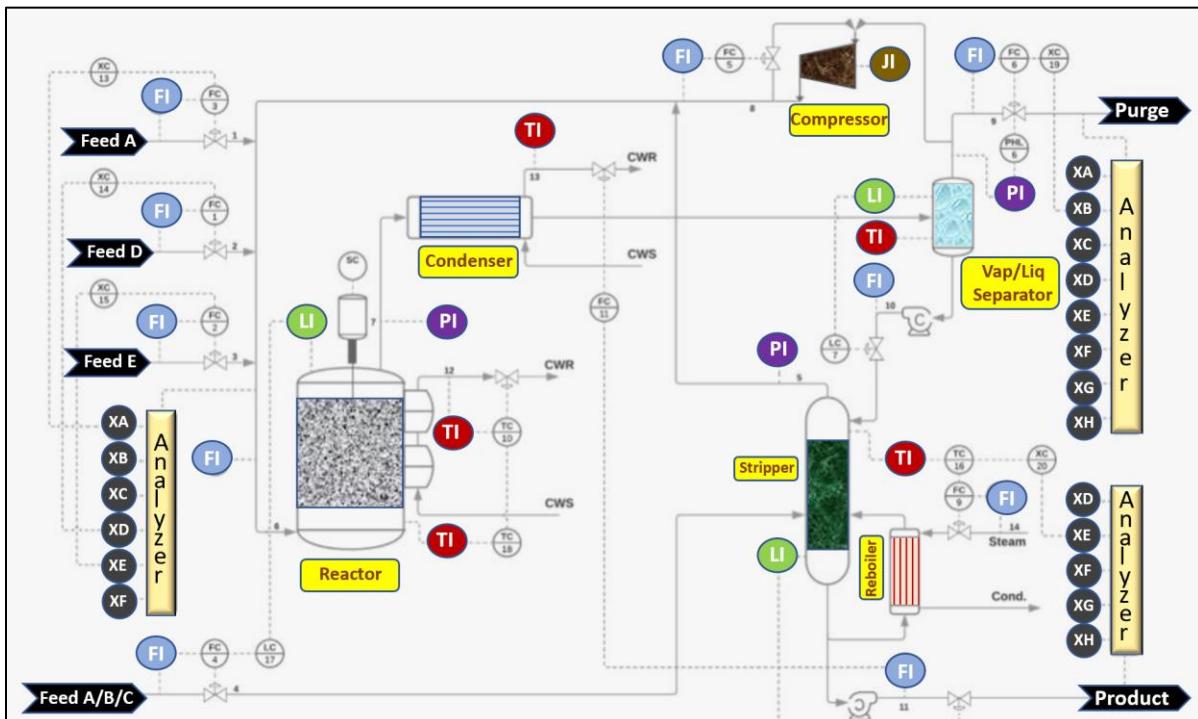


Figure 1.2: A process flowsheet¹ with typical flow (FI), temperature (TI), pressure (PI), composition (Analyzers), level (LI), power (JI) measurements.

Characteristics of process data

Industrial process data often exhibit characteristics which pose challenges to a process data scientist. Choosing an ML technique appropriate to the data is a pre-requisite for a successful project implementation. These characteristics include the following:

- **Dynamic:** Process plants rarely operate at a perfect steady-state i.e., at fixed values of process inputs and outputs. Plants are often subject to random or systematic disturbances such as changing ambient conditions, process feed quality, and product demand which are handled by control system by manipulating process variables. This leads to fluctuations around a steady-state point or plants moving from one steady-state to another.

¹ 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/>).

- *Time-varying*: Correlations between process variables change over time due to gradual changes in process parameters. For example, heat transfer coefficient in a heat-exchanger may change due to fouling or catalyst activity may degrade due to aging.
- *Batch vs continuous*: While continuous processes are dominant in process industry, batch processes are also frequently employed, for example in semiconductor and drug manufacturing. Unlike continuous processes, batch processes are of finite durations and offer its own set of unique challenges such as batch-to-batch variations and inherent non-steady-state operation. Time itself becomes a crucial dimension for accurate modeling of temporal evolution of process variables within a batch.
- *Multimode operations*: Multimode characteristics show up when a process plant operates primarily around a few distinct steady states. For example, a plant producing species a, b, and c may have different production recipes (set-points of process variables) for each of these recipes. Depending on the production schedule, the plant will switch from one recipe to another. Deploying a single global ML model for a multimode process often leads to lower accuracy compared to that obtained from building local models for each mode.
- *Discrete/Discontinuous*: Depending upon the production load, certain equipment may be switched on or off causing step changes in process behavior. For example, switching on an extra turbine or compressor causes step increase in power consumption.
- *Nonlinear*: The complex physio-chemical phenomena (chemical reactions, vapor-liquid equilibrium, etc.) occurring inside process units are most often nonlinear. While linear models often provide good approximations when processes operate around a single steady-state, non-linear models become necessary when processes experience large fluctuations in the operating conditions.
- *High dimensionality*: Modern process plants make hundreds of process-critical measurements. Considering that process plants make process adjustments in real-time, high dimensionality becomes an issue when a computationally intensive methodology is adopted.
- *Multirate sampling*: Not all process measurements are made at the same frequency. While temperature or pressure readings are sampled every second or minute, composition/analyzer measurements are often sampled at much lower frequency (once an hour or a day).

In this book, we will study several ML methods in detail which have been designed to handle these different varieties of process systems. Let us first understand what we mean by machine learning.

1.2 What is Machine Learning

At its core, machine learning simply means using computer programs and data for finding relationship between different components of a system and discovering patterns that were not immediately evident. This characteristic of using data to learn about the system makes machine learning interesting and different. System specific knowledge is not explicitly embedded into a ML program; the program extracts the knowledge from data. Figure 1.3 compares ML approach with a non-ML approach for distillation column modeling: ML approach does not require system specific data such as number of stages or the type of packing.

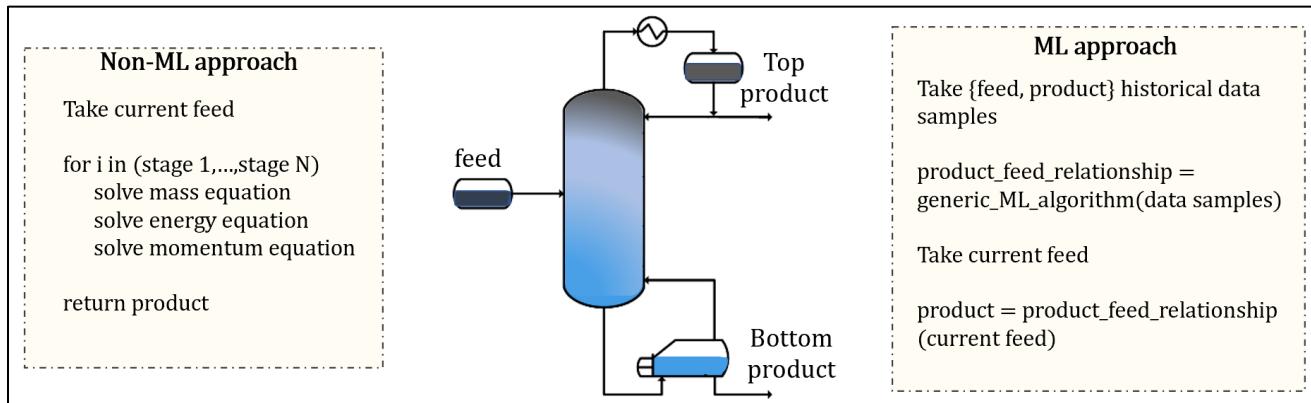


Figure 1.3: Computer program using first-principal approach (left) and ML approach (right) for modeling a distillation column

We often marvel at the accuracy of recommendations made by Netflix or amazon for potential shows or products. These companies do not possess explicit information about our personal preferences or psychology (whether we like sci-fi movies or not). The data does all the trick! ML algorithms process past purchase data to discern the likes and dislikes of its customers and make recommendations. In process industry, manufacturers use ML methods to determine optimal equipment maintenance schedule using past maintenance and operating conditions data. Here again, data-based analysis provides considerable convenience over the alternative method of complex metallurgical analysis.

The reliance on data alone to obtain reasonably good system approximation is one of the major reasons behind ML's growing popularity. The barrier of requiring specific domain knowledge to be able to analyze a system has been circumvented by machine learning. ML algorithms are also universal in nature. For example, we can use the same data-clustering algorithm for analyzing demographics data, factory data, or economic data to obtain actionable knowledge. These properties combined with the surge in the amount of data

collected and the dip in the cost of computational resources had led machine learning to revolutionize several industries.



The increasing popularity of machine learning does not mean domain knowledge is redundant. Domain expertise often helps to decide which ML algorithm to use and proper infusion of domain knowledge into ML methodology can sometimes increase model accuracy considerably.

"Why don't we just build a model using first-principles and get very accurate models? Why rely on data?" This is a valid question. There is no doubt that fundamental models have better accuracy and generalization capability, however, developing fundamental models are often time consuming and require expert resources. These models can sometimes be too complex to execute in real-time. Adopting ML methodology can help getting around these difficulties.

Machine learning workflow

Figure 1.4 shows the typical tasks involved in a machine learning project. The tasks are categorized into offline computations and online/real-time computations. In online computations, process data are parsed through the process model to provide real-time insights and results. Note that the models could be process input-output models, process monitoring models (process normal/abnormal classification), mode categorization models, or fault classification models.

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:

- *Sample and variable selection:* One does not simply dump all the available historical data and sensor measurements into model training module. Only the portion of historical data that best represents the current process behavior or the behavior of interest is utilized. For process systems, it is common to use data over the past couple of years as training data. If steady-state models are being build, then data from steady-state operation periods are used.

Input variable selection warrants judicious consideration as well. While including too many model inputs leads to overfitting and high computational complexity, leaving out important variables leads to underfitting. The basic principle is to include only those inputs that are known to influence the model outputs. Specific algorithms for variable selection are covered in Chapter 4.

- *Data cleaning*: “Garbage in, garbage out” is an age-old principle in computer simulations. The same holds for ML model training. Your model will be practically useless if training data is not ‘clean’. For example, process monitoring model won’t be able to detect process abnormalities accurately if it has been trained with outlier-infested training data. Data cleaning includes, amongst others, identification and removal of outliers, and removal of noise effects. Detailed algorithms are covered in Chapter 4.
- *Model training and validation*: Model training imply estimating the parameters of the chosen ML model, for example, the coefficients in a linear regression model. Model validation is employed for finding optimal values of model hyperparameters, for example, regularization coefficient in ridge regression. At the end of this step, the coveted process model is obtained. Several best practices for model training and validation are covered in Chapter 3.

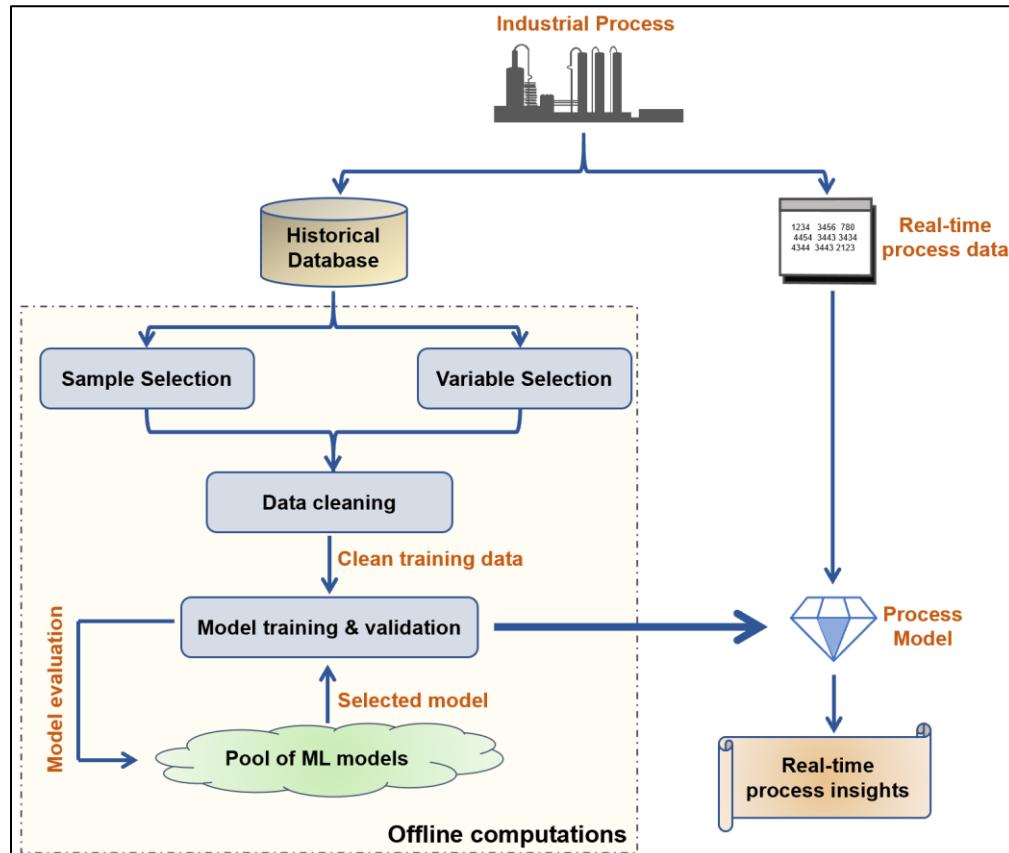


Figure 1.4: Steps involved in a typical ML-based methodology

Distinct from the offline-online paradigm, there is another approach employed in process industry, especially for nonlinear and multimode processes. It is called just-in-time learning or lazy learning. As shown in Figure 1.5, the model building exercise is carried out online as well. 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.

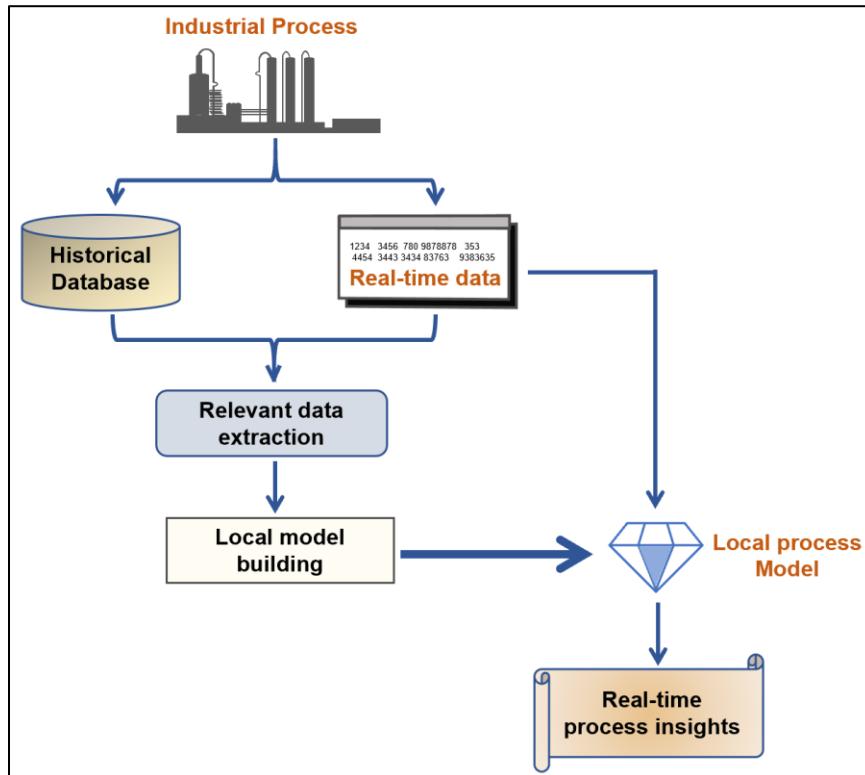


Figure 1.5: Steps involved in a just-in-time learning methodology

Types of machine learning systems

Although there is a plethora of ML models available, all of them can be organized into three broad categories, namely, supervised learning, unsupervised learning, and reinforcement learning. The categorization is based on the nature of model output associated with each sample in the training dataset. Note that there is another category, semi-supervised learning – this method, however, is used less often compared to the other mentioned methods. Figure 1.6 gives a sneak peek into some of the ML algorithms and application areas corresponding to supervised and unsupervised learnings that we will cover in this book.

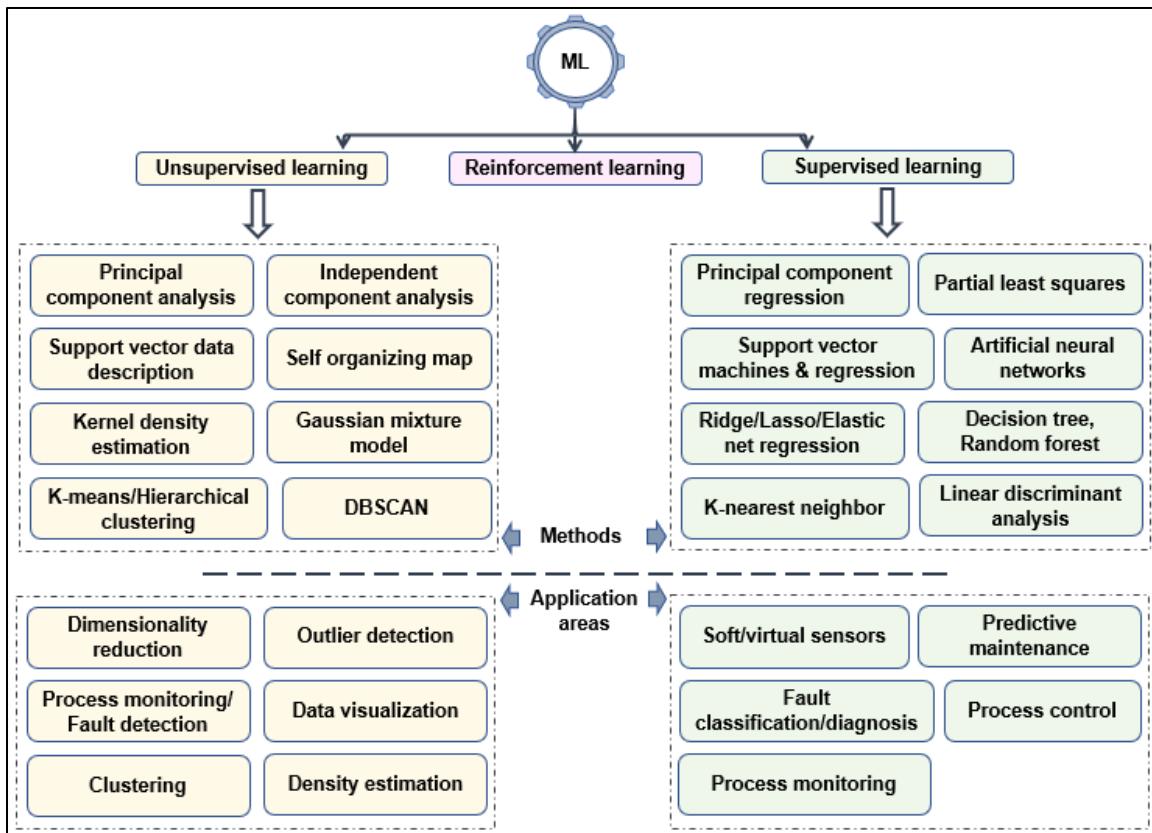


Figure 1.6: Classification of machine learning methods

Let's now try to understand these categories a bit more clearly.

Supervised learning

Supervised learning is used when training data includes sets of inputs and associated outputs. As illustrated in Figure 1.7, supervised learning models learn the input-output relationship and use this relationship to predict the unknown output for a new input. The outputs can be discrete values (for classification problems) or continuous values (for regression problems).

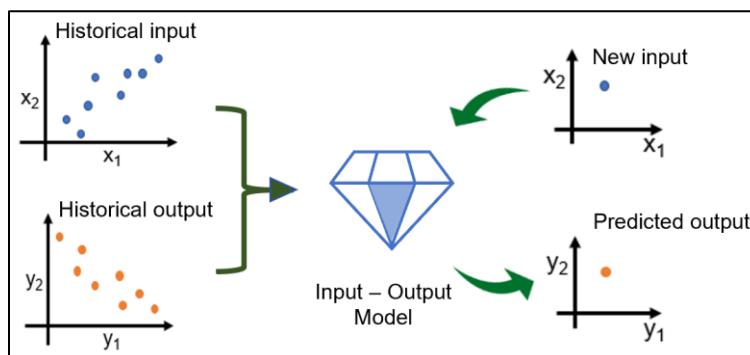


Figure 1.7: Supervised learning scheme

Unsupervised learning

Unsupervised learning is used when training data is not divided into inputs and outputs. The primary purpose is to find hidden pattern, if any, in data. An example situation is illustrated in Figure 1.8, where an unsupervised learning model finds prevailing structure (distinct clusters) in historical data and buckets them into different groups. The model can now be used to assign any new incoming sample into either of the groups. Unsupervised learning is often used in conjunction with supervised learning. For example, in Figure 1.8, separate local models can be built via supervised learning for data in different clusters. As you would have guessed correctly, data would need to be separated into inputs and outputs before application of supervised learning.

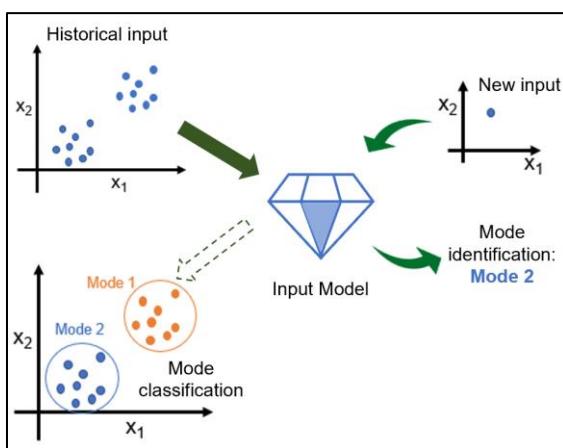


Figure 1.8: Unsupervised learning scheme

Reinforcement learning (RL)

Unlike supervised and unsupervised learning where there is one-time interaction between the system (environment) and model (ML agent) during training, in reinforcement learning the agent continuously interacts with the environment to generate training data to ‘learn’ an optimal strategy for accomplishing a task. The actions decided as per the learnt strategy are such that the long-term rewards are maximized. For example, consider a simple task of controlling water level in a container at a fixed height during rainy days.

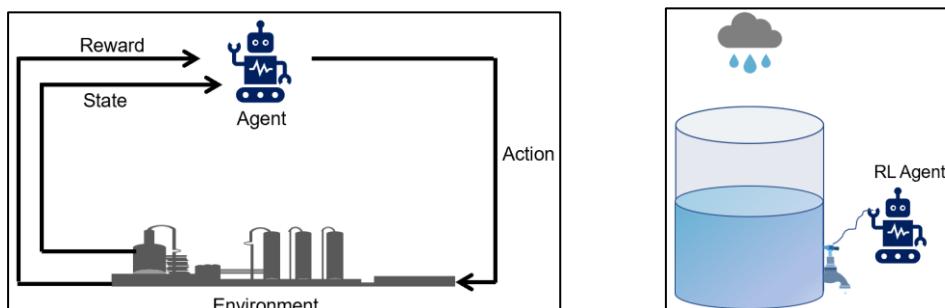


Figure 1.9: Reinforcement learning scheme and simple application setup

The RL agent takes actions to adjust the tap opening according to the current system state to maintain the level at some setpoint. The trivial policy of opening and closing the tap completely upon any level change can lead to high water level fluctuations. Therefore, during training the agent learns the best control policy automatically by just interacting with its environment.

1.3 Machine Learning Applications in Process Industry

Product quality control, safe work environment, optimal operations, and sustainable operations are the primary objectives in any industrial plant. Today, ML-based solutions are being utilized for all these purposes. Surrogate models are developed for online prediction of key quality variables and optimizing plant productivity, process monitoring and fault diagnosis models are developed for real-time tracking of process operating conditions, data mining is used for alarm management, data clustering is used for operation mode identification, and the list goes on.

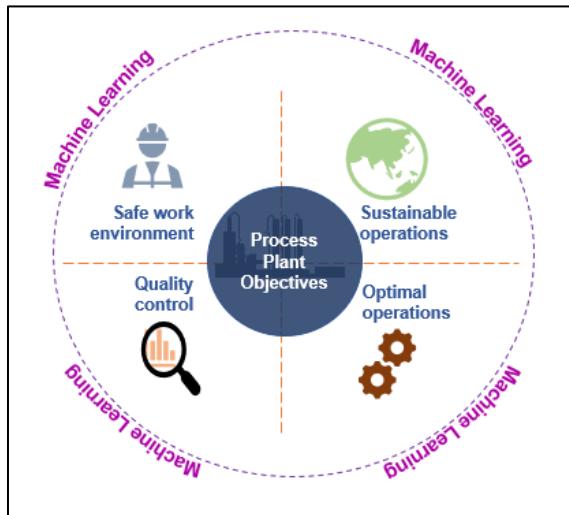


Figure 1.10: Use of machine learning to solve process plant objectives

Several success stories on machine learning application in process industry are publicly available. Shell² used recurrent neural networks (RNNs) for early prediction of valve failures, Saudi Aramco used ML tools for alarm analytics and predictive maintenance of turbines³, a polymer manufacturing company used ML-based feature extractions⁴ for troubleshooting quality control issues. In recent times, there has been a proliferation in the number of

² <https://www.aiche.org/conferences/aiche-spring-meeting-and-global-congress-on-process-safety/2018/proceeding/paper/37a-digital-twins-predicting-early-onset-failures-flow-valves>

³ <https://pubs.acs.org/doi/abs/10.1021/acs.iecr.8b06205>

⁴ <https://www.yokogawa.com/at/library/resources/references/successstory-sumitomoseika-chemicals-en/>

commercial data-analytics software (Aspen Mett, IBM SPSS) or services offerings for process industry. This is just a testament to the growing trend of ML-driven process control and operations.

As process data scientists, we should be proud of the fact that process industry has always been a pioneer in utilizing process data for plant operations. Model predictive control (MPC) is a classic example which uses data-based model for process control. It has been used as a standard supervisory controller long before Big Data and ML became the buzzwords. Partial least squares (PLS), a popular dimensionality reduction-based soft sensing method, has long been used for online product quality predictions. The new craze about machine learning has only imparted a renewed push to explore non-traditional applications of ML in process industry.

Decision hierarchy levels in a process plant

Before we look at some specific examples of ML applications for process systems, let us first familiarize ourselves with the typical decision-making hierarchy in a process plant. Once you understand this, you will be able to easily identify the avenues where ML renders itself useful in the world of process operations and appreciate how data science has percolated all levels of process decision hierarchy.

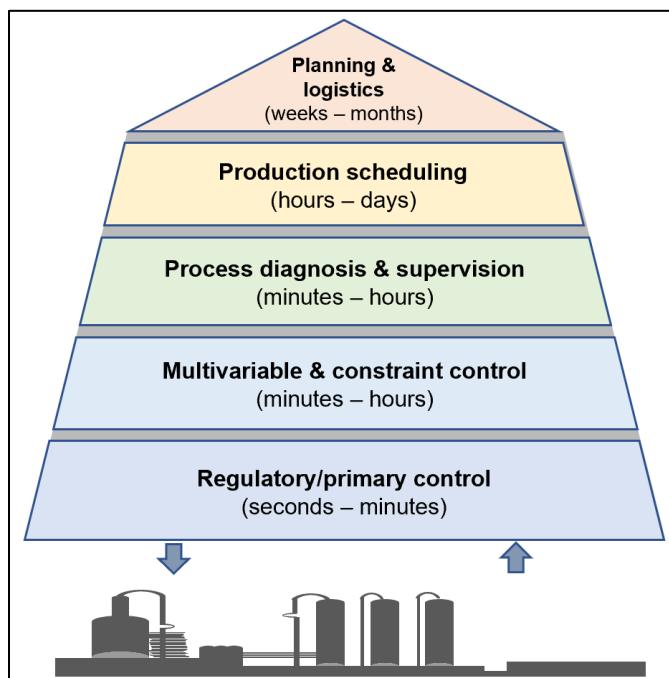


Figure 1.11: Industrial process control/decision-making hierarchy. Interval ranges in brackets show the timescales at which corresponding decisions are made.

At the base level, in Figure 1.11, resides the basic regulatory control layer which primarily comprises of control valves; these valves are used to modulate the flow of process streams and indirectly the pressures and temperatures as per process requirements. These requirements are in turn determined by the multivariable control layer which usually consists of MPC and RTO (real-time optimization) modules. This layer determines the base layer requirements using multivariable relationships between plant variables to ensure optimal performance of the plant. This layer is also responsible for ensuring that plant operations remain safe by ensuring safety-critical variables remain within stipulated bounds. The process diagnostics layer, if present, ensures reliability of the process through timely fault detection and diagnosis. This layer may also perform the task of controller performance assessment.

The production scheduling layer has models and methods to determine resource allocation and short-term production schedules considering external influences such as electricity prices or raw material price variations. Time-based or predictive equipment maintenance decisions may also be made in this layer. Results from this layer are communicated to the multivariable control layer. The top-most layer, planning and logistics, make enterprise-wide decisions. An enterprise operating multiple facilities use this layer to determine site-wise production targets based on the enterprise's strategic goals.

Application areas

After familiarization with the process operation decision hierarchy, we are now well-equipped to see the broad application areas of ML in a process plant. We have already seen some of the application areas in Figure 1.6. In this section we will use a furnace (Figure 1.12) as an example system to investigate these in more details.

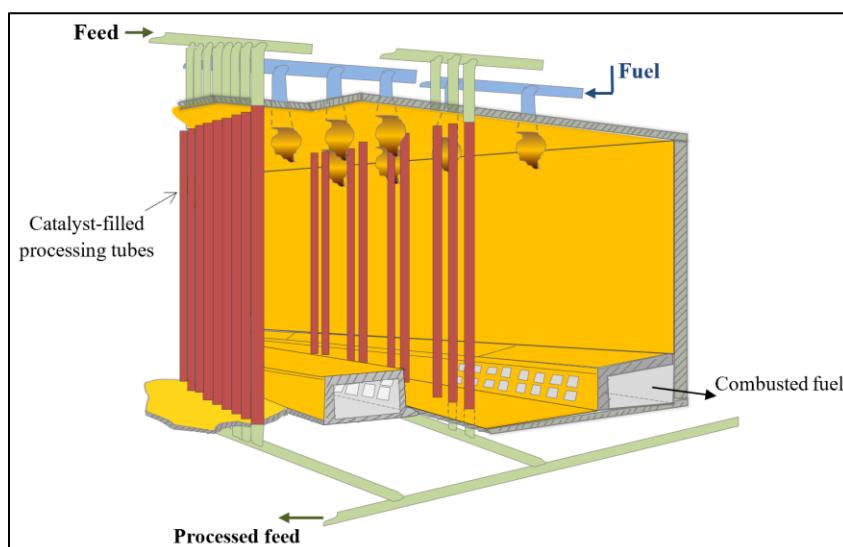


Figure 1.12: Furnace system with catalyst-filled tubes

The furnace system consists of several catalyst-filled tubes suspended vertically in a natural-gas fired furnace. Unreacted gas stream enters the tubes from the top, undergoes chemical reactions as they flow down the tubes and exit at the bottom. The heat from fuel combustion provides energy for the chemical reactions.

Soft Sensing

Soft sensors, also called virtual sensors or inferential sensors, are mathematical models used to estimate the values of unknown process variables using available measurements. Soft sensor models can be first principles-based or data-based and are employed when physical sensors are very costly or real-time measurements are not possible (in case of composition measurements).

Usage of data-based soft sensors are popular where process systems are too complex to build mechanistic models, or the mechanistic models are too complex to provide required estimates in real-time. For example, for the furnace system, one can build a computational fluid dynamic (CFD) model to predict the species mole fractions in the processed gas stream as a function of process inputs – fuel, air, unprocessed gas. This estimate can be used by multivariable control layer to adjust input flows accordingly, for example, increase fuel if conversion is low. However, CFD models have large execution times. As an alternative, data-based models can utilize past process data to estimate an appropriate relationship and provide compositions in real-time.



As a process modeler, you should pay careful attention that the training data is sufficiently rich in information, otherwise, a poor/low-accuracy model will result. One way to ascertain data-richness is to check if process inputs show adequate variability.

Partial least squares (PLS), principal component regression (PCR), support vector regression (SVR) are some of the popular ML method choices for estimating process quality variables. In recent times, artificial neural networks (ANNs) have also seen increased usage.

Process Monitoring

Process monitoring/fault detection/abnormality detection is among the most popular application of ML in process industry. The ML model flags an alarm when current process data shows inconsistency with historical behavior as estimated from historical data. These discrepancies could be indications of severe process faults. In the furnace system, there are

White-box/grey-box/black-box models

For soft sensing, terms like white-box, grey-box, black-box are often used. Figure 1.13 clarifies the difference between the terms. White-box models utilize fundamental mass/energy/momentum conservation laws to relate model inputs and outputs. Black-box methods build estimation models using only process data. Grey-box methods combine the two approaches to generate a hybrid model.

Consider our furnace system again. CFD model would be the white-box model. PLS model relating furnace inputs to product stream composition would fall in black-box model category. However, to balance the trade-off between model accuracy and computational expense, a hybrid model can be built. Mechanistic model of radiation energy transfer is the most complex part of the furnace model. One can build a black-box model to estimate the amount of radiation energy supplied to the tubes. This model can then be combined with the mechanistic model of tube to predict product composition.

Hybrid models tend to have better extrapolation accuracies compared to black-box models and are preferred when the amount of training data is low.

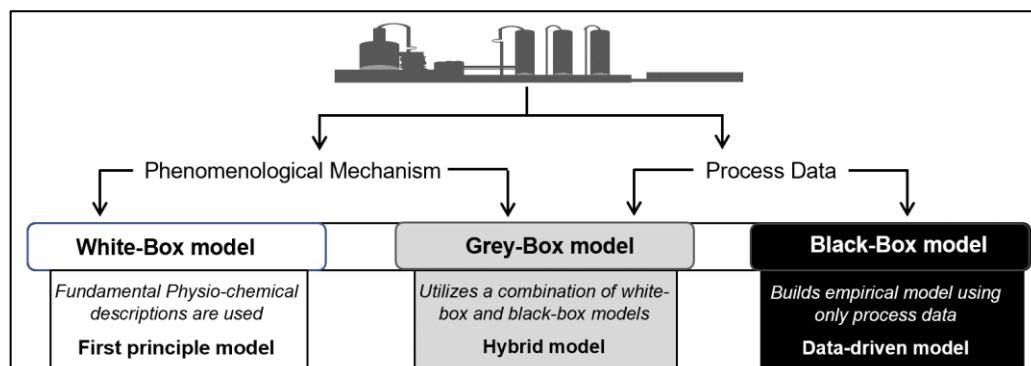


Figure 1.13: Soft sensing methodology spectrum

hundreds of tube temperature measurements. Unfavorable conditions like catalyst damage or tube leaks may induce abnormal increase in some of the temperatures. However, it is impractical to monitor all these temperatures manually all the time. Instead, process monitoring models could be built using historical data that provides early warning for temperature upsets so that timely corrective actions can be taken.

Principal component analysis (PCA) is the most popular method employed for process monitoring. PLS, independent component analysis (ICA), support vector data description (SVDD), self-organizing maps are some other commonly used methods. We will study all these methods and their applications for process monitoring in the later chapters.

Fault Classification

Once a process fault has been detected, the next challenge lies in timely identification of root-cause of the issue or identification of the process variables that are responsible for the process upset. This exercise is called fault diagnosis. If historical data on past failures are available, a classification model could be built to determine the specific fault.

For furnace system, refractory damage, tube leaks, burner malfunctions, catalyst damage can all cause temperature upsets. These different faults tend to impact tube temperatures differently; these differences are exploited by a fault classification model to identify the current fault. In this book, we will study how methods like ANNs, support vector machine (SVM) and linear discriminant analysis (LDA) can help in fault classifications.

Process Optimization & Control

Optimizing a complex system can be computationally expensive. It is not uncommon to find practitioners developing ML-based surrogate models from data to optimize the system offline or in RTO⁵ layer. Surrogate models are also used in MPCs for highly non-linear systems. Reinforcement learning models are being tried in the regulatory control layer or for adaptively tuning regulatory controllers⁶.

Data Clustering & Mining

Data clustering and mining methods are frequently used for activities like alarm management, operating mode characterization, pattern recognition, etc. For the furnace system, clustering models can be used to find which set of tubes tend to show similar temperatures. Data mining models could be used to find the effect of process conditions on tube lifespan.

Predictive Maintenance

Predictive maintenance is another very popular usage of ML in process industry. Predictive maintenance models are built to determine the time to failure of any equipment or detect

⁵ Real-Time Optimization and Control of Nonlinear Processes Using Machine Learning, Zhang et. al., Mathematics, 2019

⁶ Adaptive PID controller tuning via deep reinforcement learning, US patent 187631, 2019

patterns in process data that could signal an impending process failure. Detailed planned maintenance can be carried out if failure times are known in advance. In the furnace system, tube leak occurrences may be preceded by a specific pattern in temperatures of neighboring tubes. These patterns are identified during model training and then utilized for real-time failure predictions. Advance warning can help plant operators plan plant shutdown properly.

Forecasting

Uncertainty in product demands and prices of raw materials lead to poor production scheduling. Advanced ML forecasting models are built to determine optimal production plan to maximize resource utilization and minimize production costs. In furnace system, frequent furnace temperature swings have detrimental effect on tube lifespan. If accurate monthly product demand is known in advance, then the furnace can be operated at a steady state throughout the month while using product storage to handle momentary spikes in demand.

Choosing the right ML algorithm



One of the trickiest tasks in a machine learning project is selection of modeling algorithm. Even experienced ML practitioners often recommend trial-and-error approaches. However, with experience, understanding of underlying details of ML algorithms, and process knowledge you can narrow down the trial candidates.

Let's take a sneak peek into the model selection process for the task of process monitoring. For monitoring, if the system is linear, PCA or PLS models should be tried first. If variables are non-Gaussian distributed, kernel density estimation (KDE) or SVDD can be used for control-limit determination. For nonlinear systems, kernelized PCA/PLS, ANN, SVMs may be explored. SVMs are preferred when training data are limited. If process exhibits multiple operation mode, then mixture modeling like Gaussian mixture model (GMM) can provide better monitoring performance.

1.4 ML Solution Deployment

While working on any project, you are most likely to experiment with several ML algorithms on your personal laptop/computer. After having done all the hard work and having decided the final form of your ML workflow, you might find yourself asking the following questions:

- Where does my ML tool reside where it can run uninterrupted?
- How does an end-user access the results of my tool?

The answer to the above questions is summarized in Figure 1.14 which shows a common architecture followed for deploying a Python machine learning solution from scratch within an enterprise network. You will install Python and transfer your tool (Python files) in a tool server machine (a virtual or physical machine where the ML tool would run). The tool server will be configured to execute the tool continuously (as a windows service) or on a schedule. During execution, your ML tool will fetch historical or real-time plant data and store the ML results in a database (MS SQL, MYSQL, etc.).

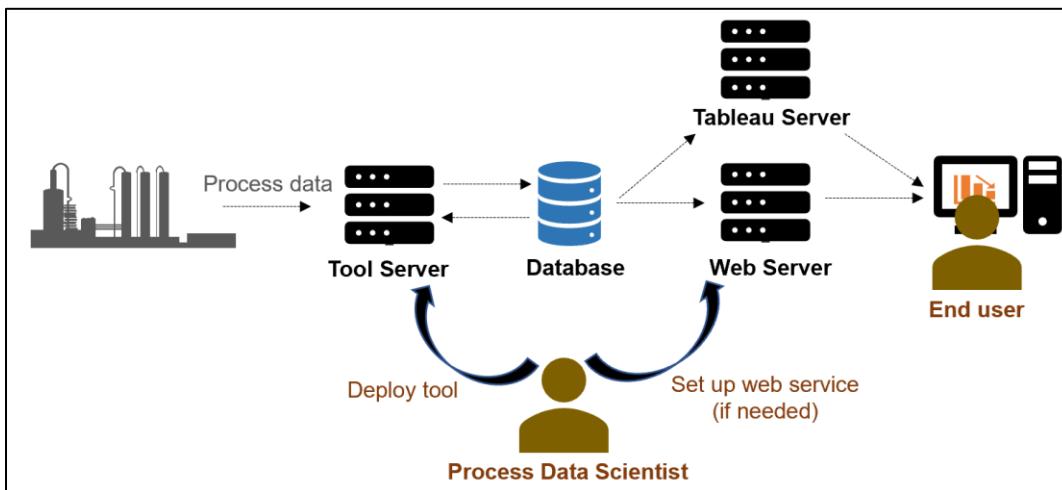


Figure 1.14: ML solution deployment

Let's tackle the second question now. The end-users such as plant operators can access tool's results via a web browser. The user interface could be either built using third-party visualization software (Tableau, Sisense, Power BI) or completely custom-built using front-end web frameworks like bootstrap. If building custom website, then you will also need to setup a web server (using Python, .Net, etc.) which will serve the user-interface webpage when requested through web browser. The user interface communicates with the database to display appropriate data to the end-user. The web server may be configured to execute your tool on demand as well. The web server may be hosted on a separate machine or on the tool server machine itself.

That is all it takes to deploy a ML solution in a production environment. If all this IT stuff has overwhelmed you, don't worry! It is simpler than it seems and in Chapter 14 we will build and deploy an end-to-end solution following this architecture.

1.5 The Future of Process Data Science

It is not an exaggeration to say that this is a wonderful time to be a process data scientist. Process industry is witnessing higher and higher product demands due to increasing population and growing lifestyle globally, but there is also a push to run production facilities more efficiently and sustainably. Consequently, adoption of Industry 4.0, which mandates utilizing process data for process improvements all along the production chain, is on the rise. There is palpable interest among process industry executives to implement ML-based solutions and the responsibility to show that the ML hype is true has fallen on the shoulders of process data scientists.

It's a foregone conclusion that ML is a powerful tool for PSE. Process data hold tremendous power if they are put to use in the right way. However, blind application of ML often leads to discouraging results. As a process data scientist with expert process knowledge and ML skills, you are in a unique position to combine process systems knowledge and power of ML to unleash the true potentials of data science in process industry. Let's cheer to your bright career prospects as a process data scientist and continue our journey to now learn the intricate details of ML algorithms.

Summary

In this chapter we tried to get a conceptual understanding of where ML fits in the world of process industry. We looked at the different types of machine learning workflows and methodologies. We also explored some application areas in process industry where ML has proved useful. We hope that you got the chapter's overarching message that process data science has already proven to be an indispensable tool in process operations to turn data into knowledge and support effective decision making. In the next chapter we will take the first step and learn about the environment you will use to execute your Python scripts containing ML code.

Chapter 2

The Scripting Environment

In the previous chapter, we studied the various aspects of machine learning and learned about its different uses in process industry. 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
- Overview of a typical ML script

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 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 easy.



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 popular and 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.

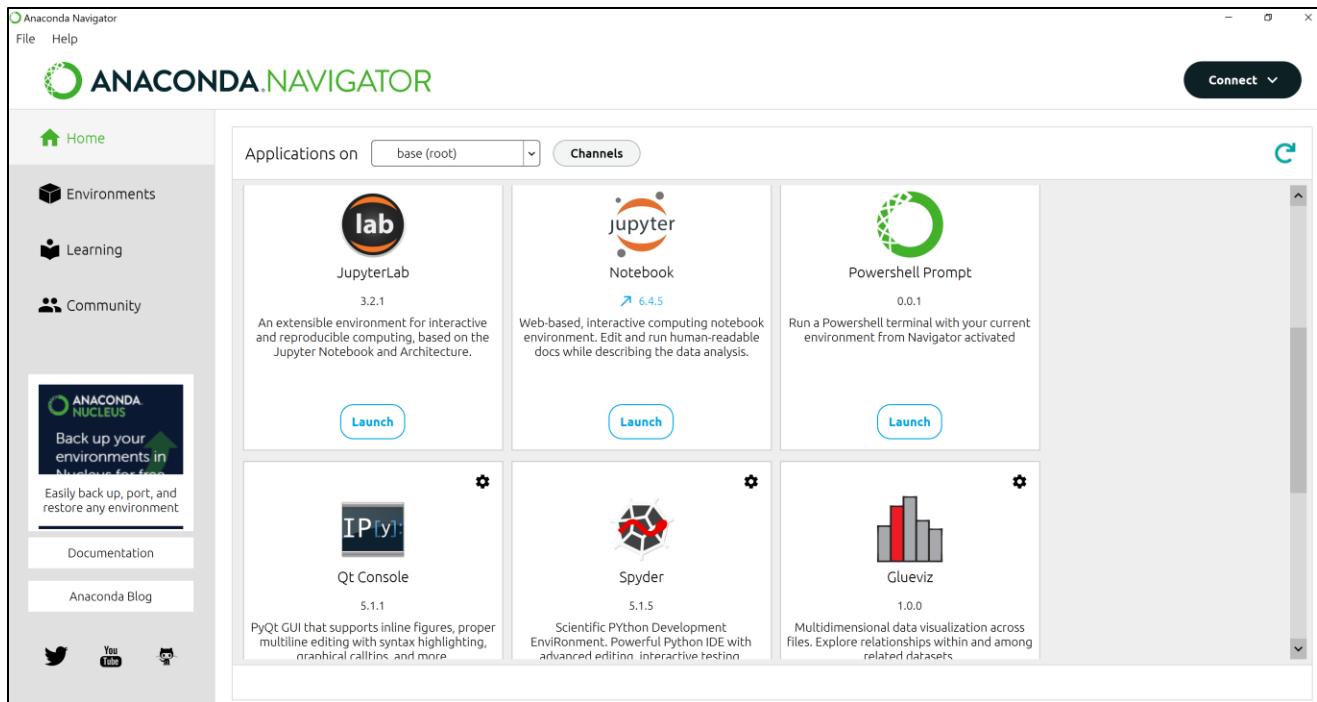


Figure 2.2: Anaconda Navigator GUI

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, and 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.

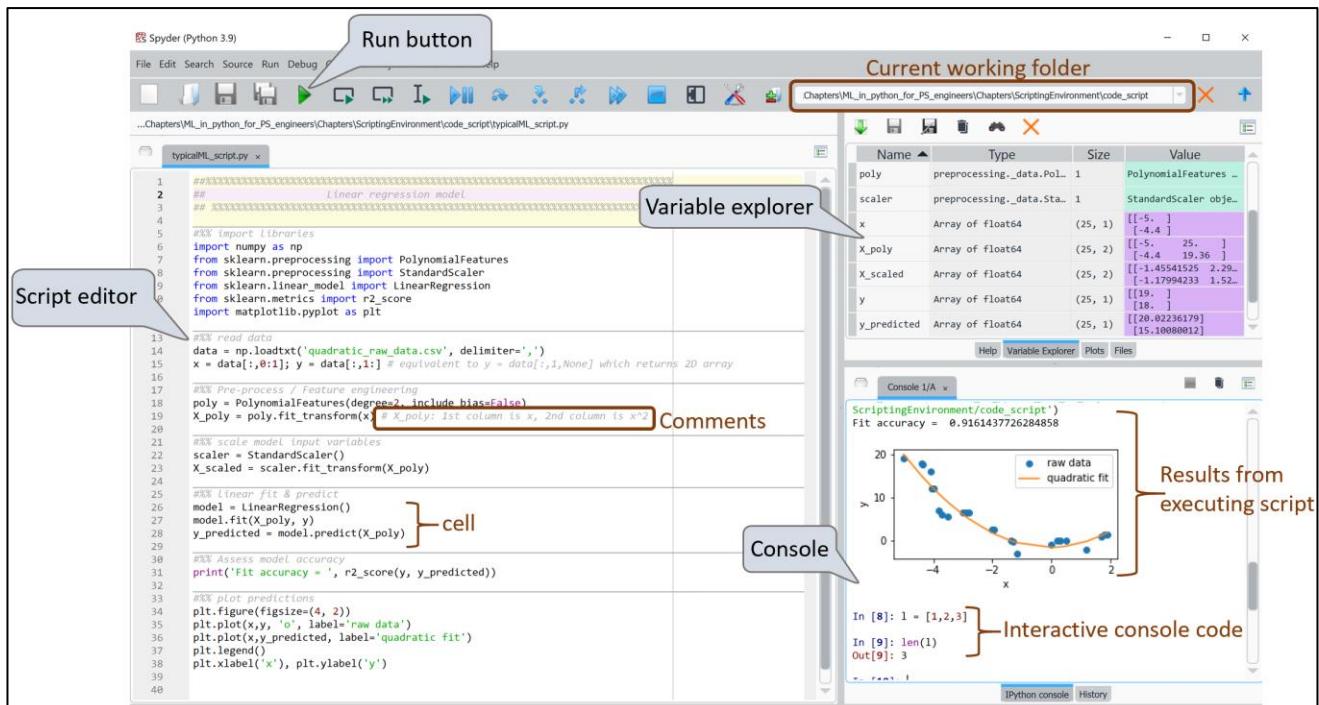


Figure 2.3: Spyder interface

Like any IDE, Spyder offers several features. You can divide your script into cells and execute only 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 up 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.

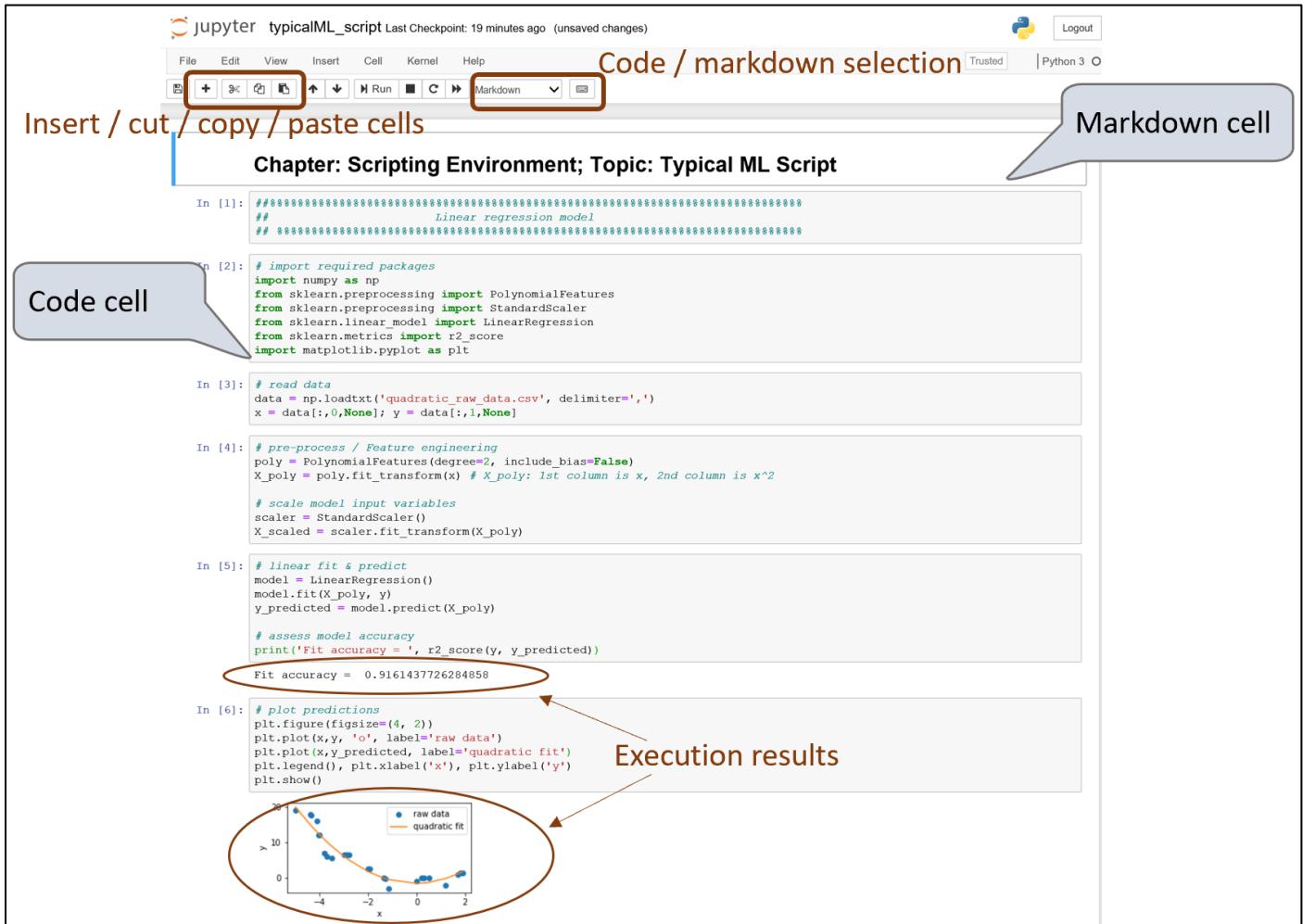


Figure 2.4: Jupyter interface

You can designate any input cell as a code or markdown (formatted explanatory text). You can press Ctrl + 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 interfaces. 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]
```

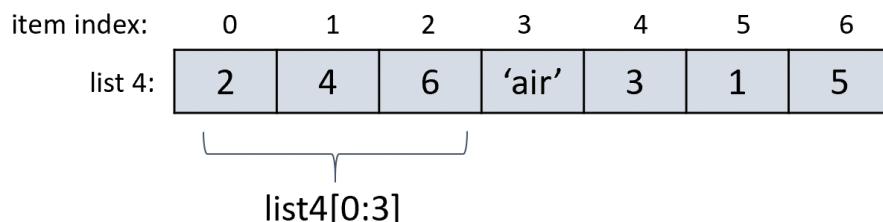
Indexing and slicing sequences

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
print(sumSquares(list4)) # displays 55

```



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

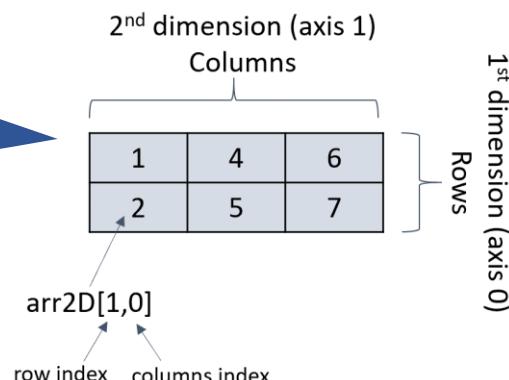
While the core Python data-structures are quite handy, they are not very convenient for the advanced data manipulations we require for machine learning tasks. Fortunately, specialized packages like NumPy, SciPy, Pandas exist which provide convenient multidimensional tabular data structures suited for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

NumPy

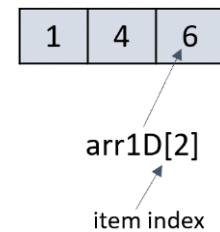
In NumPy, ndarrays are the basic data structures which put data in a grid of values. Illustrations below show how 1D and 2D arrays can be created and their items accessed

```
# 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.]]

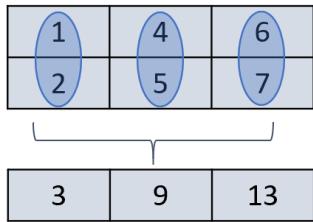
print(np.vstack((arr5,arr6))) # rows of array 6 added onto arr5
>>> [[0. 0.]
      [0. 0.]
      [1. 0.]
      [0. 1.]]

```

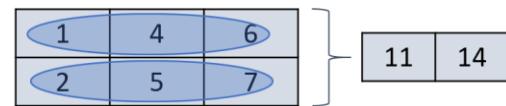
Basic Numpy functions

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

```

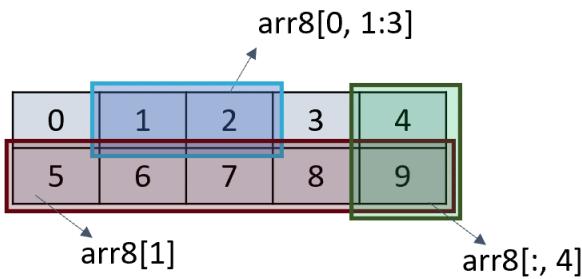
# accessing individual items
print(arr2D[1,2]) # returns 7

# 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.

arr2D	+	arr1D	=	arr_sum																		
<table border="1"> <tr><td>1</td><td>4</td><td>6</td></tr> <tr><td>2</td><td>5</td><td>7</td></tr> </table>	1	4	6	2	5	7		<table border="1"> <tr><td>1</td><td>4</td><td>6</td></tr> <tr><td>1</td><td>4</td><td>6</td></tr> </table>	1	4	6	1	4	6		<table border="1"> <tr><td>2</td><td>8</td><td>12</td></tr> <tr><td>3</td><td>9</td><td>13</td></tr> </table>	2	8	12	3	9	13
1	4	6																				
2	5	7																				
1	4	6																				
1	4	6																				
2	8	12																				
3	9	13																				

Implicit extension of arr1D
to match arr2D shape

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

```

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':s}) # 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 examples that we will frequently employ in this book.

```
# 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.

2.5 Typical ML Script

By now you understand where to write your ML code and how to check the ML output. Let's now take a slightly closer look at the ML script that we executed in the earlier section. This

script will give a fairly good idea about what a typical ML script looks like. You will also understand how Python, NumPy, and advanced ML packages are utilized for ML scripting. We will study several ML aspects in much greater detail in the next few chapters, but this simple script is a good start. The objective of this simple script is to take data for an input and an output variable from a file and build a regression model between them. The first few code lines take care of importing the libraries that the script will employ.

```
# import libraries
import numpy as np
from sklearn.preprocessing import PolynomialFeatures ← Used for data transformations
from sklearn.preprocessing import StandardScaler ←
from sklearn.linear_model import LinearRegression ← Used for building linear models
from sklearn.metrics import r2_score ←
import matplotlib.pyplot as plt ← Used for model assessment and plotting
```

We studied NumPy before; another library⁹ that we see here is Sklearn which is currently the most popular library for machine learning in Python. Sklearn provides several modules (like preprocessing, linear_model, metrics that are used here) to handle different aspects of ML. The last library is matplotlib¹⁰ which is used for creating visualization plots. The next few lines of code fetch raw data.

```
# read data
data = np.loadtxt('quadratic_raw_data.csv', delimiter=',')
x = data[:,0:1]; y = data[:,1:] # equivalent to y = data[:,1,None] which returns 2D array
```



Here NumPy's `loadtxt` function is used to read comma-separated data in the file `quadratic_raw_data.csv`. The data get stored in a 2D NumPy array, `data`, where the first and 2nd columns contain data for the input and output variables, respectively. NumPy slicing is

⁹ If any package/library that you need is not installed on your machine, you can get it by running the command `pip install <package-name>` on Spyder console

¹⁰ Seaborn is another popular library for creating nice-looking plots

used to separate the x and y data. The next part of the script performs input data pre-processing to transform it into proper form before generating a regression model.

```
# Pre-process / Feature engineering
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(x)
# X_poly: 1st column is x; 2nd column is  $x^2$ 
```

	x	X_{poly}
0	-5	0
1	-4.4	25
2	-4.35	-4.4
3	-4.1	19.36
4	-4.05	18.9225
5	-4	-4.1

```
# scale model input variables
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_poly)
```

	X_{poly}	X_{scaled}
0	0	-1.45542
1	25	2.29253
2	-4.4	-1.17994
3	19.36	1.52777
4	18.9225	-1.15699
5	-4.1	1.46845
		-1.04221
		1.182
		-1.01925
		1.12675
		-0.996294
		1.07217

Here, two pre-processing transformations are done on the input data. First, quadratic powers of x are generated and then both x and x^2 values are scaled. You will learn more about these transformations in the next chapter. Next, a regression model is fitted and used to make predictions

```
# fit linear model & predict
model = LinearRegression()
model.fit(X_poly, y)
y_predicted = model.predict(X_poly)
```

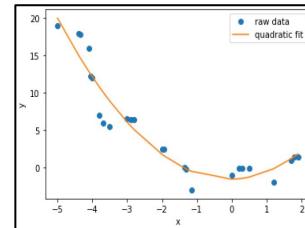
$$y = 0.88x^2 + 0.01x - 1.56$$

fitted model

In the last step, model performance assessment is done.

```
# Assess model accuracy
print('Fit accuracy = ', r2_score(y, y_predicted))
>>> Fit accuracy = 0.916
```

```
# plot predictions
plt.figure(figsize=(4, 2))
plt.plot(x, y, 'o', label='raw data')
plt.plot(x, y_predicted, label='quadratic fit')
plt.legend()
plt.xlabel('x'), plt.ylabel('y')
```



Although there are many more advanced ML aspects, the above code is very representative of what goes in a script. Hopefully, you also got a feel of the powerful capabilities of the

Sklearn library that helps to encapsulate complex ML algorithms into user-friendly fit/transform/predict constructs.

Virtual environments

Imagine that you wrote a Python ML code two years ago that makes use of Sklearn packages which you now want to execute. However, the version of Sklearn packages that you have on your computer currently installed is most likely different from what you had 2 years ago. Therefore, your old script may show errors upon execution (due to changed syntaxes for example) and it can be frustrating to have to debug the old code due to package version mismatches. This is one of the many reasons why you may want to have separate Python environments for separate projects such that these projects have access to separate sets of libraries. Note that this does not mean that you need to install Python itself multiple times. The package `venv` can help create virtual Python environments where different versions of the same packages can reside ‘peacefully’. In our hypothetical scenario, if you had used a virtual environment two years ago, you could simply activate that virtual environment and run your old script without any package-version-related issues.

Using virtual environments is one of the best practices we recommend you should adopt as you become a more experienced programmer. For more details you can check out the official documentation docs.python.org/3/tutorials/venv.html.

Summary

In this chapter we made ourselves familiar with the scripting environment that we will use for writing and executing ML 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 be exposed to several crucial aspects of ML that you should keep in mind as you plan your ML strategy for any given problem.

Chapter 3

Machine Learning Model Development: Workflow and Best Practices

A model is the most crucial part of a machine learning-based solution as it encapsulates the relationships among process variables extracted from data. A universally accepted completely automated ML tool that can generate good models for all kinds of problems is not yet available. ML model development is still an art and it takes an artist to create a good model! While it is true that we don't necessarily need to know the algorithmic implementation details of the models, there are several model development stages which require our active participation. You cannot obtain a good model by just dumping raw data into a model training module. You need to ensure that you are providing right data for model training, using correct performance assessment metric, properly assessing if model is overfitting or underfitting, and fine-tuning the model properly if needed

There are several established best practices that have been devised to help simplify a modeler's task. You can use these as guidance as you go about developing your ML model. In this chapter we will study these best practices and learn what it takes to obtain a good model (as well as how to quantify the 'goodness' of a model!). This knowledge will help us when we start working with different models in the next parts of the book. Specifically, the following topics are covered

- Transforming raw process data to obtain better model training dataset
 - Overview of performance metrics and best practices for reporting model performance
 - Diagnosis of overfitting and underfitting using validation curves
 - Tackling overfitting via regularization
 - Best practices for model fine-tuning and hyperparameter optimization
-

3.1 ML Model Development Workflow

The prime objective of the ML modeling task is to obtain a model that generalizes well, which means that the model performs satisfactorily when supplied with data different from that used during model training. Achieving this takes more than just executing a ‘model = <some ML_model>.fit()’ command on the available data. In Chapter 1, we had a brief introduction to ML modeling workflow. In this chapter, we will learn about the different components of that workflow in more details. Figure 3.1 lists the common tasks you can expect yourself to carry out while building a ML model. While separate books can be written on each of these tasks, we will cover enough details on each of these to get you started on your ML journey.

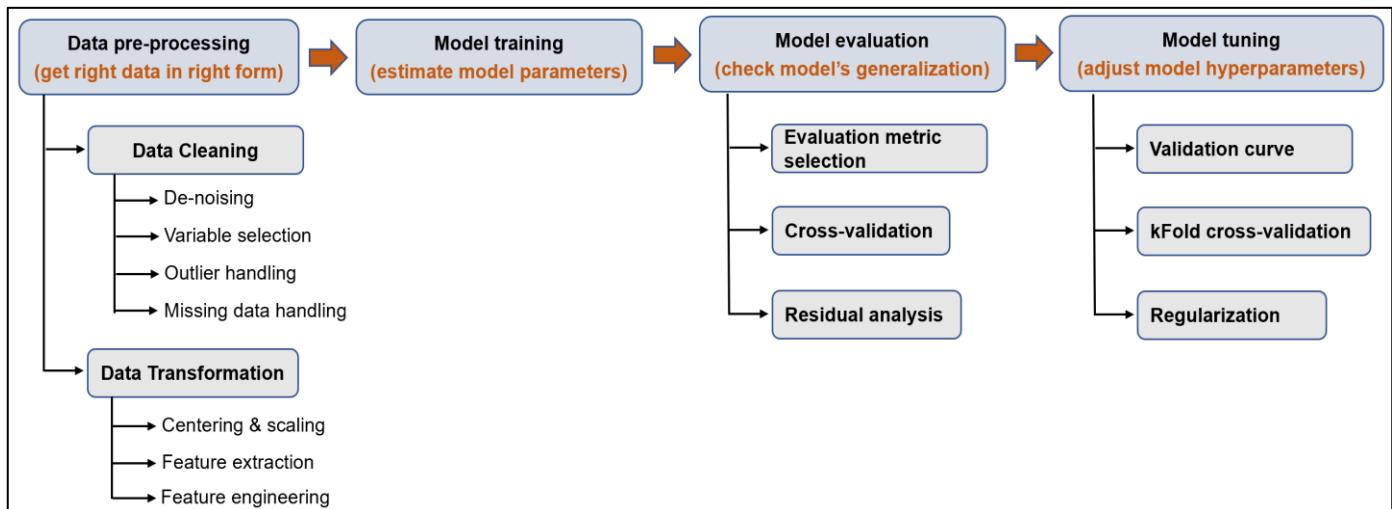


Figure 3.1: Recommended steps in a typical ML model development workflow

As you can see, there are four main steps in the workflow. In the first step, the available raw data are pre-processed to ensure ‘correct’ dataset is supplied to the subsequent model fitting step. This step can involve data cleaning (discarding irrelevant variables or corrupted/incomplete data samples) and data transformation (generating new features or modifying existing ones). While we will study data-transformation techniques in this chapter, data cleaning techniques are treated extensively in the next chapter. The next step of the workflow involves selection of a suitable model and estimating its parameters. While parameter estimation is handled by off-the-shelf model libraries available in Sklearn, model selection would require your judgement based on your expert knowledge of the specific process system. A practical guideline for model selection is Occam’s razor which advises selection of the simplest model that generalizes well. After training, the model is evaluated to validate the modeling assumptions and assess model’s generalization power. Model evaluation step often provides good hint about which part of the model (or hyperparameters) to tweak, which leads to the last step of model tuning. Specialized techniques are available to assist model tuning. With this broad overview of the workflow, let’s now start learning about these techniques in detail.

3.2 Data Pre-processing: Data Transformation

Process data are seldom available in ‘ready-to-model’ format. The raw dataset may contain highly correlated variables or may have variables with very different value ranges. These data characteristics tend to negatively influence model training, and therefore, warrant data transformation to de-correlate or properly scale data. Sometimes, creating new features¹¹ out of provided variables can help to improve model accuracy. Overall, the aim of this pre-processing step is to transform given variables such that model training and model accuracy improves. Let’s look at some commonly employed transformation techniques.

Data centering & scaling

In process dataset, it is common to find variables with very different order and spread of values. For example, you may have a temperature variable that varies between 1000°C to 1500°C, a concentration variable varying between 0.01M to 0.1M, and another temperature variable that goes from 300°C to 350°C. While the order and spread of these three variables are quite different, they may exert similar influence on the model output. However, unscaled dataset can result in one variable being given more importance than the other or slow model training. Centering and scaling transform variables to bring them on an equal basis! While some algorithms like PCA necessarily require centered data, as a best practice, you should always center and scale your dataset unless you have strong reason against it.

The most common technique is standard scaling or standardization, where each variable (in training dataset) is transformed to 0 mean and standard deviation of 1. Within an ML script, you will transform the test data using the computed mean and standard deviation of the training dataset. The illustration below shows how sklearn facilitates standardization

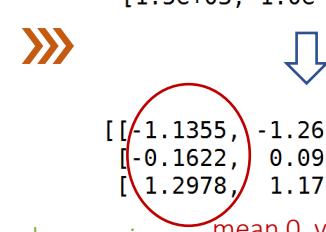
```
# import library, fit, and transform
import numpy as np
from sklearn.preprocessing import StandardScaler
X = np.array([[ 1000, 0.01, 300],
              [ 1200, 0.06, 350],
              [ 1500, 0.1, 320]])
scaler = StandardScaler().fit(X) # compute mean & std column-wise
X_scaled = scaler.transform(X) # transform using computed mean and std
```

[[1.0e+03, 1.0e-02, 3.0e+02],
[1.2e+03, 6.0e-02, 3.5e+02],
[1.5e+03, 1.0e-01, 3.2e+02]]

»»»

[[-1.1355, -1.2675, -1.1355],
[-0.1622, 0.0905, 1.2978],
[1.2978, 1.177 , -0.1622]]

mean 0. variance = 1



¹¹ Feature and (independent) variable mean the same thing. While in ML literature, ‘feature’ is used to denote variables which go into predicting outputs, the term ‘independent variable’ is commonly used for this in statistics.

Using Sklearn library classes



In (object-oriented) computer programming parlance, in the above code, we created an object-instance (scaler) of StandardScaler class¹² and set its attributes, `mean_` and `var_`, by calling the `fit()` method. They can be accessed as `scaler.mean_` and `scaler.var_`. The fitted object (scaler) was then used to scale X by calling the object's `transform` method. This is the general pattern we will use with Sklearn's classes: objects will be initialized, properties fitted, and methods called.

Another popular technique is min-max scaling, also referred to as normalization. Here, each variable is transformed to a common range of values (typically 0 to 1). For 0 to 1 normalization, each variable is subtracted by its minimum value and then divided by its range. This is accomplished in Sklearn as follows

```
# import library, fit, and transform
import numpy as np
from sklearn.preprocessing import MinMaxScaler
X = np.array([[ 1000, 0.01, 300],
              [ 1200, 0.06, 350],
              [ 1500, 0.1, 320]])
scaler = MinMaxScaler () # create object
X_scaled = scaler.fit_transform(X) # fit & transform
```

»»»

[[1.0e+03, 1.0e-02, 3.0e+02],
 [1.2e+03, 6.0e-02, 3.5e+02],
 [1.5e+03, 1.0e-01, 3.2e+02]]

↓

[[0. , 0. , 0.],
 [0.4 , 0.56, 1.],
 [1. , 1. , 0.4]]

min 0. max = 1

Although standard scaling is often the default choice, there is no straight-forward answer to which scaling method is better. Either of the methods will usually work fine. Standard scaling is preferred if variables are Gaussian distributed or there are (a few) extreme outliers which can skew the 'min' and 'range' estimates during normalization.

Robust centering & scaling

If your dataset has a lot of outliers, then both the aforementioned methods will result in improper scaling. For example, in Figure 3.2, we can see that standard scaling isn't able to center the 'normal' data around 0 due to the presence of outliers. In such situations, robust MAD scaling should be employed, where, instead of mean and standard deviation, their robust counterparts, median and MAD (median absolute deviation from the median), are used. Mathematically, MAD scaling for any variable x takes the following form

¹² A class is like a blueprint defining the properties and behavior (functions) that are common among its objects. In OOP architecture, we can define a class once and then create its objects as many times as we want.

$$x_{MAD \text{ scaling}} = \frac{x - \text{median}(x)}{\text{MAD}(x)}$$

Figure 3.2 shows that under MAD scaling, normal data does get centered around zero!

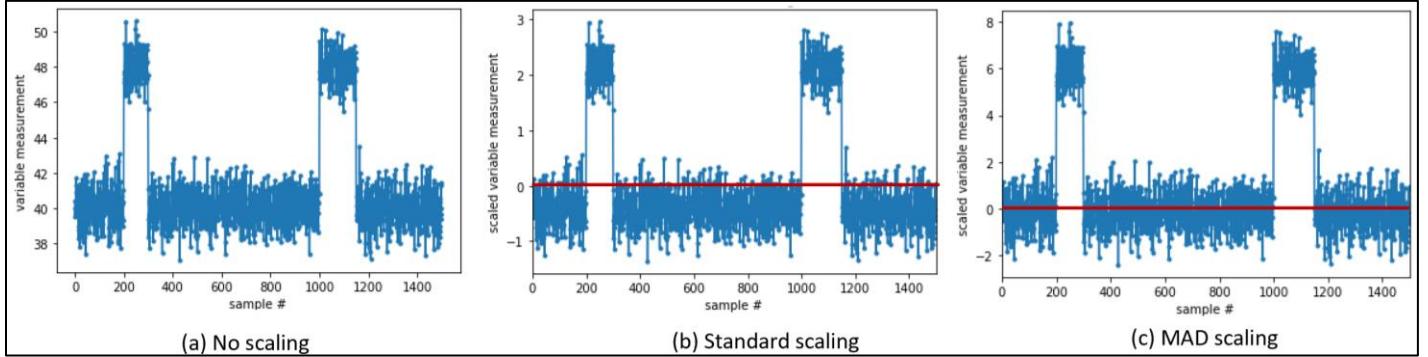


Figure 3.2: Application of standard and robust MAD scaling on data contaminated with outliers. Solid red line denotes the preferred centered location of 0. Complete code for generating these plots is available in online code repository.

Sklearn provides a RobustScaler class for robust scaling which is similar to MAD scaling, except that it uses interquartile range (IQR) instead of MAD as a robust estimator of data spread. Similar results can be expected with either of these robust scaling methods.

Feature extraction

Process systems have a lot of measurements which leads to increased data analysis complexity. However, you will often find that the original variable set can be reduced in dimensionality by extracting features which are combination of original variables and contain the same amount of information as the original variable set. This is a consequence of strong correlation among process variables.

Consider Figure 2 for example. Here, the three original variables (x, y, z) exhibit strong linear relationship to the extent that it is possible to transform (as shown) the 3D measurement space into a 2D (or even 1D) feature space while retaining most of the information about data variability. Accomplishing this in Sklearn is as simple as writing the following code

```
features = PCA(1).fit_transform(X_scaled) # extracting only 1 feature from scaled measurements
```

After feature extraction, rest of the analysis/model building is carried out using the extracted features. Specific techniques are available to decide optimal number of extractable features

for linear and nonlinear systems. Details on these techniques, concept of dimensionality reduction without ‘losing information’, PCA method, and other feature extraction methods such as ICA, FDA are covered in detail in Part 2 of the book. Apart from lower dimensionality, feature spaces may exhibit other nice characteristics such as uncorrelated or independent features, and, therefore, using these features instead of the original variables can generate better models.

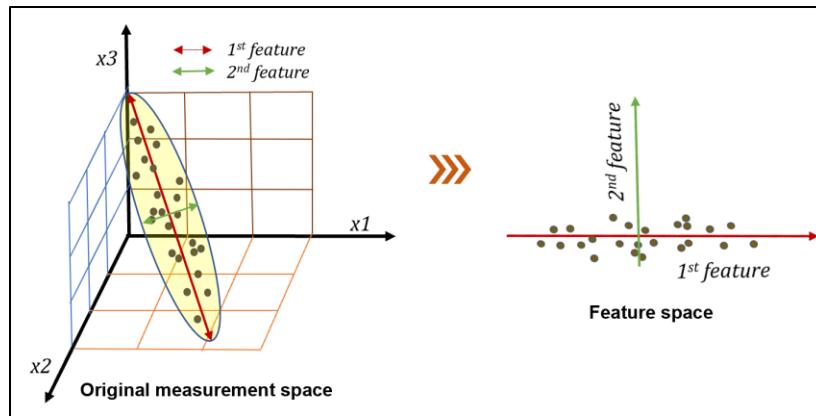


Figure 3.2: Extracting uncorrelated features from a 3D dataset

Feature engineering

While feature extraction is an unsupervised exercise (we do not explicitly specify the underlying correlation structure; algorithms extract relationships automatically), feature engineering involves careful and explicit transformation of variables which can assist ML algorithms in delivering better models. For example, if we examine the raw (x vs y) data in Figure 3.3, it is clear that a quadratic model may be needed. To accomplish this, we will engineer an additional feature (x^2) from the given x data and then fit a linear model between $\{x, x^2\}$ and y , as shown in the code below

```
# read raw data
import numpy as np
data = np.loadtxt('quadratic_raw_data.csv', delimiter=',')
x = data[:,0,None]; y = data[:,1,None]

# generate quadratic features
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(x) # X_poly: 1st column is x, 2nd column is x^2

# scale model inputs
from sklearn.preprocessing import StandardScaler
```

```

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_poly)

# fit linear model & predict
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_poly, y)
y_predicted = model.predict(X_poly)

```

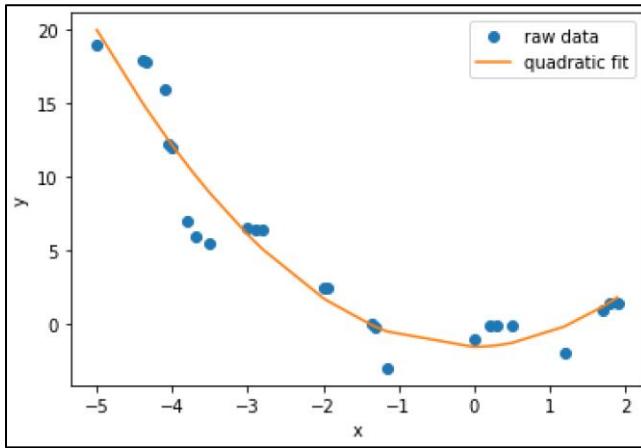


Figure 3.3: Linear model with quadratic feature

The advantage in the above example was that instead of doing a non-linear fit between y and x , we ended up doing just a simple linear fit. This is the power of feature engineering. It won't be an exaggeration to say that half of your task is done if you can engineer very relevant and informative features. Unfortunately, no well-defined generic approach currently exists for effective feature engineering. The task is system-specific and require domain knowledge. Nonetheless, let's look at a couple of feature engineering techniques that are commonly employed.

Mathematical transformations

These transformations make use of functions like log, square, square root, product, temporal aggregation (using min, max, standard deviation), sin, tanh, etc. to create custom features. As seen in the previous illustration, these transformations allow usage of simple models, while making the effective relationship between the original variables more complex. Since, the selection of proper mathematical function is the most crucial and still largely a manual task, automated feature engineering (AFE) is currently an active research area in machine learning. Quite often, AFE techniques involve generating an exhaustive list of features from commonly used mathematical functions and then performing variable selection to select potentially useful features. However, universally accepted, generically applicable AFE methods don't exist yet and this makes your domain knowledge-guided feature engineering very important!

One-hot encoding

Most ML modeling algorithms require numerical data. But sometimes you may find yourself working with categorical data: for example, one of the model input variable may have entries like ‘plant type A’, ‘plant type B’, ‘plant type C’, etc. To handle such situations, you can convert the categorical data into numerical values. A naive approach would be to simply replace ‘type A’ with 0, ‘type B’ with 1, ‘type C’ with 2, and so on. However, this approach is erroneous unless there is a natural order among the plant type categories (maybe plant types A/B/C represent small/medium/large categories of plants). ML models may end up giving more importance to plant type C relative to types A (& B) due to type C’s higher designated value of 2. The preferred approach is to use one-hot encoding, which does the following transformation

```
# import class, fit, and transform
import numpy as np
from sklearn.preprocessing import OneHotEncoder

x = np.array(['type A'],
            ['type C'],
            ['type B'],
            ['type C']))

ohe = OneHotEncoder(sparse=False) # sparse=False returns array
X_encoded = ohe.fit_transform(x) # x in numerical form
```

['type A']	1	0
['type C']	0	1
['type B']	0	1
['type C']	1	0

2nd column corresponds to type B category

As you can infer, a column has been generated for each unique plant type category (automatically identified by OneHotEncoder in the above code) and a binary value of 0/1 is assigned depending on occurrence of that category in a sample. For example, the 3rd sample is of ‘type B’ and therefore the 3rd entry in the 2nd column is assigned a value of 1.

Workflow automation via pipelines

In the previous example on linear modeling with quadratic features, we had to call fit and transform (or predict) methods multiple times for model training. These calls would be repeated for test dataset. For other models, you may have many more data transformers, such that it becomes slightly inconvenient to keep track of all transformations. To streamline this process, Sklearn provides a nice utility class called Pipeline, which allows us to chain together the transformers and estimators (predictive models) sequentially such that we need to call the fit method only once as shown in the code below for our quadratic model example.

```

# create pipeline for quadratic fit via linear model
# import relevant classes
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

# add transformers and estimators sequentially as list of tuples
# the names 'poly', 'scaler', 'model' can be used to access the individual elements of pipeline later
pipe = Pipeline([('poly', PolynomialFeatures(degree=2, include_bias=False)),
                 ('scaler', StandardScaler()),
                 ('model', LinearRegression())])

# fit pipeline and predict
pipe.fit(x, y)
y_predicted = pipe.predict(x)

```

The above code gives the same results as our previous code without pipeline and is more concise (and potentially easy to manage). Figure 3.4 shows the sequence of steps that got executed behind-the-scenes when we called the fit and predict methods of the pipeline. Really convenient, isn't it? Note that a pipeline does not have to include an estimator as its last element – you can only have transformers and call `pipeline.transform()` to transform your data through all the included transformers.

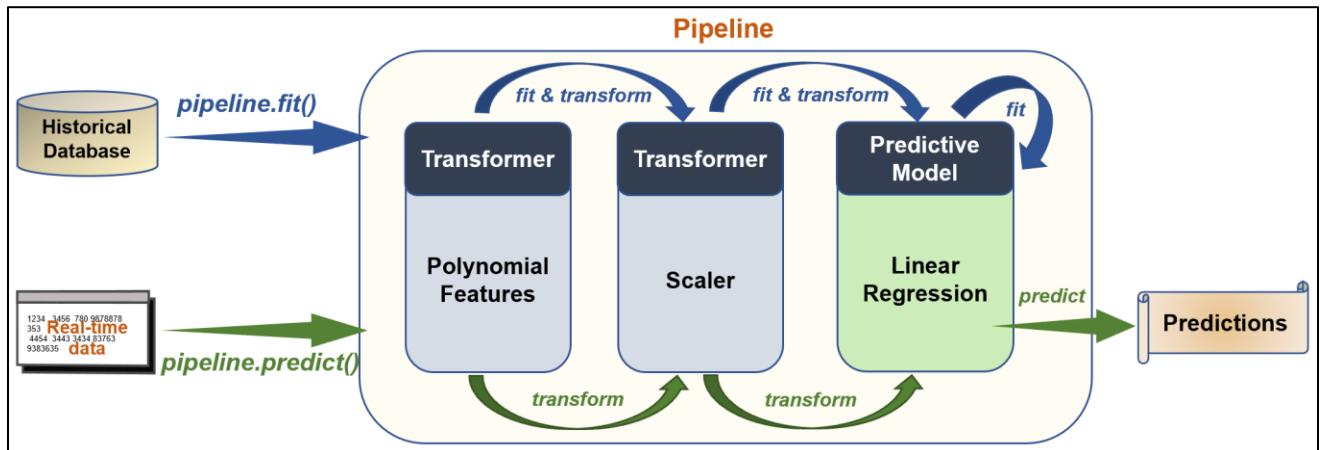


Figure 3.4: Streamlining ML workflow by combining transformers and predictors via pipeline

This completes our quick look into data transformation techniques. The takeaway message is that before dumping all the raw data for model training, you should always give some thought on what transformation you can apply to the raw data that can help the ML algorithm provide a good model. Carefully designed transformation can alleviate the need for large number of

training samples, allow usage of simple models, and reduce model training time. Next, we will see what techniques can be used to evaluate how well our model is performing.

3.3 Model Evaluation

After model parameters have been fitted, we need some means of assessing the model performance to see how well (or poorly) the model fits the training data. For this purpose, commonly computed metrics for regression and classification tasks are listed below.

Regression	Classification
<ul style="list-style-type: none">• Mean Squared Error (MSE)• Mean Absolute Error (MAE)• Root Mean Squared Error (RMSE)• R Squared (R^2)	<ul style="list-style-type: none">• Confusion Matrix• Accuracy• Precision, Recall, F1 Score• True positive rate (TPR), False positive rate (FPR)

Table 1: Common means of evaluating model performance

While it is easy to write our own python code to compute these metrics, it is even easier to use the `sklearn.metrics` module where these metrics are all implemented. We can import the desired metric class from the module and use it as shown below our quadratic fitting example.

```
# performance metric
from sklearn.metrics import r2_score
print('Fitting metric (R2) = ', r2_score(y, y_predicted))

>> Fitting metric (R2) = 0.91
```

Let us now take a quick look into the different metrics listed in Table 1.

Regression metrics

R-squared (R^2) is a good measure, technically for linear models, to determine how well the predicted output (\hat{y}) compares to the actual y . As seen in the metric formula table below, R^2 measures how much of variability in the output variable can be explained by the model. An R^2 of 1 indicates a perfect fit.

Regression Metric	Formula
MSE	$\frac{\sum(y - \hat{y})^2}{N}$
MAE	$\frac{\sum y - \hat{y} }{N}$
RMSE	$\sqrt{\frac{\sum(y - \hat{y})^2}{N}}$
R squared	$1 - \frac{\sum(y - \hat{y})^2}{\sum(y - \bar{y})^2}$

Table 2: Formulae for common regression metrics. The summation is carried out over the N samples. \bar{y} is mean of y

R^2 does not directly convey the magnitude of prediction errors. For this, the other metrics can be used. RMSE is the most popular metric for evaluating regression models and is computed as the square-root of the average of prediction errors. A related metric is the MSE which does not take the square-root. MAE is similar to MSE, except that it considers the absolute value of errors. Any of these metrics can be used to compare different models: smaller the values of these metrics (except R^2), better is the model fit.



Several Sklearn models have their own score methods which computes an evaluation metric suitable for the problem they are designed to solve. For example, the `score()` method of `LinearRegression` object returns the R^2 metric. Pipelines have a score method as well which simply executes the score method of the final model. For our quadratic fitting example, it can be used as follows

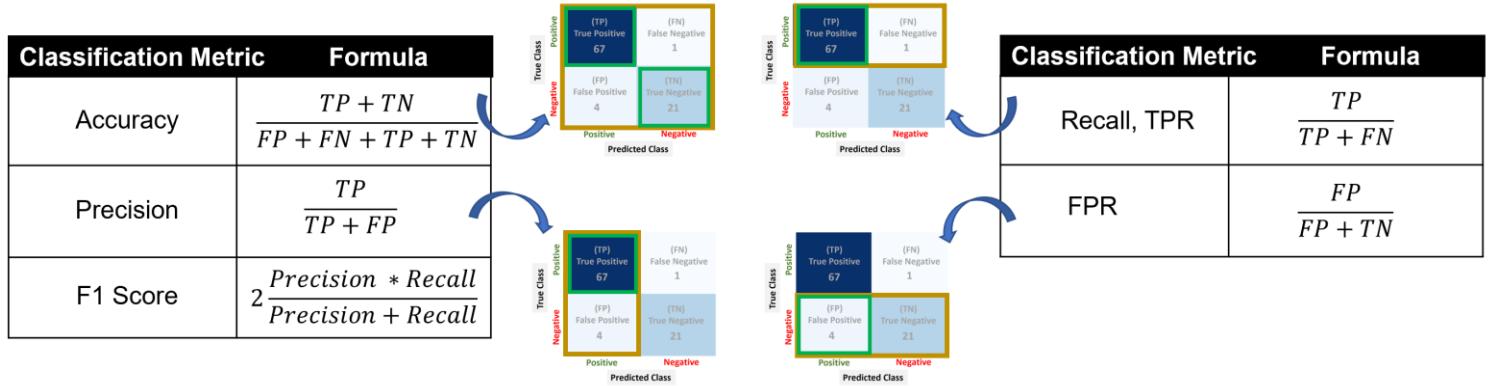
```
# compute metric using score method
R2 = pipe.score(x, y) # no need to import metrics module
```

Classification metrics

While in regression problems we care about how close the predicted values are to the actual values, in classification problems we care about the perfect match between the actual and predicted labels. Before we look at different classification metrics, let's first learn about a confusion matrix. The figure below shows a sample confusion matrix for a binary classification task; this matrix provides a comprehensive overview of how well a model has correctly (& incorrectly) classified samples belonging to the positive and negative classes. The shown matrix indicates that out of 25 (24 + 1) samples that actually belonged to negative class, the model correctly labeled 21 of those.

True Class	Predicted Class	
	Positive	Negative
Positive	(TP) True Positive 67	(FN) False Negative 1
Negative	(FP) False Positive 4	(TN) True Negative 21

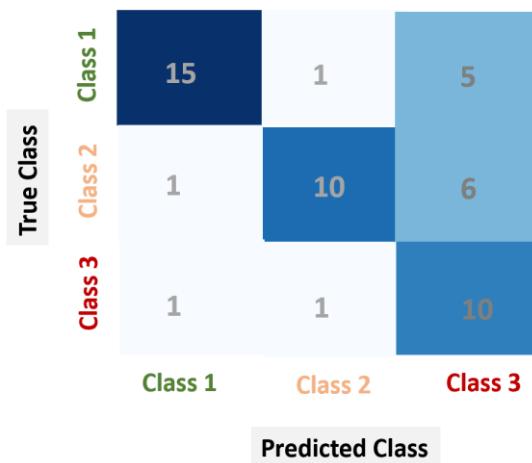
Confusion matrix can be used to compute several single-valued metrics, summarized in the metric formula table below. The most straightforward metric is ‘accuracy’ which simply communicates the ratio of true predictions to the total number of predictions. Accuracy can, however, be a misleading metric for imbalanced datasets (where more samples belong to one particular class). For example, in our example, overall accuracy is ~95% (88 out of 93), but we can see that the accuracy in correctly detecting negative class is only 84% (21 out of 25). Therefore, several other metrics are frequently employed to convey different aspects of model’s performance.



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. FPR (false positive rate) looks at the fraction of negative samples that get classified as positives. In our example, a high FPR of 0.16 slightly spoils the very rosy picture being presented by accuracy, precision, and recall numbers!!

For a multiclass problem, the confusion matrix would look something like the plot below where there is a row and column for each class type. Several important clues can be obtained from a multiclass confusion matrix plot. For example, in the plot below, it seems that some samples belonging to classes 1 & 2 tend to get classified as class 3 which suggest that some additional features should be added to help the model better distinguish between classes 1 & 2 and class 3. Metrics like precision can be computed by one vs all approach where you would choose one particular class as a positive class and the rest of the classes as a negative class.



Holdout method / Cross-validation

While the performance metrics computed using training data tell us how good our model fitting is, they are not true indicators of how well the model would perform for unseen data. For an unbiased generalization assessment, performance metrics should always be reported for a ‘test’ dataset that is not used during model training. This approach is called cross-validation. It’s great if you already have a separate test dataset, otherwise the best practice is to split the available dataset into training and test sets as shown below.



Setting aside a portion of data for cross-validation is called holdout method. A good thumb-rule is to keep 20% of data in test set. Holdout can be accomplished in Sklearn using `train_test_split` utility function as shown below for our quadratic fit example

```
# separate 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=1)

# check dataset sizes
print('Number of samples in training set: ', x_train.shape[0])
print('Number of samples in test set: ', x_test.shape[0])

>>> Number of samples in training set: 20
Number of samples in test set: 5

# fit pipeline and predict
pipe.fit(x_train, y_train)
y_predicted_train = pipe.predict(x_train)
y_predicted_test = pipe.predict(x_test)

# performance metrics
from sklearn.metrics import mean_squared_error as mse

print('Training metric (mse) = ', mse(y_train, y_predicted_train))
print('Test metric (mse) = ', mse(y_test, y_predicted_test))

>>> Training metric (mse) = 3.79
Test metric (mse) = 3.56
```

The `random_state` parameter in the `train_test_split` function is used to shuffle the dataset before applying the split. Shuffling dataset is a good practice to ensure that there is no unintentional data pattern between training and test datasets. For example, assume a dataset comes from a manufacturing plant and is sorted by production rates: without shuffling, we may end up with data only with low production rates in the training set, which is undesirable.

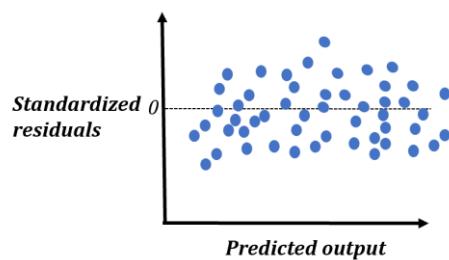
For classification problems, there is another important consideration to make. Ideally, we would like to ensure that the same class proportion that exists in the original dataset is retained in the training and test datasets. This approach is called stratified splitting and can be achieved through the `stratify` parameter in the `train_test_split` function. For example, we can pass a vector of fault / no-fault labels to this parameter and `train_test_split` function will ensure that the ratio of faulty and non-faulty samples is same in both training and test datasets.



After an estimate of model performance has been made using test data, the model may be re-trained, if desired, using the complete dataset. Since, an implicit assumption is always made that both the training and test datasets follow the same distribution, the model's generalization performance will only get better when re-trained with more data.

Residual analysis

Residual analysis is another way to validate the adequacy of (regression) models. It involves checking the model errors / residuals to ensure there are no undesired patterns in them. Presence of any obvious trend in residuals indicates a need for further data pre-processing or change of model. A convenient way to study the residuals is to simply draw scatterplots. Ideally, residuals would be small, centered around zero, independent of model inputs and outputs (predicted and measured), and uncorrelated in time (for dynamic models). An ideal residual plot would look something like this



In the above plot, no systematic pattern is apparent. If systematic patterns are present in residual plot then it implies that the model has not been able to extract all systematic pattern in the dataset. The specific patterns observed also often provide clues on potential remedies. Figure 3.5 shows some examples of 'bad' residual plots.

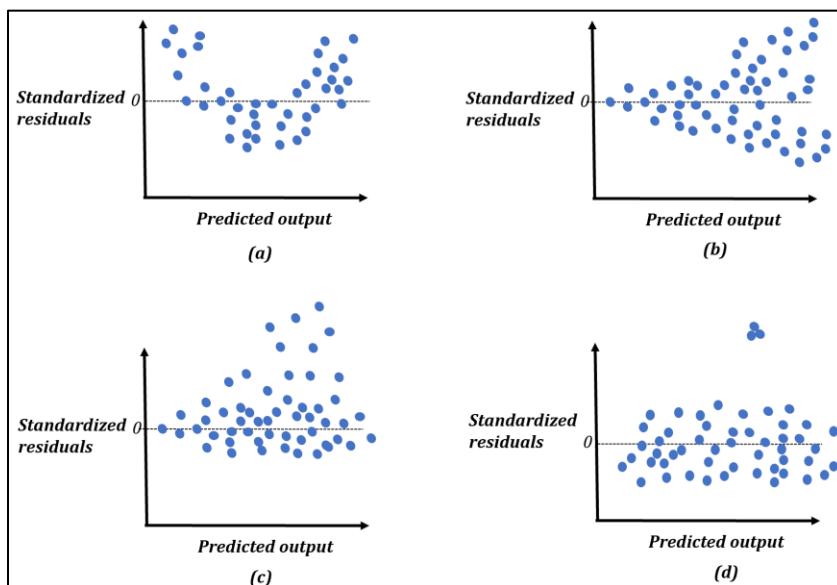


Figure 3.5: Examples of undesirable residual plots

In the above subplot (a), residuals show a nonlinear pattern which immediately conveys that the model is not able to adequately capture nonlinearities in the data. Nonlinear features may need to be included or a different model that can capture complex nonlinearities is warranted. In subplot (b) heteroscedasticity is apparent as residuals get larger for higher value of predicted output. Fixing this issue often entails appropriate transformation of variables such as log transformation. Heteroscedasticity may also indicate that a variable is missing. In subplot (c), while most of the samples show small residuals, several samples have high positive residuals. The suggested remedy in this case is the same as that for heteroscedasticity. Finally, in subplot (d), a group of samples appear as outliers. These outliers may need to be removed or further analyzed to check if these are normal samples; if samples turn out to be normal then model may need to be changed to account for the presence of these distinct groups of data.

For dynamic dataset, autocorrelation and cross-correlation (between residuals and past input/output data) function plots are used to check if the residuals are uncorrelated. Residual histograms or normal probability plots are also used to check if the residuals are normally distributed. If uncorrelatedness or normality assumptions are found violated, these plots, like scatter plots, give crucial hints on potential remedies.

We can see that residual analysis is a powerful diagnostic tool. As a best practice, you should always check the residual plots to ensure no modeling assumptions have been violated or no obvious room for improvement exists.

3.4 Model Tuning

Before we send our model for generalization assessment using test dataset, we need to ensure that the obtained model is the best model we can obtain. Or, if our model does not provide adequate generalization performance, some model adjustments are warranted. These tasks fall under the ‘model tuning’ step of the modeling workflow, and primarily involve optimizing values of model hyperparameters. *Model hyperparameters, unlike model parameters (which are the unknowns that are estimated during model fitting), are model configuration variables which need to be specified before model fitting.* For our quadratic fitting example, the weight coefficients of the linear model are the model parameters and the polynomial degree is the hyperparameter. For ANN models, number of neurons and layers are important hyperparameters, while weight coefficients are estimable model parameters. Finding optimal hyperparameter settings is also referred to as model selection in the ML literature.

Figure 3.6 shows the resulting model curves when our pipeline is fitted with different polynomial degrees. We can see that the model with high degree fits the training data better, but the fitting is a bit ‘too much’! We would not be comfortable accepting this model although it has very low training error. How do we then select the optimal hyperparameter values before we make the final evaluation using test data? How can we diagnose if our model is fitting data too much or too less? What steps we can take to ensure that neither of these two undesirable situations occur? We will answer these important questions in the next few subsections.

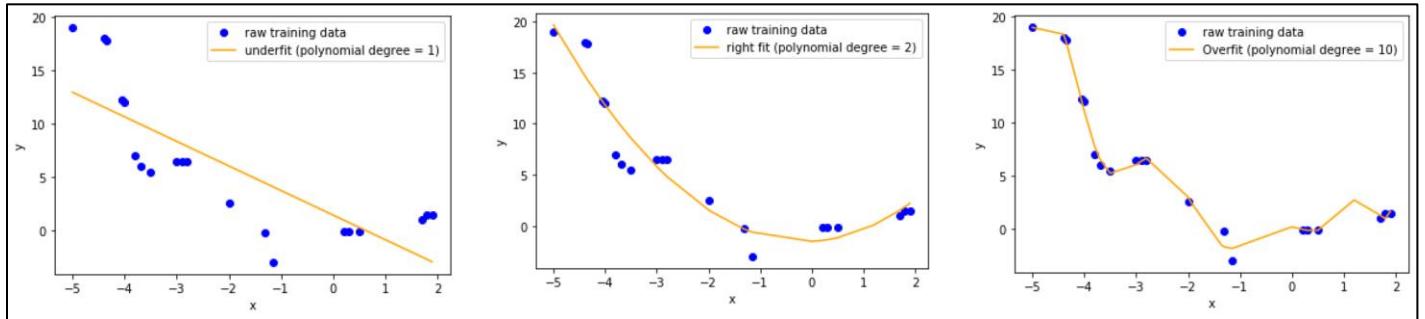


Figure 3.6: illustration of underfitting, ‘just-right’ fitting, and overfitting

Overfitting & underfitting

Underfitting occurs when our model is not able to capture the systematic relationships between predictors and response adequately (1st model in Figure 3.6) while overfitting occurs when the model captures even the non-systematic relationships such as noise (3rd model in Figure 3.6). As we increase model complexity by adjusting hyperparameter values, we move from underfitting end of the fitting spectrum to the overfitting end. A common way to visualize this is to plot validation curve using which we can also select our optimal hyperparameter value.



Overfitting is often the reason for poor performance of ML models. Overfitting is not necessarily only associated with ‘complex models’, and even simple linear regression models can overfit data (for example, if you have many irrelevant predictors or low number of training samples). Therefore, you should always pay careful attention that your model is not overfitting.

To plot validation curve, we will use a dataset that is kept separate from the data used for model fitting. We will call it validation dataset and the data used to fit the model as fitting or estimation dataset (you may wonder, ‘what happened to the test dataset?’ Hold onto that question, we will explain that shortly). Now, validation curve is simply a plot of model error (or any performance metric) for fitting and validation dataset for different hyperparameter values as shown in Figure 3.7. When hyperparameters are set such that model does not have enough

complexity or ‘handles’ to be able to describe the inherent structure in data, modeling errors will be large for both fitting and validation dataset leading to underfitting. As model complexity is increased, error on fitting dataset will decrease, however, the resulting model may not generalize well to the validation data leading to overfitting. Therefore, overfitting is characterized by small fitting errors and large validation errors. The sweet spot where both the fitting and validation errors are small, and validation errors just start increasing corresponds to the optimal hyperparameters.

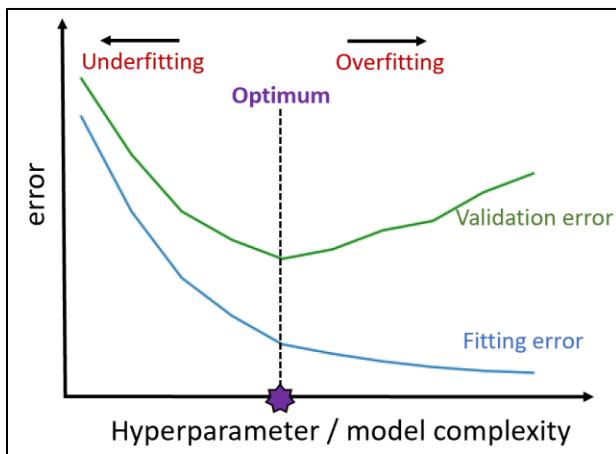


Figure 3.7: Typical validation curve

If your model is underfitting, your recourse should be to increase model complexity (for example by adding more depth in ANNs or using higher degree polynomials). You may explore adding more informative features. For overfitting, constrain model flexibility to prevent it from learning noise or random errors in fitting dataset. Removing irrelevant features or adding more fitting data can also help. If the optimal validation error is still very high, then a change of model may be needed (for example, using SVMs instead of decision trees).



The problems of underfitting and overfitting are also referred to as problems with high bias and high variance, respectively. High bias refers to the problem of model estimates being consistently far away from true values. On the other hand, if your model is overfitting, then your model estimates for the same validation dataset would vary considerably when different datasets are used for model fitting, leading to high variance in model estimates.

To generate the validation curve, you can use Sklearn’s utility function `validation_curve` available in the `model_selection` module or write your own code as shown below.

```
# separate fitting and validation data
from sklearn.model_selection import train_test_split
```

```

x_fit, x_val, y_fit, y_val = train_test_split(x, y, test_size=0.2, random_state=1)

# assess performance on validation sets for different hyperparameter values
from sklearn.metrics import mean_squared_error as mse
fit_MSEs = []
validation_MSEs = []

for poly_degree in range(1,6):
    # set hyper-parameter value
    pipe['poly'].degree = poly_degree
    # fit & predict
    pipe.fit(x_fit, y_fit)
    y_pred_fit = pipe.predict(x_fit)
    y_pred_val = pipe.predict(x_val)
    # compute scores and append
    fit_MSE = mse(y_fit, y_pred_fit)
    validation_MSE = mse(y_val, y_pred_val)
    fit_MSEs.append(fit_MSE), validation_MSEs.append(validation_MSE)

# plot validation curve
from matplotlib import pyplot as plt
plt.figure()
plt.plot(np.arange(1,6), fit_MSEs, 'b--', label='fitting MSEs')
plt.plot(np.arange(1,6), validation_MSEs, 'g--', label='validation MSEs')
plt.legend(), plt.xlabel('Polynomial degree'), plt.ylabel('MSE')

```

Figure 3.8 shows the validation curve for our quadratic fitting example dataset. Although the distinction between fitting and validation MSEs are not very pronounced, 2nd degree seems to be the optimal hyperparameter value.

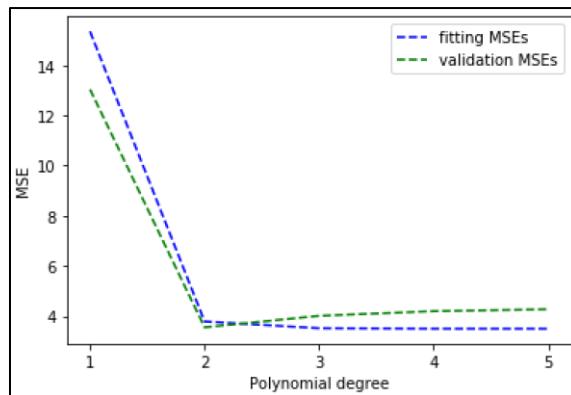
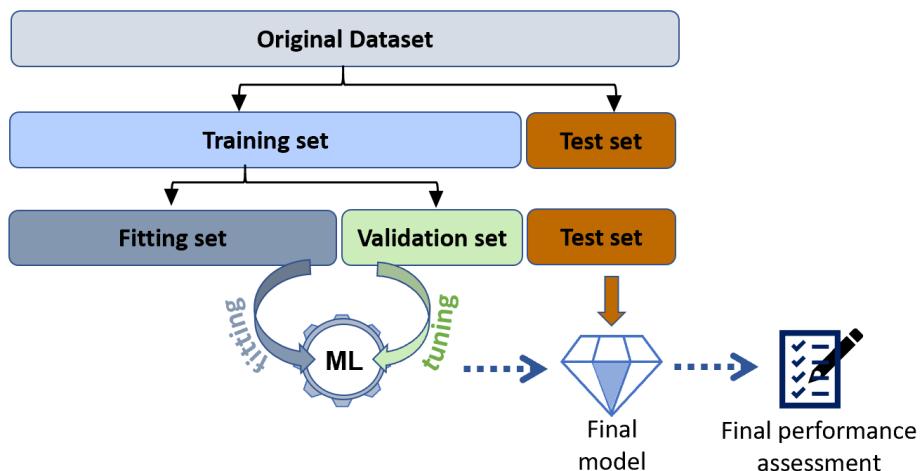


Figure 3.8: Validation curve for quadratic fitting example

A few more interesting things to note here. While validation MSEs are expected to be higher than fitting MSEs in general, here, we find them to be lower for the first 2 polynomial degrees. Also, if you extend this validation curve for even higher degrees, you will find validation MSEs decreasing instead of continuously increasing! So, does this mean that we can't trust inferences made using validation curves? You can actually trust these curves. What is happening here is that due to small dataset size, the validation dataset just happened to have data-points which show these unexpected behaviors. Infact, for small datasets, specialized multi-fold validation techniques exist which we will learn shortly.

Train/validation/test split

We used away the dataset that we had kept aside from the fitting dataset for model validation. We can no longer use it as a test dataset for generalization assessment (since this dataset has already been 'seen' by the model and thus is no longer an unseen dataset). The solution is to set aside another chunk of dataset as shown below. This approach is called 3-way holdout method.



To implement 3-way holdout, we need to call `train_test_split` function twice as shown below

```

# train-validate-test split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)
x_fit, x_val, y_fit, y_val = train_test_split(x_train, y_train, test_size=0.3, random_state=1)

print('Number of samples in fitting set: ', x_fit.shape[0])
print('Number of samples in validation set: ', x_val.shape[0])
print('Number of samples in test set: ', x_test.shape[0])

>>> Number of samples in fitting set: 14
Number of samples in validation set: 6
Number of samples in test set: 5
  
```



After model selection is completed, you may, if desired, retrain your model with the selected optimal hyperparameter settings on the combined fitting and validation dataset. Moreover, as mentioned before, after model performance assessment has been made using test data, the final model may be obtained by retraining on the entire dataset.

K-fold cross-validation

The 3-way holdout method is an elegant way of ensuring that model tuning and final assessment are based on unbiased estimates of model performance (i.e., using data not seen during model fitting and model training, respectively). However, there are several drawbacks as well with this method. If our original dataset is not large enough, we may not have the luxury of having 3 distinct subsets of data. Moreover, the performance estimates can be highly influenced by what data-points makeup the subsets and thus may lead to unfortunate selection of hyperparameters and final assessment.

The popular alternate solution is *k*-fold cross validation. The idea is simple. Instead of having a single test (or validation) subset, multiple subsets are created, and overall performance estimate is computed as the average of performances on individual subsets. Figure 3.9 illustrates this idea where *k*-fold cross validation is used for both final performance assessment and model tuning.

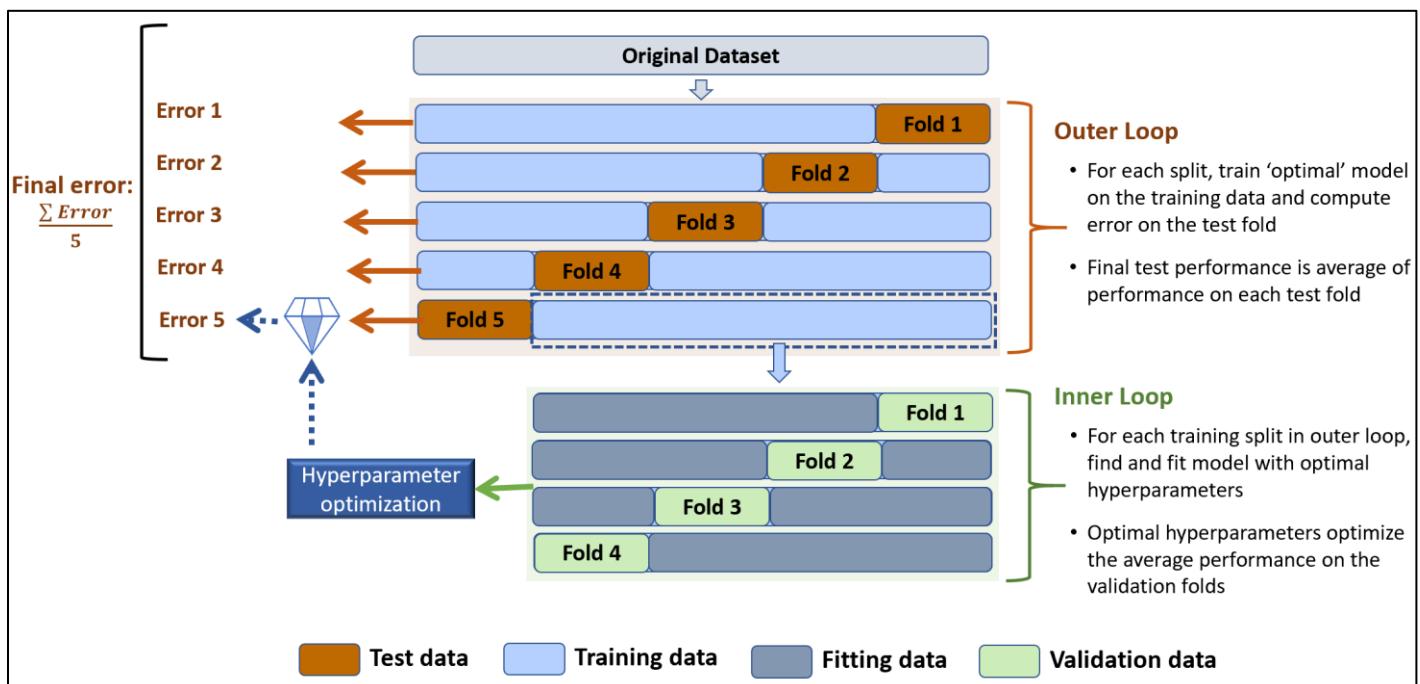


Figure 3.9: Nested *k*-fold cross validation for model tuning (inner loop) and final performance assessment (outer loop)

To understand Figure 3.9, let's consider the outer loop first

- 5-fold cross validation is employed for final performance assessment
- The original dataset is split into 5 folds and 5 separate models are trained
- For each model, one of the folds is used as test data and the rest of the data are used for model training
- The overall model performance is simply the average performance of the 5 models

For model training in the inner loop

- 4-fold cross validation is employed, i.e., the training data is split into 4 folds
- For each hyperparameter set, 4 separate models are fitted using the fitting datasets
- The overall validation performance is simply the average performance of the 4 models on their respective validation folds
- The hyperparameter values that optimize the overall validation performance is selected and the model is re-trained using the complete training data before being sent to the outer loop.

Very often, k -fold cross validation is only employed for model tuning and the final performance assessment is made using only a single test holdout set. This conventional cross-validation approach is illustrated in Figure 3.10.

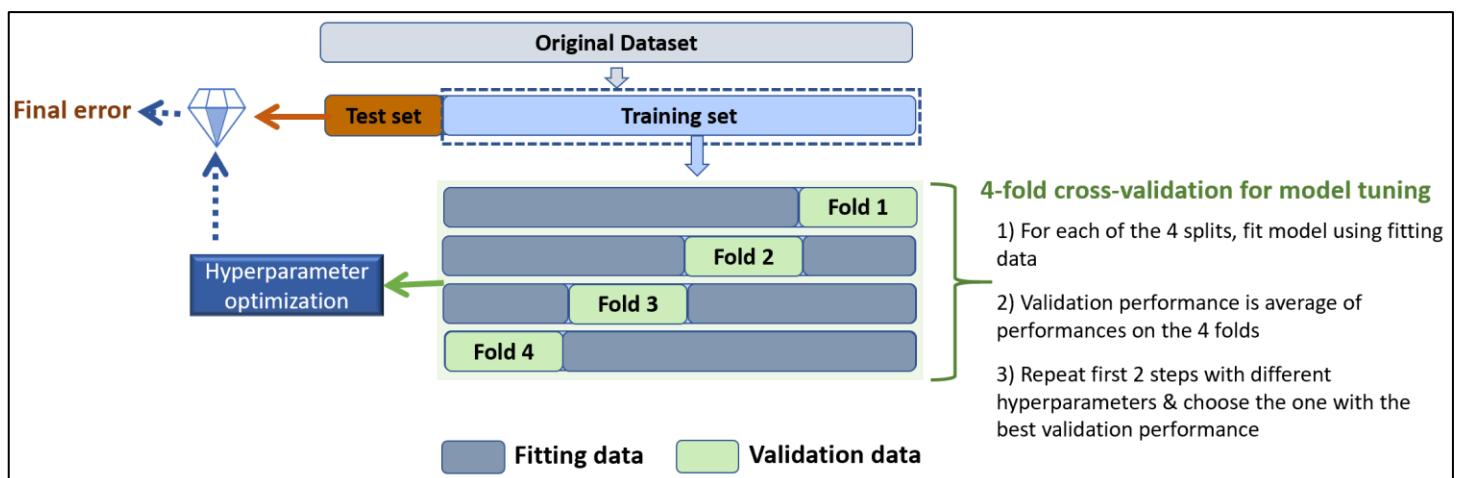


Figure 3.10: K -fold cross validation for model tuning

Let's apply k -fold cross validation technique to find the optimal polynomial degree with our illustration dataset. We will use the KFold utility function provided by Sklearn which returns the indices of data-points that belong to the fitting & validation subsets in each of the k splits.

```

# separate training and test data
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)

# generate k=3 folds on training data
from sklearn.model_selection import KFold
kfolds = KFold(n_splits = 3, shuffle = True, random_state = 1)

# compute validation performances using the K folds
from sklearn.metrics import mean_squared_error as mse
overall_fit_MSEs = [] # overall errors for different polynomial degrees
overall_val_MSEs = []

max_polyDegree = 5
for poly_degree in range(1,max_polyDegree+1): # loop over hyperparameters
    pipe['poly'].degree = poly_degree

    split_fit_MSEs = [] # errors for different splits
    split_val_MSEs = []

    for fit_indices, val_indices in kfolds.split(x_train): # loop over splits
        x_fit = x_train[fit_indices]
        y_fit = y_train[fit_indices]
        x_val = x_train[val_indices]
        y_val = y_train[val_indices]

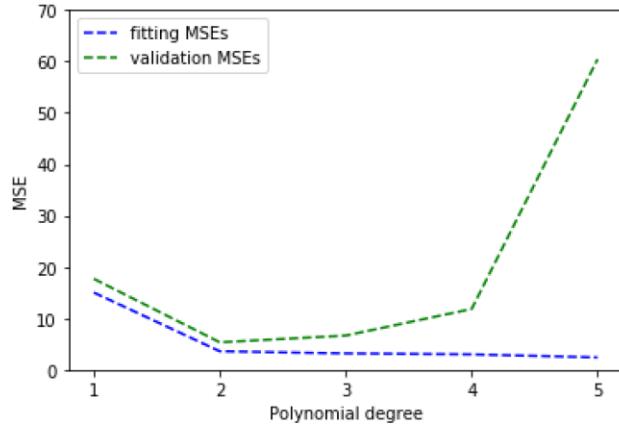
        # fit & predict
        pipe.fit(x_fit, y_fit)
        y_pred_fit = pipe.predict(x_fit)
        y_pred_val = pipe.predict(x_val)

        # compute average error and append
        split_fit_MSEs.append(mse(y_fit, y_pred_fit))
        split_val_MSEs.append(mse(y_val, y_pred_val))

    overall_fit_MSEs.append(np.mean(split_fit_MSEs))
    overall_val_MSEs.append(np.mean(split_val_MSEs))

```

Figure below shows the resulting validation curve which is more along the expected lines. Note that Sklearn provides a `cross_val_score` function which can be used to replace the loop over the splits in our code above.



K-fold cross-validation is the default cross validation techniques inbuilt in several of Sklearn's functions such as `validation_curve`. The downside to this technique is high computational cost as model fitting is carried out multiple times. Therefore, for deep learning-based solutions, which anyway employs large datasets, 3-way holdout method is often the preferred choice.

Choosing number of folds

The typical (and often the default) value of k in k -fold cross-validation is 10. For large datasets, smaller value ($k = 3$ or 5) can be used. When we increase k , less samples are put in the validation set; this results in unpreferable higher variance in the validation scores among the different splits. An extreme scenario is when number of folds equals the number of training samples. This is referred to as leave-one-out cross validation and is employed when number of training samples is very limited.

Regularization

If your model is overfitting, one of the things to do is to check the values of the model parameters (weight coefficients for linear models). You may find some or all of them with very large values. This is problematic because the model becomes very sensitive to input variables with high coefficients, i.e., model predicts big response changes for even very small variations in these input variables. This is unlikely to be true for real process systems. For example, following are the model coefficients with 10-degree polynomial features for our illustration dataset. Look at the higher degree coefficients

```
# get model coefficients
print(pipe['model'].coef_) # does not include intercept term

>>> [-1.96123e+00 -1.70142e+01  1.61112e+02  5.53503e+02 -1.47185e+03 -5.24500e+03
9.88824e+02 1.58830e+04 1.72273e+04 5.73349e+03]
```

Bayesian Information Criterion (BIC) for model selection

If you want to utilize all your training data for model fitting or validation subset cannot be set aside for model selection (e.g., when using Gaussian mixture models), BIC technique can be used which provides an optimal trade-off between model complexity (or number of model parameters) and model accuracy. We have seen that model accuracy on training data can be increased by adding more model parameters, but this can also lead to overfitting. BIC resolves this conundrum by minimizing a metric that includes a penalty term for the number of model parameters.

Mathematically, $BIC = n_p \ln(n) - 2\ln(\hat{L})$, where

n_p equals number of estimated parameters

n equals number of data-points or observations

\hat{L} equals the optimal value of likelihood function (model accuracy)

We can see that a model with high accuracy/likelihood and fewer parameters will result in low BIC value, and hence, optimal model corresponds to the minimum BIC value. BIC owes its popularity for its effectiveness and low computational cost: It has been shown theoretically that for sufficiently large training data, if the correct model is included among the set of candidate models, then BIC is guaranteed to select it as the best model. We will see an application in Chapter 8.

Another popular model selection metric is AIC (Akaike information criterion). The penalty term is smaller in AIC. Consequently, AIC tends to select more complex models compared to those selected by BIC. Do remember that more complex models might overfit while simpler models might underfit. Therefore, selection between AIC and BIC methods is problem dependent.

Therefore, to keep model parameters ‘simple’, modeling algorithms are tweaked to decrease the likelihood of overfitting. This modulation is called regularization and is a widely used approach to tackle overfitting. You should always employ some form of regularization in your model.

Common mechanisms of regularization

In one of the forms of regularized model, the objective function (which is minimized during fitting) includes an additional term that is proportional to summation of square of model coefficients (θ) as shown below

$$J_{\text{regularized}}(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_i \theta_i^2$$

The additional term penalizes the algorithm for choosing high coefficient values. Therefore, during model fitting, the algorithm will be forced to minimize fitting error while keeping coefficients small. This version of regularization is called L2 or ridge regularization. The ' α ' term is a hyperparameter which modulates the magnitude of the penalty term.

In another form of regularization, the objective function is modified as follows

$$J_{\text{regularized}}(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_i |\theta_i|$$

Here, the penalty term is simply proportional to the summation of absolute value of the coefficients. This is called L1 or lasso regularization. Let's see how these mechanisms help us with the overfitting problem with our 10th degree polynomial fitting. L1 / L2 can be implemented by just switching the linear model in the pipeline with the regularized linear models as shown below

Ridge regression

```
from sklearn.linear_model import Ridge
pipe_L2 = Pipeline([('poly', PolynomialFeatures(degree=10, include_bias=False)),
                    ('scaler', StandardScaler()),
                    ('model', Ridge(alpha=0.1))])
```

Lasso regression

```
from sklearn.linear_model import Lasso
pipe_L1 = Pipeline([('poly', PolynomialFeatures(degree=10, include_bias=False)),
                    ('scaler', StandardScaler()),
                    ('model', Lasso(alpha=0.1))])
```

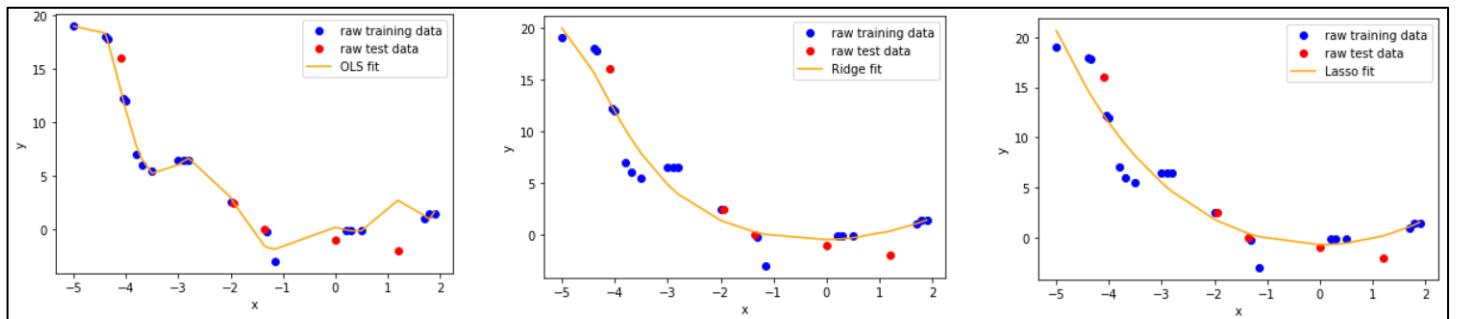


Figure 3.11: Non-regularized and regularized polynomial fitting

Figure 3.11 confirms that regularization works!! To see what is happening under the hood, let's check the model coefficients of the regularized models

```
# get model coefficients
print(pipe_L2['model'].coef_)
```

```
>>> [ 0.27326  3.49270 -0.35861  0.59074 -1.20522  1.90652 -1.80640  0.86690  0.87244
-3.20685]
```

```
print(pipe_L1['model'].coef_)

>>> [-0. 4.38109181 -0.09273215 1.679048 -0. 0. -0. 0. 0. -0. ]
```

While both L1 and L2 regularizations ensure that model coefficients remain small, Lasso actually has pushed several coefficients to zero. This is representative of the general behavior of Lasso which is often used for variable selection as the coefficients of irrelevant variables get set to zero. The advantage of L2 is that it is computationally favorable as the penalty term is differentiable unlike L1's penalty term. The choice between L1 and L2 is problem dependent and requires trial-and-error.

L1 and L2 regularizations can be combined into elastic-net regularization. Not all forms of regularizations involve putting penalty terms in objective function. You will study methods like dropout, batch normalization, early stopping in Chapter 11, which help to ensure model coefficients remain 'stable' to avoid overfitting.



You will have noticed that we have talked about regularizing only the weight coefficients and not the intercepts. This is true in general. Large intercepts (or biases in neural networks) will not cause large response changes upon small variations in inputs, and therefore, regularizing them is often not needed.

Choosing value of α

The ' α ' term is an important hyperparameter which should be carefully specified before model fitting. Very small α will make regularization ineffective potentially leading to overfitting. Very high value will push the model coefficients to zero causing underfitting. The optimal value is difficult to know before-hand and need to be found via cross-validation. However, a general recommendation is to keep α low rather than high.

Hyperparameter optimization via GridSearchCV

Previously we used validation curves to find the optimal hyperparameters. Sklearn provides a GridSearchCV utility function that we can leverage to do this task more conveniently. GridSearchCV takes lists of potential hyperparameter values and does an exhaustive search to find the optimal combination of hyperparameter values. The code below shows the implementation.

```
# separate training and test data
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)
# find optimal hyperparameter via GridSearchCV
```

```

from sklearn.model_selection import GridSearchCV

param_grid = {'poly_degree': np.arange(1,6)}
gs = GridSearchCV(pipe, param_grid, scoring='neg_mean_squared_error', cv=3)
gs.fit(x_train, y_train)

print('Optimal hyperparameter:', gs.best_params_)

>>> Optimal hyperparameter: {'poly_degree': 2}

```

As expected, GridSearchCV also returns the optimal polynomial degree as 2. Note that we are using 3-fold cross validation here as specified via cv=3. By default, GridSearchCV refits the estimator on the complete supplied dataset using the best found hyperparameters. The best estimator can be obtained via `gs.best_estimator_` for subsequent predictions (predictions can also be done via `gs.predict` function). In the above code we explicitly supplied mean squared error as the scoring metric; the default behavior is to use the estimator's default scorer which would be R-squared for linear regression in Sklearn.

This concludes our quick look into the standard ML model tuning approaches and ML modeling workflow. You now know how raw data should be transformed before model fitting, how to evaluate and report model's performance, and how to find optimal model hyperparameters during model tuning.

Summary

In this chapter, we learnt the best practices of setting up a ML modeling workflow. During the course of learning the workflow, we saw several nice utilities (pipelines, k -fold, grid-search, etc.) provided in Sklearn library which make ML scripting easier and manageable. The takeaway message from this chapter should be that ML modeling is a collaborative work between you and machine. This chapter exposed you to your side of the task!!

Chapter 4

Data Pre-processing: Cleaning Process Data

A universal statement about machine learning is that more data is always better! However, not all data-points and/or variables in your dataset are necessarily good for your model. You may wonder why? The reason is that industrial process data inevitably suffers from issues of outliers, measurement noise, missing data, and irrelevant variables. While irrelevant variables become part of modeling dataset often due to imprecise knowledge of the process system at hand (for complex processes, it may not be very straightforward to manually choose the right set of model variables right away), other data impurities creep in due to failures of sensors and data acquisition system, process disturbances, etc. These impurities negatively impact the quality of our ML models and can lead to incorrect conclusions.

Decontaminating data of the aforementioned unfavorable attributes is called data cleaning and forms an important component of the data pre-processing task before model fitting. We cannot emphasize enough the importance of data cleaning. Irrespective of the how much ever advanced modeling algorithm you may deploy, you cannot escape the ‘garbage in, garbage out’ principle. ML models cannot ‘learn’ about our systems correctly when supplied with corrupted data.

Fortunately, several methods exist which can be used for automated data cleaning. We will study these methods in this chapter. Specifically, the following topics are covered

- Removing measurement noise
 - Selection of relevant variables for improved model performance
 - Outlier removal in univariate and multivariate settings
 - Common techniques for handling missing data
-

4.1 Signal De-noising

Process measurements are inevitably contaminated with high-frequency noise. If not dealt with appropriately, noise can sometimes result in errors in model parameter estimation or unfavorable characteristics in predicted variables. For example, in Figure 4.1, we can see the noisy fluctuations in a flow measurement signal. These kinds of fluctuations are generally not real process variations and are just an artifact of measurement sensors. If not removed, these may result in noisy fluctuations in predicted variables, let's say product purity, which would be very undesirable.

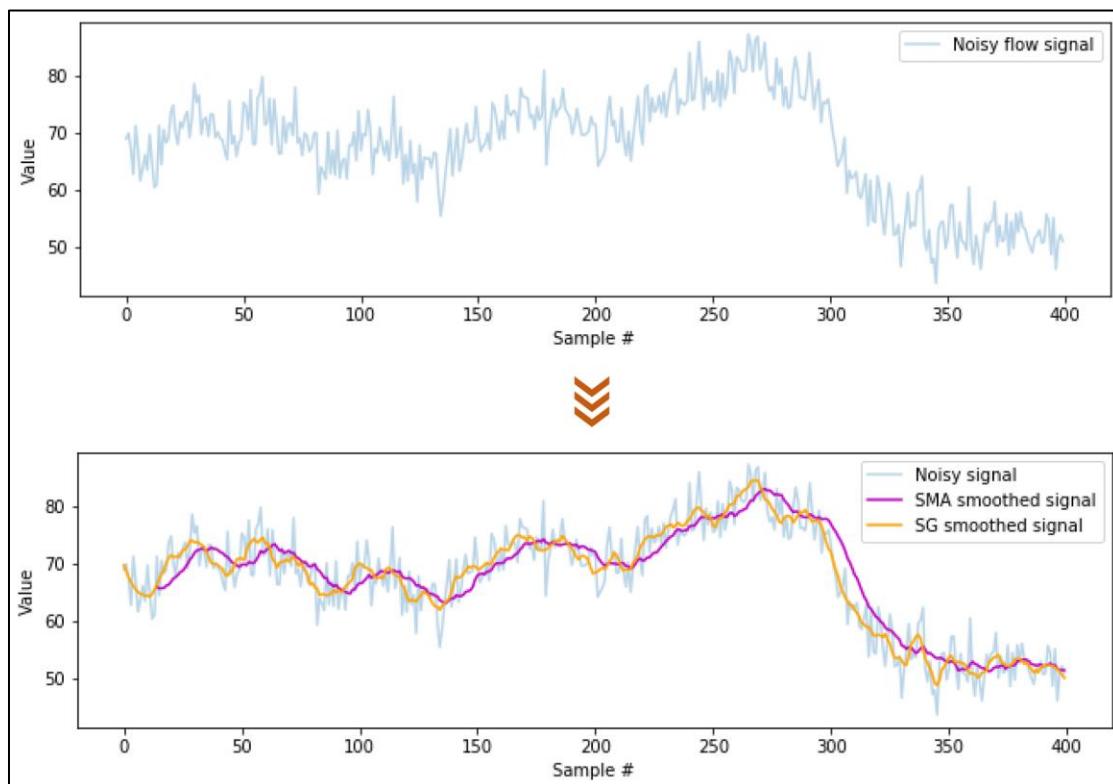


Figure 4.1: Time series signal smoothed by simple moving average (SMA) and SG filters

Two common ways to de-noise process signals is to smooth them using moving window averages and Savitzky-Golay (SG) filtering. Figure 4.1 shows the smoothed flow signals. We can see that crucial process variations have been preserved while noisy fluctuations have been removed! Let's quickly study the two techniques and learn how to implement them.

Moving window average filter

In moving window average, the smoothed value of a variable x at time t is taken as a weighted combination of previous measurements of x in a window of finite or infinite length. In the most

common variant, simple moving average (SMA), the combination is a simple average as shown below

$$x(t)_{smoothed} = \frac{\sum_{j=0}^{W-1} x(t-j)_{raw}}{W}$$

Where W is the window size. To generate the SMA smoothed signal in Figure 4.1, Pandas was utilized as follows

```
# read noisy flow data
import numpy as np
noisy_signal = np.loadtxt('noisy_flow_signal.csv', delimiter=',')

# denoise using SMA filter
import pandas as pd
windowSize = 15
smoothed_signal_MA = pd.DataFrame(noisy_signal).rolling(windowSize).mean().values # taking
mean on a rolling window and converting back to numpy array
```

Alternatives to SMA are LWMA (linearly weighted moving average) and EWMA (exponentially weighted moving average) where the past measurements are weighted in a finite and infinite length window, respectively. The value of W in finite window length variants depend on ‘how much’ smoothing is desired and is guided by the knowledge of the underlying process system (window size should ideally be smaller than the period over which systematic process variations occur) or through trial and error. As would be obvious, larger W achieves more smoothing.

SG filter

In SG filtering, the smoothed value at any point t^* is realized by approximating the values of x in an odd-sized window centered at t^* with a polynomial of time (t) and then evaluating the polynomial at t^* . Specifically, at point t^* , a m^{th} order polynomial of the following form is fitted and then evaluated as

$$x(t^*)_{smoothed} = \sum_{k=1}^m b_k (t^*)^k$$

where b_k are the estimated polynomial coefficients. In Figure 4.1, SG filter was implemented using Scipy with 2nd order polynomials as follows

```
# import savgol_filter from scipy
from scipy.signal import savgol_filter
smoothed_signal_SG = savgol_filter(noisy_signal, window_length = 15, polyorder = 2)
```

Note that, under the default setting, the $(W-1)/2$ values at the extreme edges are handled by fitting a polynomial to the W values at the edges and evaluating this polynomial to get smoothed values for the $(W-1)/2$ data-points nearest to the edges. You are encouraged to check the official documentation¹³ for other available settings.

SG filter enjoys the advantage of preserving the important features of the raw signals better than moving window average filters. SMA has another drawback which is apparent in Figure 4.1 - if we observe closely around sample 300 where a step change in flow occurs, the SMA smoothed signal shows a small time-delay/offset unlike SG smoothed signal which follows the original signal very closely during the step change. This time offset can become problematic for certain problems. SMA, however, enjoys computational advantage as it is faster to compute.



SMA and SG filters belong to the category of low-pass filters which block high frequency component of raw signals and allow low frequencies to pass through. This effectively removes spurious fast transitions, spikes, and noise while retaining the slowly evolving systematic variations. On the other hand, high-pass filters block low frequency components and allow high frequencies. These are often utilized to remove long-term slow drifts from process signals.

There are other advanced methods as well like wavelet filters, LOWESS smoothing which can be explored if SMA or SG filters don't provide satisfactory performance. Irrespective of the method deployed, you should pay due attention tuning the filters to ensure that the smoothed signal is acceptable around edges, transients, and step changes in the original signal.

4.2 Variable Selection / Feature Selection

Imagine that you are tasked with building a regression (or a classifier) model for predicting product quality in a large petrochemical plant. For such complex and highly integrated processes, it would not be surprising if you don't have enough process insights about which variables effect product quality and therefore, you end up with hundreds of variables as potential inputs to your model. The problem with inclusion of irrelevant inputs is that model training (and update) time increases, the model becomes prone to overfitting, and there is a greater requirement of training samples to adequately represent the model input space. Moreover, unnecessarily bulky input set makes physical interpretation of the model difficult.

¹³ https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.savgol_filter.html

The task of selecting the most relevant subset of inputs/feature is called variable selection / feature selection. A trivially optimal way of variable selection is to explore all possible combinations of input variables and select the one that gives the best performance on the validation dataset. However, this can be computationally infeasible when dealing with large number of input candidates. To circumvent these issues, several variable selection methods have been devised which try to find a good subset of relevant model inputs. Figure 4.2 lists some of these methods that are commonly employed. In the next few subsections, we will learn these and try to understand their strengths and drawbacks.

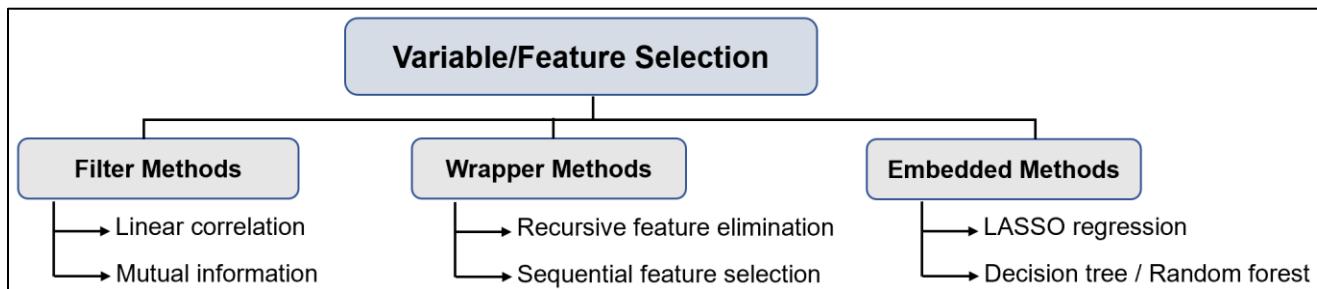


Figure 4.2: An overview of variable selection methods

As you can see, the methods are divided into three categories depending on the employed search strategy for the relevant inputs. Filter methods use statistical measures such as correlation coefficients, mutual information to quantify the relevance of any input w.r.t. the target variable. Wrapper methods utilize the model to quantify the predictive power of different input subsets via metrics like MSE, BIC, etc. In the third category, the embedded methods perform variable selection during the process of model fitting itself. Before we delve deeper into these methods, let's take a quick look at the dataset we will use in this section.

Illustration dataset

The dataset is provided in the file VSdata.csv and has been simulated using a mechanism (with slight modifications) devised by Chong & Jun¹⁴. The mechanism was designed to ensure that the dataset mimics a real manufacturing process with interconnected unit processes. There are 40 input variables out of which 10 (inputs 17 to 26) are known to be relevant while the rest are irrelevant to the target variable. Fitting and validation dataset contain 1000 and 250 samples, respectively. The target samples were generated using the relevant inputs as follows

$$y_{sample} = \sum_{j=17}^{26} \beta_j x_{sample,j} + \varepsilon_{sample}$$

¹⁴ Ching & Jun, Performance of some variable selection methods when multicollinearity is present, Chemometrics and intelligent laboratory systems, 2005

where ε_{sample} is measurement noise. All the inputs are generated from a multivariate normal distribution with zero mean and covariance matrix Γ which is designed to induce strong correlations among neighboring process inputs.

$$\Gamma_{ij} = 0.9^{|i-j|}, \quad (i,j = 1,2,\dots,40)$$

Figure 4.3 shows the target data in the training dataset and the input data. From these plots, it is not immediately clear whether an input is (more) relevant for prediction of the target or not. A multivariate linear regression with all the inputs gives a R^2 value of 0.614 on the validation dataset, while using only 10 relevant inputs results in slightly better R^2 of 0.634. Let us see if our variable selection techniques can correctly find the relevant inputs in the dataset.

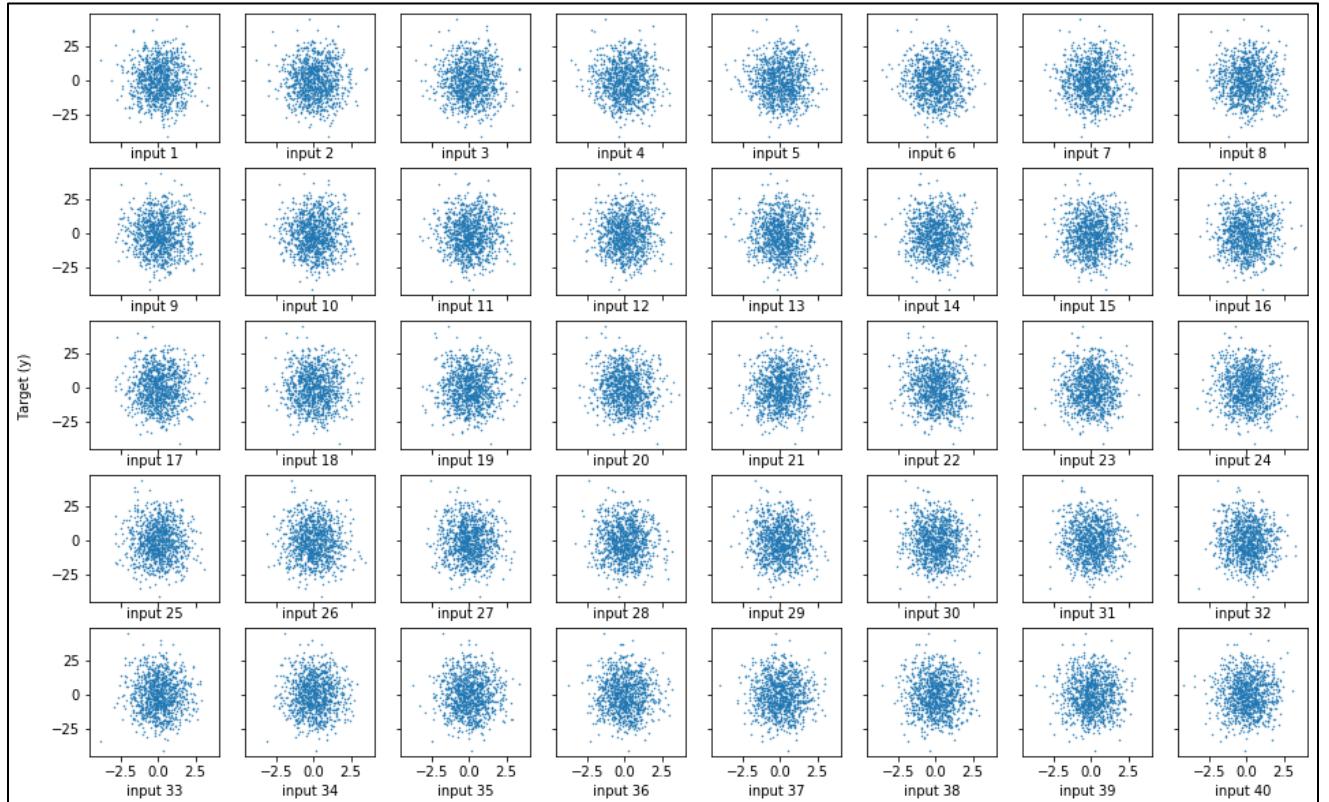


Figure 4.3: Target vs input plots for a simulated manufacturing process dataset

Filter methods

In the most common form of filter methods, the relationship between each input and the target is estimated using some statistical measure and then the inputs are ranked according to the estimated relationship strengths. Once ranked, the top ranked variables can be picked. Figure 4.4 shows an overview of the strategy.



Figure 4.4: Filter method strategy for feature selection

Let's look at the two commonly used statistical measures.

Correlation coefficients

For continuous inputs and target, the common statistical measure is Pearson correlation coefficient which measures the linear correlation between an input and the target and is given by

$$R_{xy} = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}$$

For variable selection, the top k variables with highest correlations, where k is known beforehand or decided via cross-validation, or all variables with correlations significantly different from zero can be selected.

Mutual information

Correlation coefficients can be deceptive for nonlinear dependencies between target and inputs and can lead to wrong variable selections. The solution is to use 'mutual information (MI)' metric which can efficiently quantify any kind of dependency between variables. Mathematically, MI is given by

$$MI(x, y) = \iint p(x, y) \log \frac{p(x, y)}{p(x)p(y)} dx dy$$

where $p(x, y)$ denotes the joint probability density of variables x and y and $p(x)$ denotes the marginal probability density of x . From process data, these densities can be estimated using k -nearest neighbor method (used by Sklearn), KDE, or histograms.

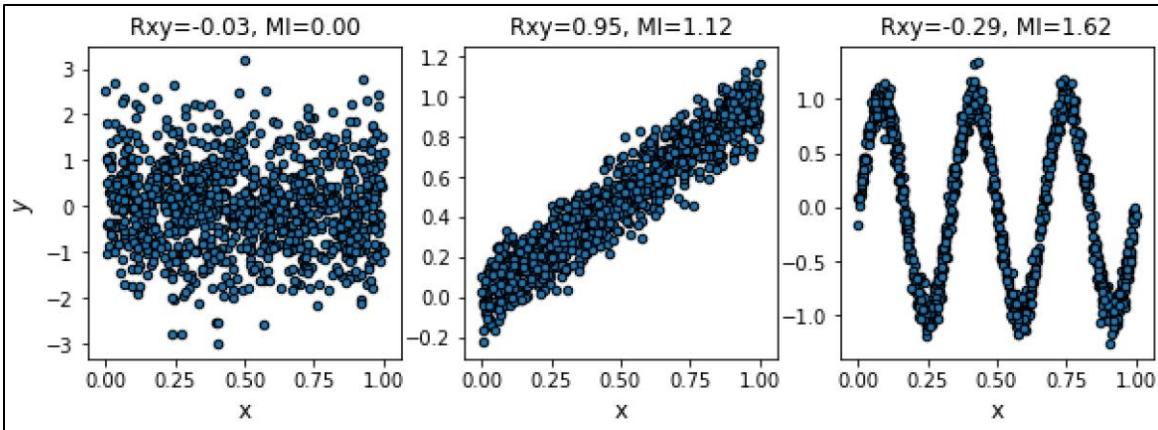


Figure 4.5: Linear correlation and mutual information metrics for zero, linear, and nonlinear dependencies between target and input

Consider the scenarios in Figure 4.3. Pearson correlation coefficient fails to identify the nonlinear (sinusoidal) dependency between target and input, but mutual information is able to determine the dependencies correctly. Colloquially speaking, MI measures the amount of information about a variable (y) that is provided by another variable (x). Therefore, if a target and an input are independent then no information about y can be obtained from knowing x and therefore, their MI would be 0. On the other hand, if y depends on x , then $MI > 0$ and higher value implies higher dependency.

As you would have noticed, filter methods do not entail building any predictive models and therefore are fast to execute. Filter methods are particularly favored when the number of input candidates are very large. However, the quality of ‘relevant’ input subset may be low. This is because filter methods consider the inputs in isolation and don’t consider the interactions among the inputs. An input that is ranked low may provide significant boost in predictive accuracy when combined with some other input. Therefore, these methods are often used as a first step of variable selection to only remove very low ranked variables and the remaining variables are further screeed via other more sophisticated methods.

Sklearn implementation

Sklearn provides `f_regression` and `mutual_info_regression` methods to compute the statistical metrics we need. `f_regression` computes linear correlation between each input variable and target which is then converted into an F score. F scores come from F-test (you may remember from your statistics classes) which does a statistical check on whether the difference in modeling errors between 2 linear models (one relating target to just a constant and the other to an input and a constant) are significant or just happened by chance. If this does not ring any bell, don’t worry. It suffices to know that the ranking obtained through F scores is the same as that obtained using correlation coefficients.

`f_regression` can be combined with `SelectKBest` transformer to conveniently transform raw input data matrix into one containing relevant inputs only as shown below

```
# read data
import numpy as np
VSdata = np.loadtxt('VSdata.csv', delimiter=',')

# separate X and y
y = VSdata[:,0]
X = VSdata[:,1:]

# compute linear correlation-based scores
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression

VSmodel = SelectKBest(f_regression, k=10).fit(X, y)
input_scores_ = VSmodel.scores_

# find the top ranked inputs
top_k_inputs = np.argsort(input_scores_)[:-1][-10] + 1
# [:-1] reverses the array returned by argsort() and [:n] gives that last n elements

# reduce X to only top relevant inputs
X_relevant = VSmodel.transform(X)
```

For our simulated dataset, inputs 22, 21, 24, 20, 30, 29, 32, 31, 28, 27 got selected as the top 10 relevant variables, i.e., only 4 inputs got correctly picked. We can see that the selection is not that good, primarily due to the multivariable dependence of our target. Let's see if wrapper methods can do any better.

Wrapper methods

Figure 4.6 gives an overview of wrapper approach to feature selection. Here, unlike filter methods, a predictive model is used to find the relevant features. The method begins with an initial selection of feature subset and enters a loop. In each cycle of the loop, a new model is built with the currently selected subset and the decision to include/exclude a feature from the subset is based on model's performance or some model attributes. The specific strategy employed distinguishes the different sequential methods. The loop concludes once the desired number of relevant features are obtained or some criteria on model performance is met. Note that the model performance may be assessed using a validation dataset or fitting dataset itself using BIC/AIC metrics. Let's study a couple of well-known wrapper methods.

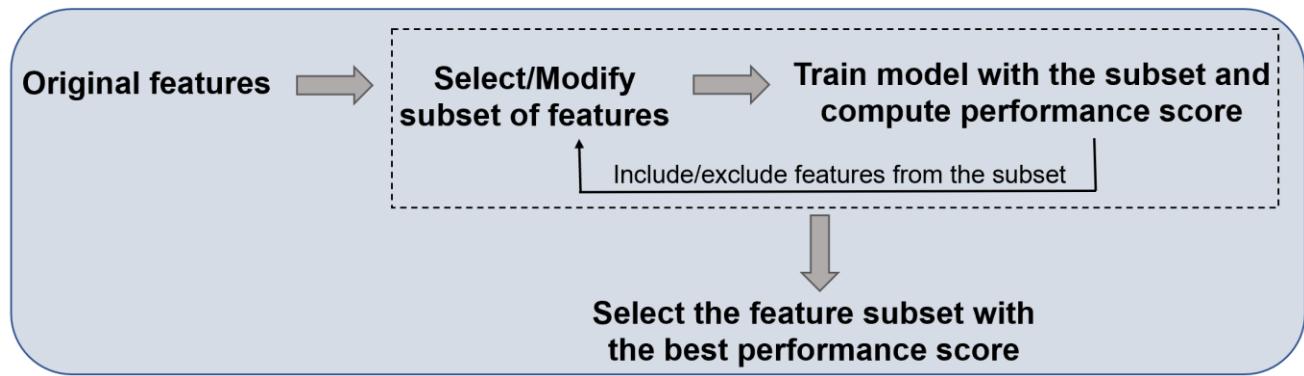


Figure 4.6: Wrapper method strategy for feature selection

Recursive feature elimination (RFE)

In this method, a model is initially built using all the input variables. The importance of each input is quantified using any specific model attribute (such as model coefficients) or through some combination of model attributes. A few of the inputs with the least importances are discarded and the entire process is repeated until the desired number of relevant inputs are obtained or model performance begins to deteriorate.

PLS is a very popular regression method for process systems which we will study in detail in Chapter 5. We are mentioning this here because PLS allows for computation of VIP (variable importance in projection) scores that quantify the importance of each input. PLS VIP¹⁵-based RFE algorithm can be used to iteratively exclude inputs that contribute the least to the prediction of the target variable.

Sequential feature selection (SFS)

Sequential feature selection technique has two common variants. In forward SFS, the search begins with an empty subset. The input that leads to the best performance when the model is trained on this single input gets added to the subset. Next, the previously selected subset is combined with each of the remaining inputs one at a time. The combination of two inputs that gives the best performance becomes the selected subset. This procedure is repeated till k inputs are selected or predictions no longer improve by adding new variables.

The other variant, backward SFS, works similarly as forward SFS, except that it starts with all the inputs. The input whose exclusion leads to the best model performance is excluded from the subset. This procedure is repeated to exclude inputs one by one.

¹⁵ PLSRegression class in sklearn does not have any built-in VIP computation. pyChemometrics package can be used which provides a ChemometricsPLS class with a VIP method

Since wrapper methods use models, they outperform filter methods. However, because they execute multiple model fittings, they can become computationally infeasible for very large number of inputs and complex models (such as ANNs). Between forward and backward SFS, backward SFS takes into account the interactions effects of all the variables better than the forward SFS but can entail higher number of model fittings when k is much lower than the number of input candidates.

Sklearn implementation

Sklearn provides RFE and SequentialFeatureSelector classes that transform raw input data matrix into relevant subset using any specified estimator/model. Below is an implementation for backward SFS using linear regression model.

```
# scale data
from sklearn.preprocessing import StandardScaler
xscaler = StandardScaler()
X_scaled = xscaler.fit_transform(X)

yscaler = StandardScaler()
y_scaled = yscaler.fit_transform(y[:,None])

# implement backward SFS
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.linear_model import LinearRegression

BSFS = SequentialFeatureSelector(LinearRegression(), n_features_to_select=10, direction='backward',
cv=5).fit(X_scaled, y_scaled)

# check selected inputs
print('Inputs selected: ', BSFS.get_support(indices=True)+1)

>>> Inputs selected: [18 19 20 21 22 23 24 25 31 33]

# reduce X to only top relevant inputs
X_relevant = BSFS.transform(X)
```

We can see that wrapper method has performed much better than filter method and has recovered 8 out of 10 relevant inputs! Another impressive thing to highlight here is how SequentialFeatureSelector allows condensing all the complexities of SFS variable selection with cross-validation into just a couple of lines of code!!

Embedded methods

Embedded methods make use of algorithms where selection of variables is an inherent part of the model fitting process itself. We already saw one of the methods in the previous chapter, Lasso regression, where irrelevant inputs are eventually removed from model by assigning them zero coefficients. Decision trees and random forests are another set of algorithms which directly provide feature importances after model fitting. We will study them in Chapter 9.

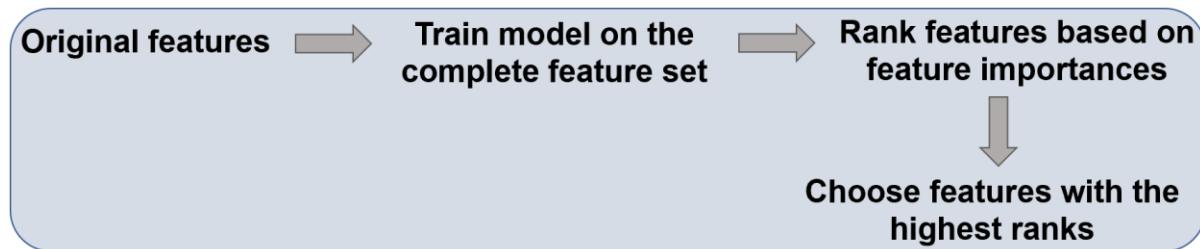


Figure 4.7: Embedded method strategy for feature selection

Embedded methods are computationally less expensive than wrapper methods and work best when the number of samples are much higher than the number of inputs. Below is a quick implementation of Lasso regression for our simulated dataset where we use the model coefficients to judge any input's importance.

Sklearn implementation

We will employ LassoCV model which automatically selects a penalization strength using cross-validation. As shown, Lasso has recovered 8 out of 10 relevant inputs.

```
# fit Lasso model
from sklearn.linear_model import LassoCV
Lasso_model = LassoCV(cv=5).fit(X_scaled, y_scaled)

# find the relevant inputs using model coefficients
top_k_inputs = np.argsort(abs(Lasso_model.coef_))[:-1][:-10] + 1
print('Relevant inputs: ', top_k_inputs)

>>> Relevant inputs: [21 22 20 23 24 19 25 18 33 14]
```

This concludes our look into the variable selection methods. The presence of nonlinearity, input interactions, computational resource availability, and the number of input candidates influence the eventual method selection. You are now familiar with the mechanism of these methods which should help you choose the method most suitable for your problem.



Feature extraction and feature selection are similar in the sense that both achieve feature dimension reduction. Feature extraction creates new features by using all the original features and therefore does not help in simplification of original measurement space. The focus is more on dimensionality reduction and removal of correlation among the features. Feature selection, as we have seen, discards irrelevant features altogether. Here the focus is on improved model performance with simpler physical interpretation due to low number of model inputs.

4.3 Outlier Handling

Outliers are abnormal observations that are inconsistent with the majority of the data. Presence of outliers in the training dataset results in a model that does not adequately represent the true process and therefore negatively impacts its prediction accuracy. It is imperative to remove outliers before model fitting or use a modeling algorithm that is robust to outliers.

Several methods are at our disposal - the choice depends on whether we are dealing with univariate or multivariate system, the degree of outlier contamination, Gaussian or non-Gaussian data distribution, etc. Figure 4.8 lists some of these methods that are commonly employed.

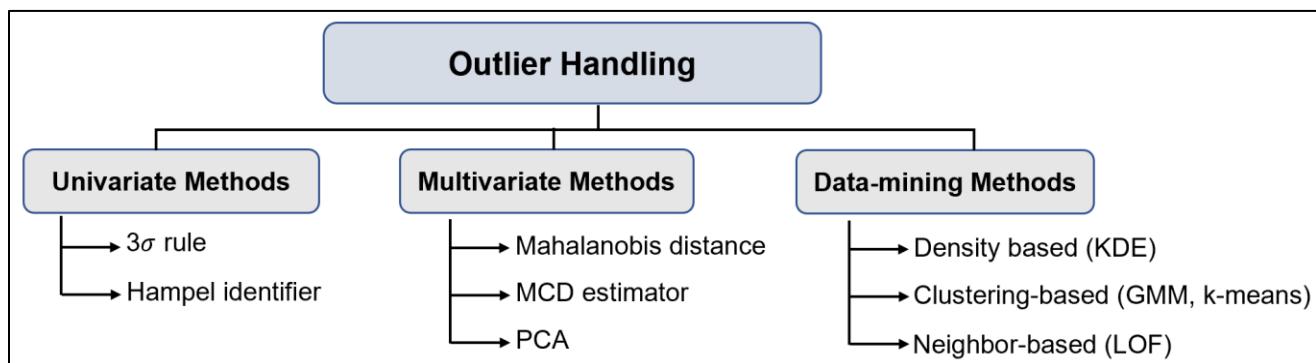


Figure 4.8: An overview of variable selection methods

Univariate methods

Univariate methods clean each variable separately. The 3-sigma rule and Hampel identifier are the popular methods available under this category.

3σ rule

The 3σ rule works as follows: if μ and σ denote the mean and standard deviation of a variable, then any sample beyond the range $\mu \pm 3\sigma$ are considered as outliers. This rule follows from the properties of a Gaussian-distributed variable for which 99.87% of data lie within the $\mu \pm 3\sigma$ range. However, by using a factor other than 3, this rule is often used for non-Gaussian variables as well.

Hampel identifier

You will agree that the 3σ rule is quite simple and easy to implement. Expectedly, it has some limitations. If a variable is severely contaminated with outliers, then μ and σ can get impacted to such an extent that abnormal samples end up falling within the $\mu \pm 3\sigma$ range and fail to be identified as outliers. The solution is to use Hampel identifier which replaces μ and σ with median and MAD, respectively. For any variable x , Hampel identifier tags an observation as an outlier if it lies outside the range $\text{median}(x) \pm 3\sigma_{\text{MAD}}$ where $\sigma_{\text{MAD}} = 1.4826 * \text{median}(|x - \text{median}(x)|)$.

We worked with median and MAD in the previous chapter for robust scaling. Using the same dataset, Figure 4.9 now illustrates the effectiveness of Hampel identifier for univariate outlier detection. The ‘normal range’ computed using 3-sigma rule is inflated enough to include all the data within the normal range! Hampel identifier, however, provides a normal range which is more representative of the normal data and therefore can flag the abnormal samples as outliers. We can see that Hampel identifier provides quite a superior performance with very minimal additional complexity.

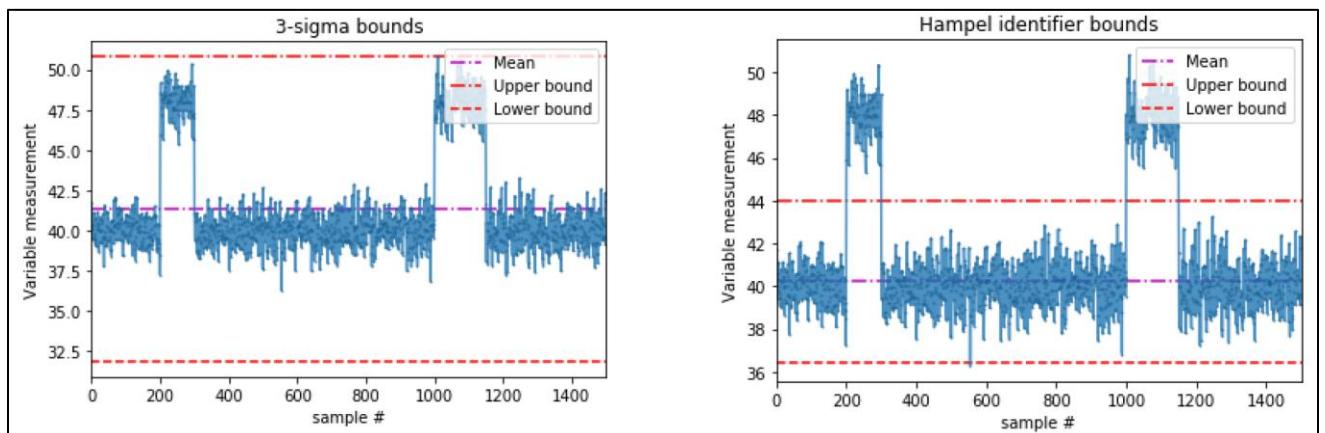


Figure 4.9: Univariate bounds obtained with 3-sigma and Hampel identifier rules for data contaminated with outliers

Multivariate methods

In process dataset where variables are highly correlated, outliers can escape detection when variables are ‘cleaned’ separately. For example, consider Figure 4.10. The red samples are clearly outliers in the 2D space. However, if we look only along the x_1 (or x_2) axis alone, these abnormal samples lie close to the center of the data distribution, and therefore will not be considered outliers in univariate sense.

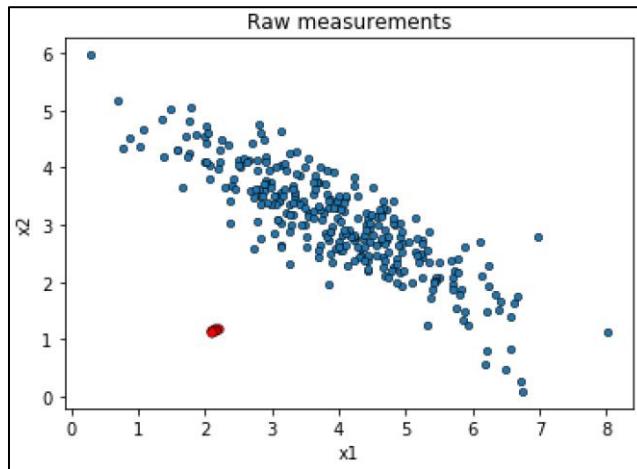


Figure 4.10: Outliers in 2D space difficult to detect via univariate methods

The solution is to use multivariate methods of outlier detection. These methods take into consideration the shape of multivariate data distribution and are often based on using some distance metrics to flag samples that are far away from the center of data distribution. Let’s study these methods in detail.

Mahalanobis distance

Mahalanobis distance (MD) is a classical multivariate distance measure that takes into account the covariance/shape of data distribution for computing distances from the center of data. Specifically, MD of any observation \mathbf{x}_n is given by

$$MD(\mathbf{x}_n) = \sqrt{(\mathbf{x}_n - \bar{\mathbf{x}})\Sigma^{-1}(\mathbf{x}_n - \bar{\mathbf{x}})^T}$$

where $\bar{\mathbf{x}}$ and Σ are mean and covariances of the sampled observations. The presence of Σ differentiates MD from Euclidean distance. Computation of MD essentially converts a multivariate outlier detection problem into a univariate problem. For illustration, Figure 4.11 shows the MDs for the dataset from Figure 4.10. We can see that the outlier samples are clearly very distinct from the normal samples and can be easily flagged via univariate techniques.

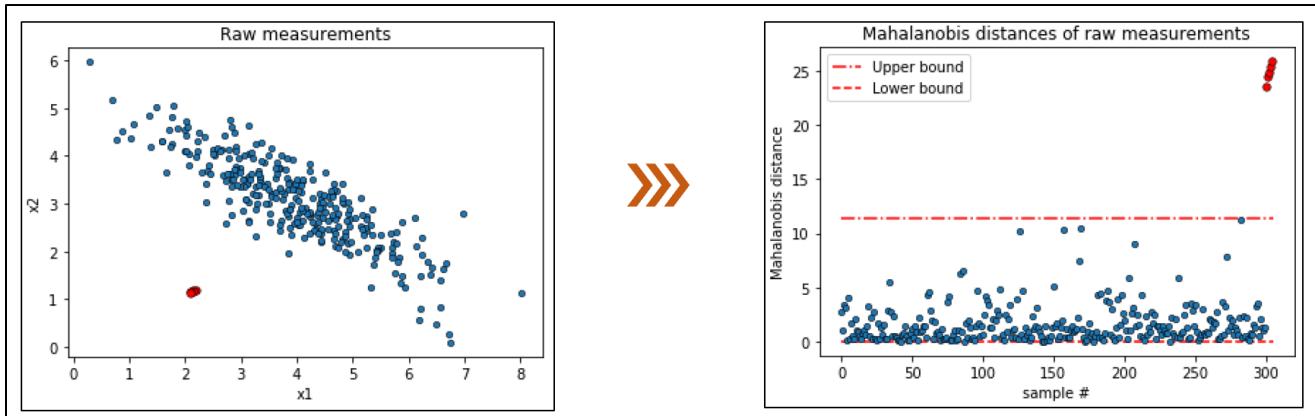


Figure 4.11: Multivariate outlier detection via Mahalanobis distance and Hampel identifier

In this illustration, the shown bounds were obtained via Hampel identifier on the cubic roots of MDs (cubic root taken to make the MDs distribution approximately Gaussian¹⁶) as shown in the code below

```
# read data
import numpy as np
data_2Doutlier = np.loadtxt('simple2D_outlier.csv', delimiter=',')

# compute Mahalanobis distances and transform into Gaussian distribution using cubic-root
from sklearn.covariance import EmpiricalCovariance
emp_cov = EmpiricalCovariance().fit(data_2Doutlier)
MD_emp_cov = emp_cov.mahalanobis(data_2Doutlier)
MD_cubeRoot = np.power(MD_emp_cov, 0.333)

# find Hampel identifier bounds
from scipy import stats
median = np.median(MD_cubeRoot)
sigma_MAD = stats.median_absolute_deviation(MD_cubeRoot)

upperBound_MD = np.power(median+3*sigma_MAD, 3)
lowerBound_MD = np.power(median-3*sigma_MAD, 3)

# plot Mahalanobis distances with bounds (last 5 samples are the outliers)
plt.figure(), plt.plot(MD_emp_cov[:-5], '.', markeredgecolor='k', markeredgewidth=0.5, ms=9)
plt.plot(np.arange(300,305), MD_emp_cov[-5:], '.r', markeredgecolor='k', markeredgewidth=0.5, ms=11)
plt.hlines(upperBound_MD, 0, 305, colors='r', linestyles='dashdot', label='Upper bound')
plt.hlines(lowerBound_MD, 0, 305, colors='r', linestyles='dashed', label='Lower bound')
```

¹⁶ Wilson & Hilferty, The distribution of chi-square, Proceedings of the National Academy of Sciences of the United States of America, 1931

MCD estimator

Consider the data distribution and its Mahalanobis distances in Figure 4.12. Here again a bunch of samples lie away from the normal data, but as shown, the MD-based method fails miserably in detecting all the outliers. So, what happened here?

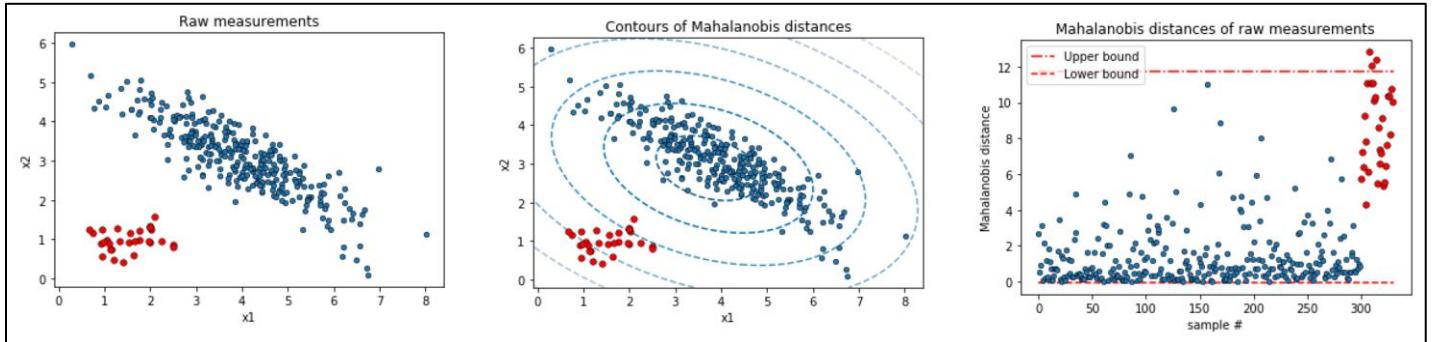


Figure 4.12: Outliers in 2D space difficult to detect via classical Mahalanobis distance method

What's happening here is that the outliers are distorting the estimated shape/covariance structure of the data distribution. As shown in the middle subplot, the contours of MDs have been inflated to such an extent that MDs of several outliers are similar to those of inliers. This unwanted effect is called masking effect. Moreover, if inliers get classified as outliers due to the impact of outliers, it is called swamping effect.

Previously, in univariate methods, we saw that a robust outlier detection solution can be obtained by utilizing robust estimates of data location and spread. Similar concept exists for multivariate methods. A popular estimator of location (centroid) and scatter (covariance matrix) is MCD (Minimum Covariance Determinant). MCD finds h samples out of the whole dataset for which the covariance matrix has lowest determinant, or, simply speaking, the enveloping ellipsoid for a given confidence level has lowest volume among all possible subsets of size h . The centroid and covariance of h samples are returned as robust estimates.

The `MinCovDet` class in Sklearn implements a FAST-MCD algorithm which can efficiently analyze large datasets with high dimensionalities. The superior performance of this method is validated in Figure 4.13. The MD contours now fit the inlying samples much better and the outliers are well beyond the computed upper bound. Note that Sklearn provides a `EllipticEnvelop` class that uses FAST-MCD to directly detect an outlier but requires specification of data contamination level which may not be known beforehand. The code below shows the computation of MCD-based MDs; bound are computed as before.

```
# read data
import numpy as np
```

```

data_2Doutlier = np.loadtxt('complex2D_outlier.csv', delimiter=',')
# compute MCD-based Mahalanobis distances
from sklearn.covariance import MinCovDet
MCD_cov = MinCovDet().fit(data_2Doutlier)
MD_MCD = MCD_cov.mahalanobis(data_2Doutlier)

```

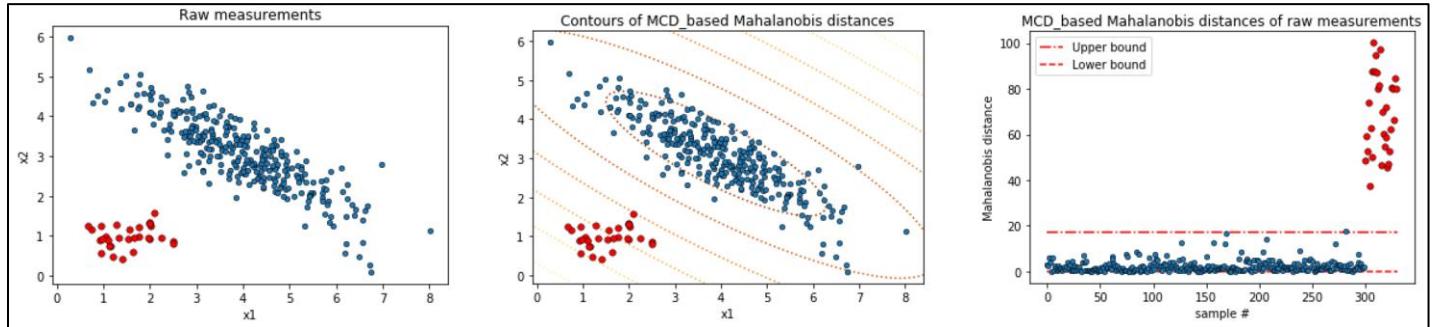


Figure 4.13: Multivariate outlier detection via MCD-based robust Mahalanobis distances

PCA

Not all outliers are bad and warrant removal. For example, consider Figure 4.14. Here, two kinds of outliers are highlighted. While samples *A* tend to break away from the majority correlation, samples *B* follow the correlation but lie far away from the majority of data.

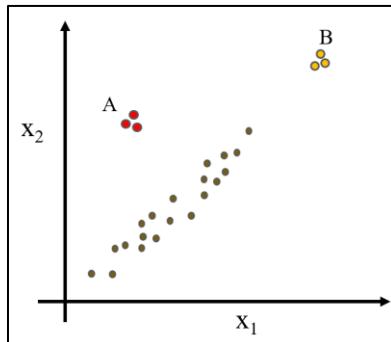


Figure 4.14: Different types of outliers which can be distinguished via PCA

Depending on the context of our problem, we may or may not want to remove samples *B*. The MD-based approach won't distinguish between these two kinds of outliers and therefore, won't work for us if we want to retain samples *B*. In that case, PCA (principal component analysis) can be employed. We will learn PCA in detail in Chapter 5. There you will understand how PCA distinguishes between these two different kinds of outlying observations. Classical PCA is based on sample covariance of data. Therefore, if the given dataset is highly contaminated with outliers, MCD-based covariance can be used in PCA for robust analysis.

Data-mining methods

The multivariate methods we studied in previous section make an implicit assumption about Gaussianity or unimodality of data distribution. In real life, it is not rare to encounter situations where these assumptions do not hold. One such data distribution is shown in Figure 4.15, where 3 distinct clusters of data is present with outliers in between them. MD or PCA-based approach will fail here!

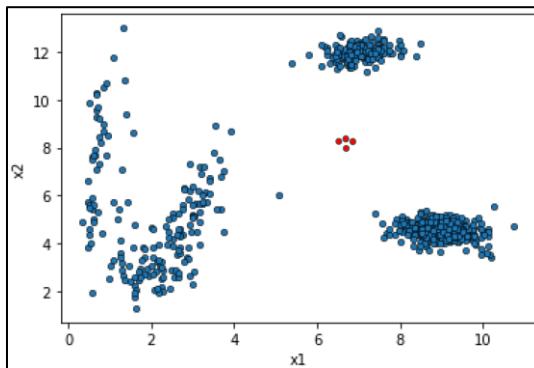


Figure 4.15: Complex multivariate data distribution with outliers

Advanced data-mining methods are employed to handle these complicated situations. The methods can be categorized into density-based, clustering-based, and neighbor-based. Density-based methods (like KDE) rely on estimation of distribution density of the data; observations lying in low-density regions are flagged as outliers. Cluster-based methods (like GMM, Kmeans) find distinct clusters of data; samples belonging to small-sized or low-importance clusters are flagged as outliers. Neighbor-based methods (like KNN, LOF) rely on finding the neighbors of each data-point. Neighboring data-points are used to compute any sample's distance from its k-nearest neighbor or the local density; samples with large distances or low densities are flagged as outliers.

We will study most of these techniques in Part 2 of the book. Once you study their nuances, you will realize that these are very powerful and useful methods but do require careful selection of their intrinsic model parameters such as the number of clusters in Kmeans, number of neighbors in LOF, or the kernel function & its width in KDE.



Time-series or dynamic dataset require different approaches for outlier detection. A common practice is to first model the given dataset using an appropriate time-series modeling technique such as ARIMA or ARX and compute residuals between model prediction and given data. With properly chosen model, residuals will be approximately normally distributed and uncorrelated. Then, any traditional outlier detection method can be applied on the residuals. The outliers will tend to have large residuals and will get flagged.

Outlier-resistant robust techniques

The outlier-handling method we studied till now relied on explicit identification of outliers. Once identified, these outliers are removed from model fitting dataset before model parameter estimation. There is another set of methods, called robust methods^{*+}, which directly use outlier-contaminated data for model fitting. These algorithms are designed such that the effect of outliers is reduced by down-weighting them and only inliers primarily determine the model parameters.

Figure 4.16 illustrates an application of robust linear regression using RANSAC algorithm in Sklearn. The robust method is able to successfully ignore the outliers to find the correct fit between target and predictor.

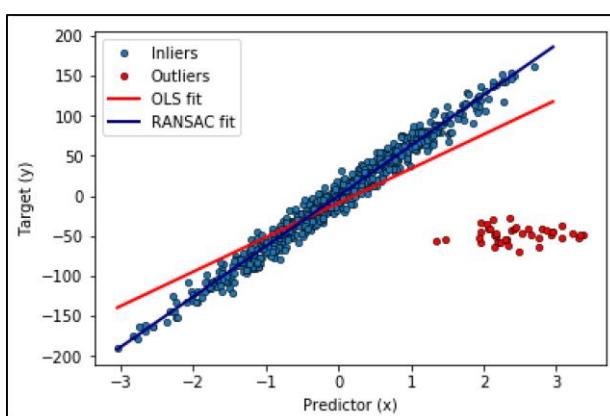


Figure 4.16: Robust model fitting with outlier-infested data

IRPLS (iteratively re-weighted PLS) is another popular robust regression method using PLS. Most of the outlier-resistant robust methods employ iterative algorithm where model parameters are computed iteratively and samples with high residuals identified in each iteration are assigned lower weights.

*Zhu et. al., Review and big data perspectives on robust data mining approaches for industrial process modeling with outliers and missing data, Annual Reviews in Control, 2018

+ Frosch + Bro, Robust methods for multivariate data analysis, Journal of Chemometrics, 2005

This concludes our study of methods for outlier detection. Our objective was to help you understand the pros and cons of different techniques because there is no universal method applicable to all problems. A good understanding of the underlying process and the nature of data can greatly help make the right choice of technique.

4.4 Handling Missing Data

Missing data issues arise when values of a variable or multiple variables are missing in one or more observations of the dataset. The samples with missing data can simply be discarded if their number is much smaller than the total number of available samples. However, if this is not the case or there are reasons to believe that discarding samples will negatively impact model fitting, imputation techniques can be employed which attempt to fill-in the missing values of a variable using available data from the same variable or other variables.

A simple imputation strategy is mean imputation where the missing values of a variable are replaced with the mean of all available data for the variable and is illustrated in the code snippet below

```
# Mean imputation
import numpy as np
from sklearn.impute import SimpleImputer

sample_data = [[1, 2, np.nan], [3, 4, 3], [np.nan, 6, 5], [8, 8, 7]]
mean_imputeModel = SimpleImputer(missing_values=np.nan, strategy='mean')

print(mean_imputeModel.fit_transform(sample_data))

>>> [[1. 2. 5.]
       [3. 4. 3.]
       [4. 6. 5.]
       [8. 8. 7.]]
```

Another widely used approach is hot-deck imputation, where missing values of a sample are replaced from the available values of ‘similar’ samples. Sklearn provides KNNImputer class which imputes missing value with the weighted (or mean) combination from k-nearest neighbors in the fitting dataset. The distance metric between 2 samples is computed using features that neither samples are missing.

```
# KNN imputation
from sklearn.impute import KNNImputer

knn_imputeModel = KNNImputer(n_neighbors=2)
print(knn_imputeModel.fit_transform(sample_data))
```

```
>>> [[1, 2, 4],  
     [3, 4, 3],  
     [5.5, 6, 5],  
     [8, 8, 7]]
```

Although not currently available in Sklearn, more advanced techniques like regression imputation, likelihood-based imputation also exist. The former method imputes the missing values in a variable using a regression model with other correlated variables as predictors. The later method uses expectation-maximization (EM) algorithm for imputing missing values in a probabilistic framework.

As with variable selection and outlier handling methods, it cannot be said beforehand which missing data technique will work best for your problem. A good recourse is to select a few methods based on your process knowledge, build separate models with them, and select the best performing one!

Summary

In this chapter, we studied the techniques commonly employed to handle different types and degree of data contamination. We looked into the issues of measurement noise, irrelevant features, outliers, and missing data. A natural pre-processing workflow entails denoising the dataset first, followed by variable selection and outlier removal. With this, we conclude the first leg of our ML journey. With the fundamentals in place, let's start the second phase of the journey where we will look into several classical and very useful ML algorithms.

Part 2

Classical Machine Learning Methods

Chapter 5

Dimension Reduction and Latent Variable Methods (Part 1)

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

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

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

5.1 PCA: An Introduction

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

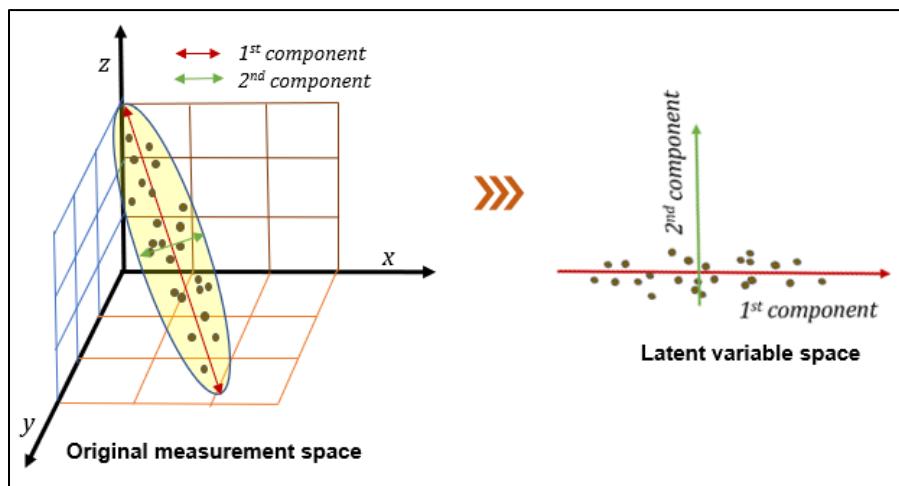


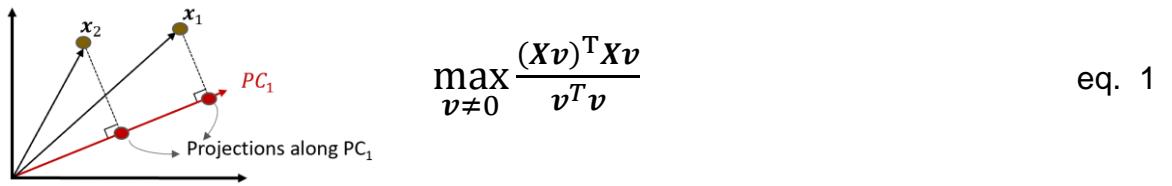
Figure 5.1: PCA illustration

In ML world, it is common to find applications of classification and clustering techniques in the PC space. In process industry, process modeling (via principal component regression (PCR)) and monitoring are common application of PCA¹⁷. In PCR, the process outputs are regressed onto the principal components of the input data. By doing so, the problem of ill-conditioning frequently encountered in classical multiple linear regression (MLR) due to high input space dimensionality and high degree of correlation among input variables is circumvented. PCA is also frequently utilized for process visualization. For many applications, two or three PCs are adequate for capturing most of the variability in process data and therefore, the compressed process data can be visualized within a single plot. Plant operators and engineers use this single plot to find past and current patterns in process data.

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

Mathematical background

Consider a data matrix $\mathbf{X} \in \mathbb{R}^{N \times m}$ 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 $\mathbf{v} \in \mathbb{R}^m$ 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 (\mathbf{S}) of \mathbf{X}

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

Above is the form you will find commonly in PCA literature. The columns of eigenvector matrix $\mathbf{V} \in \mathbb{R}^{m \times m}$ 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 \mathbf{V} (which corresponds to the first $k \lambda_s$) are taken to form the loading matrix $\mathbf{P} \in \mathbb{R}^{m \times k}$. Transformed data in the PC space can now be obtained

$$\xleftarrow[\text{Projected values along } j^{\text{th}} \text{ PC}]{\quad} \mathbf{t}_j = \mathbf{X}\mathbf{p}_j \text{ or } \mathbf{T} = \mathbf{X}\mathbf{P}$$
eq. 3

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

$$\widehat{\mathbf{X}} = \mathbf{T}\mathbf{P}^T$$
eq. 4

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

Dimensionality reduction for Polymer Manufacturing Process

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

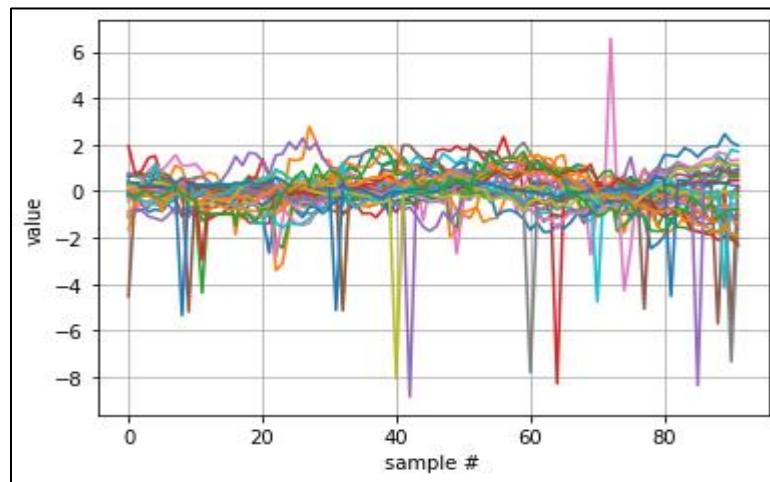


Figure 5.2: Process data from a polymer manufacturing plant. Each colored curve corresponds to a process variable

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

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

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

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

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

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

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

Correlation matrix:

```
[[ 1.0000000e+00  8.24652750e-16 -1.88830953e-16]
 [ 8.24652750e-16  1.0000000e+00  2.36966153e-16]
 [-1.88830953e-16  2.36966153e-16  1.0000000e+00]]
```

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

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

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

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

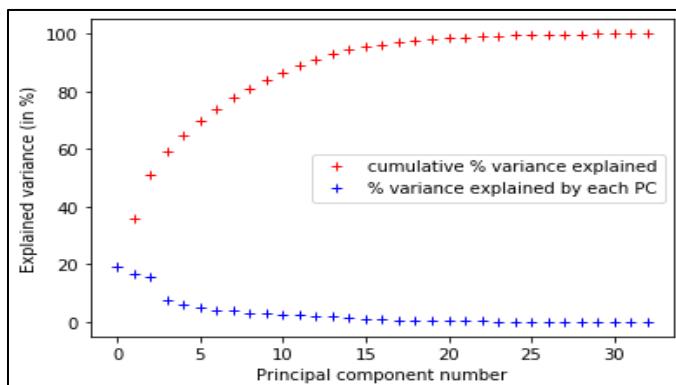


Figure 5.3: Variance explained by principal components

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

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

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

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

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

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

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

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

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

% information lost = 9.0469
```

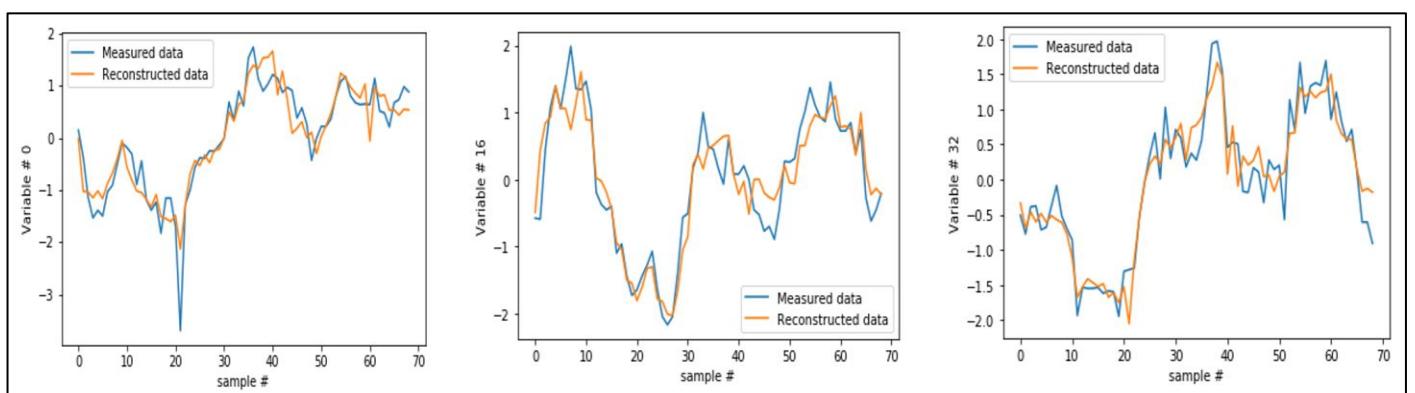


Figure 5.4: Comparison of measured and reconstructed values for a few variables

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

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

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

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

% information lost = 9.0469
```

5.2 Process Monitoring via PCA for Polymer Manufacturing Process

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

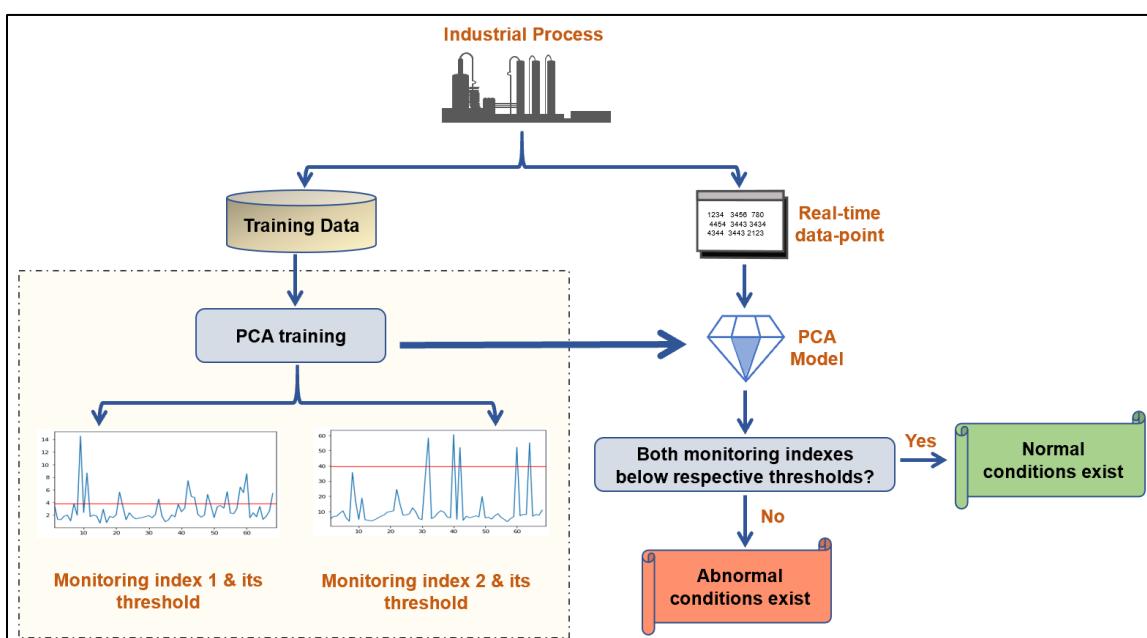


Figure 5.5: PCA-based process monitoring workflow

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

Process monitoring/fault detection indices

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

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

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

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

$F_{k,N-k}(\alpha)$ 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. The second index, Q, represents the distance between the original and reconstructed data-point. Let \mathbf{e}_i denote the i^{th} row of \mathbf{E} . Then

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

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

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

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

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

```

# calculate T2 for training data
lambda_k = np.diag(pca.explained_variance_[0:n_comp]) # eigenvalue = explained variance
lambda_k_inv = np.linalg.inv(lambda_k)

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

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

# T2 control limit
import scipy.stats

N = data_train_normal.shape[0]
k = n_comp

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

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

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

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

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

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

```

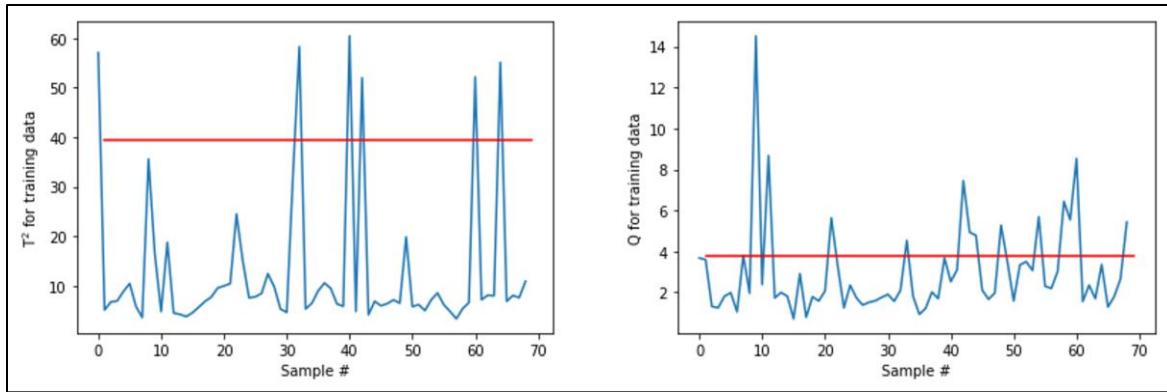


Figure 5.6: Monitoring charts for training data

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

Importance of both T^2 and Q statistic

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

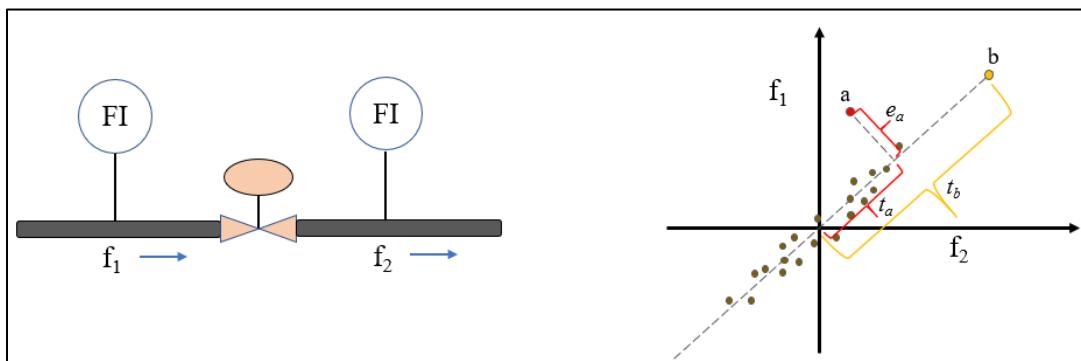


Figure 5.7: (Left) Flow measurements across a valve (Right) Mean-centered flow readings with two abnormal instances (samples *a* and *b*)

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

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

Fault detection

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

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

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

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

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

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

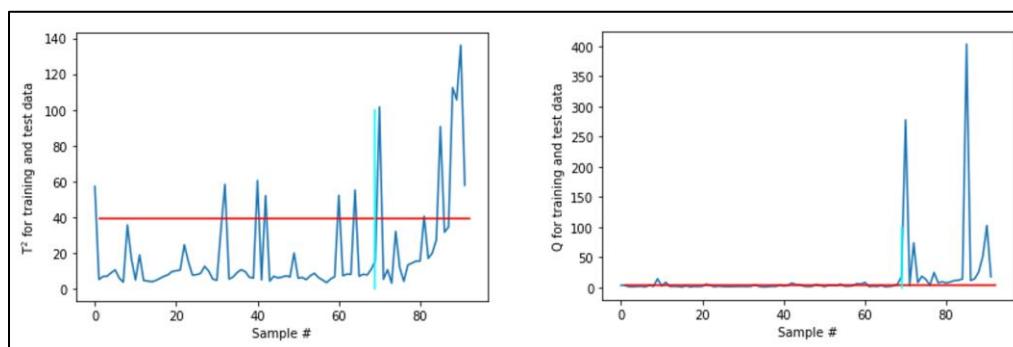


Figure 5.8: Monitoring charts for training and test data. Vertical cyan-colored line separates training and test data

Fault diagnosis

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

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

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

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. For T^2 contributions, calculations are not as straight-forward. Several expressions have been postulated in literature¹⁸. The commonly used expression below was proposed by wise et al.¹⁹.

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

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

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

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

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

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

¹⁸ S. Joe Qin, Statistical process monitoring: basics and beyond, Journal of Chemometrics, 2003

¹⁹ Wise et. al., PLS toolbox user manual, 2006

```

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

```

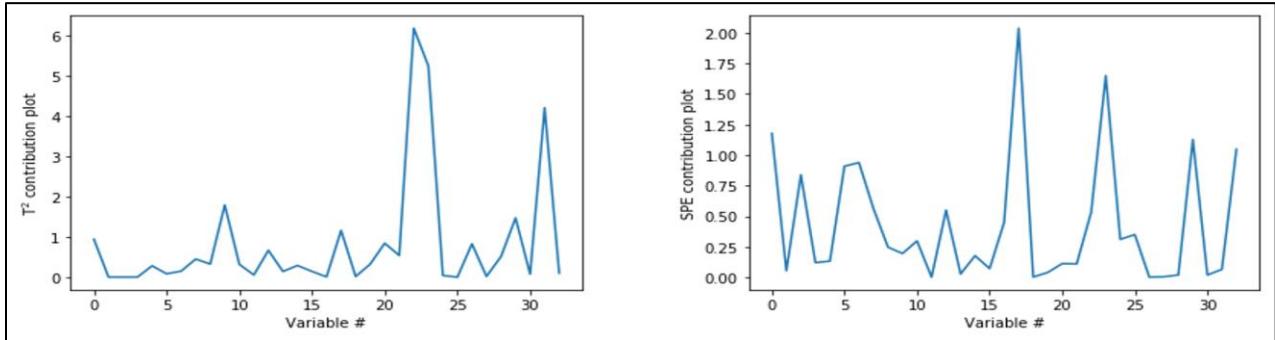


Figure 5.9: T^2 and SPE contribution plots for sample 85

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

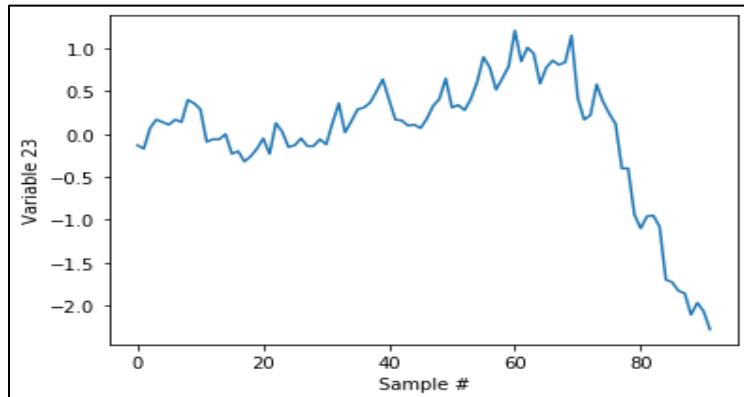


Figure 5.10: Temporal evolution of variable # 23

5.3 Variants of Classical PCA

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

Dynamic PCA (DPCA)

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

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

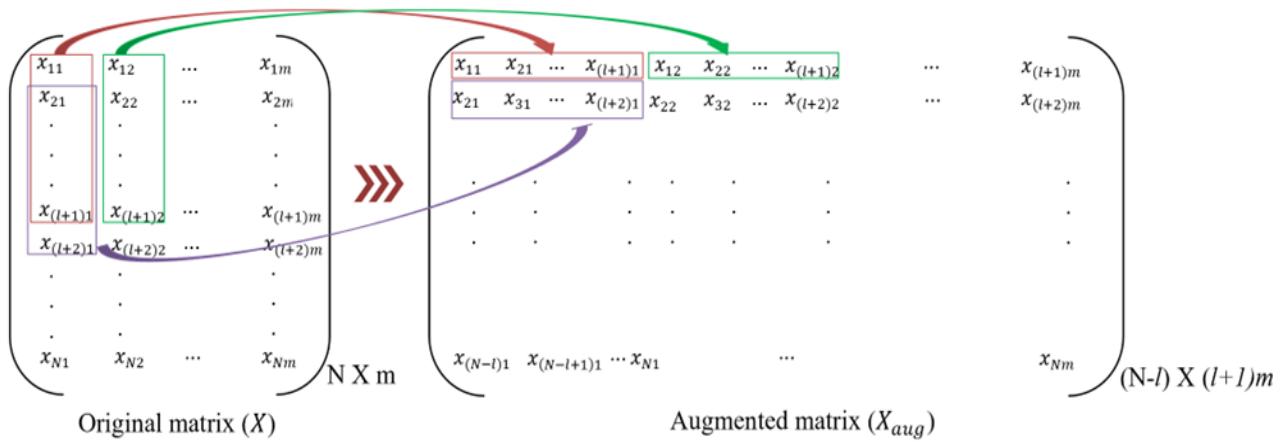


Illustration: Augmentation of lagged/past values to generate augmented data matrix

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

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

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

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

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

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

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

Multiway PCA

PCA has been found useful for monitoring batch processes as well where raw data have an additional layer of variability, the inter-batch variability. To capture the variability in batch data, the three-dimensional data is unfolded to form a two-dimensional data matrix as shown in Figure 5.11. Static PCA is applied on this unfolded data matrix after normalizing each column²¹. Once a test batch data becomes available, test scores are obtained for further analysis.

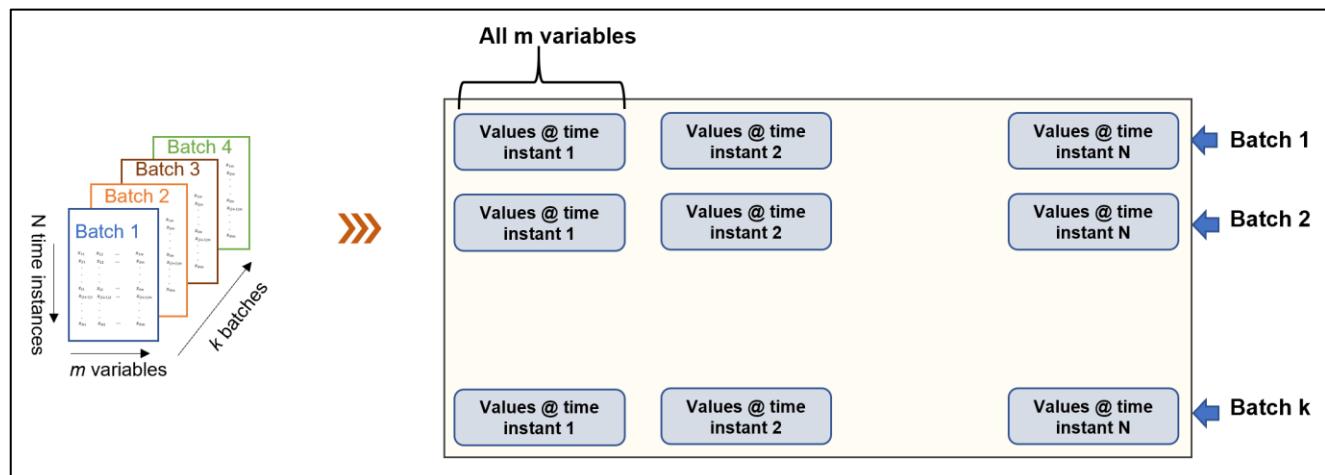


Figure 5.11: Unfolding batch data for multiway PCA

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

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

Kernel PCA

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

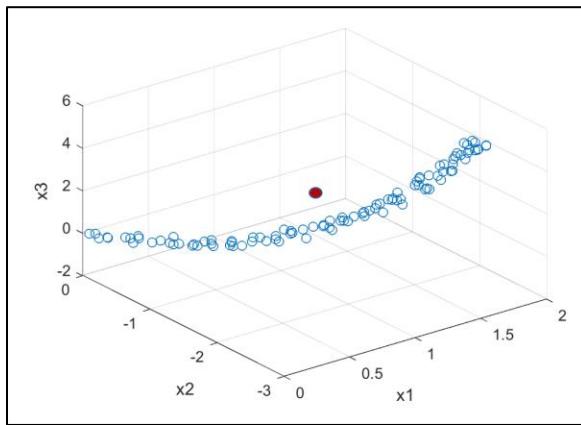


Figure 5.12: Nonlinearly related data with an abnormal (red) data-point

In such scenarios, the frequently employed solution is to transform the data into a higher-dimensional space where different classes (normal and abnormal classes) of data become linearly separable. To understand this trick, consider Figure 5.13, where the two classes of data are not linearly separable in the original measurement space. However, when an artificial variable ($x_1^2 + x_2^2$) is added to the system, the classes become easy to separate. This high-dimensional space is called the feature space.

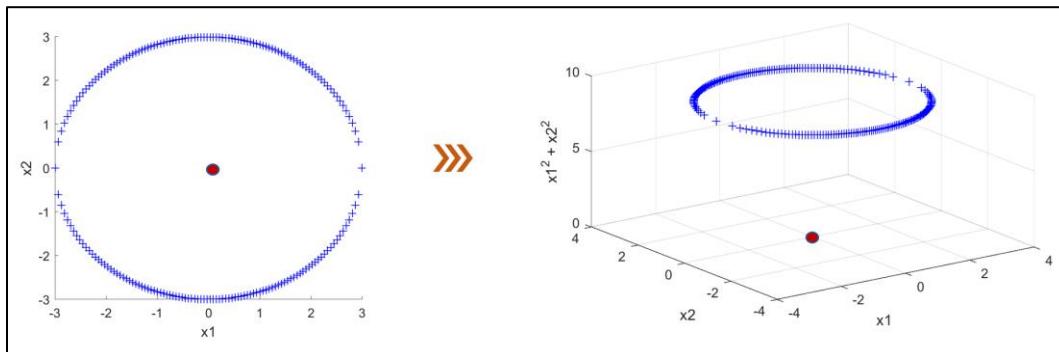


Figure 5.13: Mapping to feature space to enable linear separation

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

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

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

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

5.4 PLS: An Introduction

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

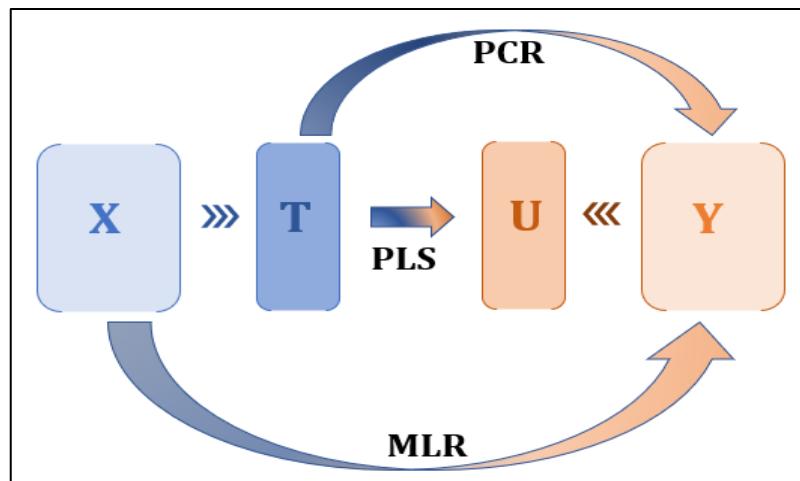


Figure 5.14: PLS, PCR, MLR methodology overview. Note that the score matrix, \mathbf{T} , for PLS and PCR can be different.

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

While MLR computes the least-squares fit between \mathbf{X} and \mathbf{Y} directly, PCR first performs PCA on input data and then computes least-squares fit between the score matrix and \mathbf{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 \mathbf{X} which are not relevant for predicting \mathbf{Y} . PLS overcomes this issue by estimating the score matrices, \mathbf{T} and \mathbf{U} , simultaneously such that the variation in \mathbf{X} that is relevant for predicting \mathbf{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 \mathbf{X}
- Capture maximum variability in \mathbf{Y}
- Maximize correlation between \mathbf{X} and \mathbf{Y}

To see how PLS achieves its objectives, consider again the data matrix $\mathbf{X} \in \mathbb{R}^{N \times m}$ 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, $\mathbf{Y} \in \mathbb{R}^{N \times p}$. 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:

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

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

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

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

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

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

$$\begin{aligned} \mathbf{X} &= \mathbf{T}\mathbf{P}^T + \mathbf{E} = \sum_{i=1}^k \mathbf{t}_i \mathbf{p}_i^T + \mathbf{E} \\ \mathbf{Y} &= \mathbf{U}\mathbf{Q}^T + \mathbf{F} = \sum_{i=1}^k \mathbf{u}_i \mathbf{q}_i^T + \mathbf{F} \end{aligned} \quad \text{eq. 14}$$

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

$$\begin{aligned} \mathbf{u}_i &\approx b_i \mathbf{t}_i \\ \text{or} \quad \mathbf{U} &= [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k] \approx [b_1 \mathbf{t}_1, b_2 \mathbf{t}_2, \dots, b_k \mathbf{t}_k] = \mathbf{T}\mathbf{B} \end{aligned}$$

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

$$\hat{\mathbf{Y}} = \hat{\mathbf{U}}\mathbf{Q}^T = \mathbf{T}\mathbf{B}\mathbf{Q}^T \quad \text{eq. 15}$$

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

5.5 Soft Sensing via PLS for Pulp & Paper Manufacturing Process

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

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

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

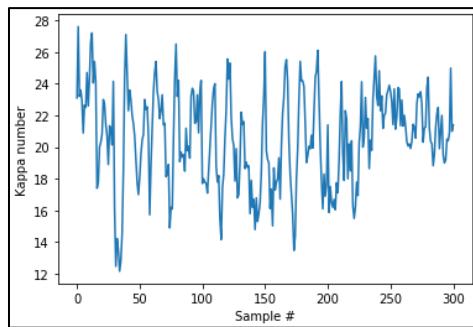


Figure 5.15: Kappa number data in Kamyr dataset

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

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

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

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

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

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

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

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

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

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

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

We are now ready to build the PLS model.

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

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

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

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

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

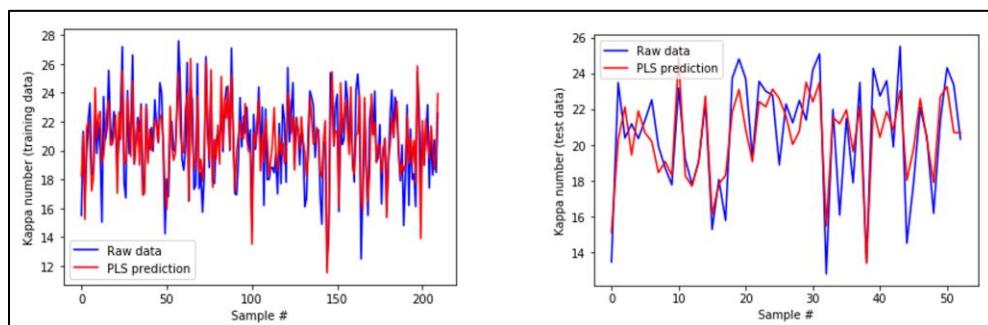


Figure 5.16: Measured vs predicted Kappa number for training and test data



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

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

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

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

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

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

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

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

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

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

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

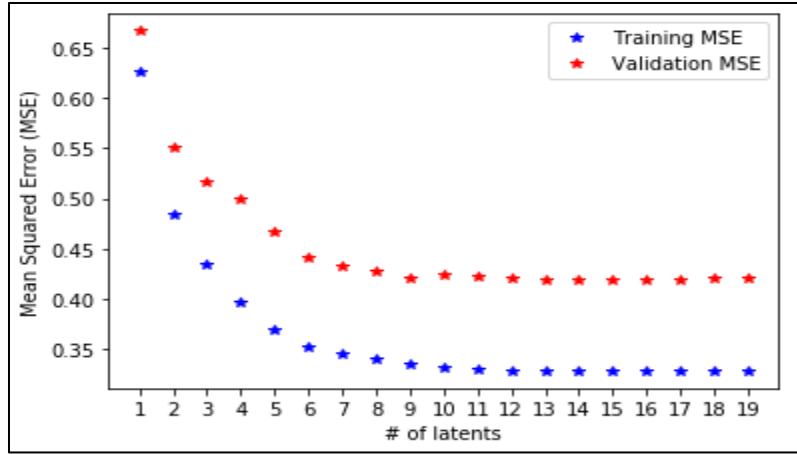


Figure 5.17: MSE plot for `n_components` determination

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

5.6 Process Monitoring via PLS for Polyethylene Manufacturing Process

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

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

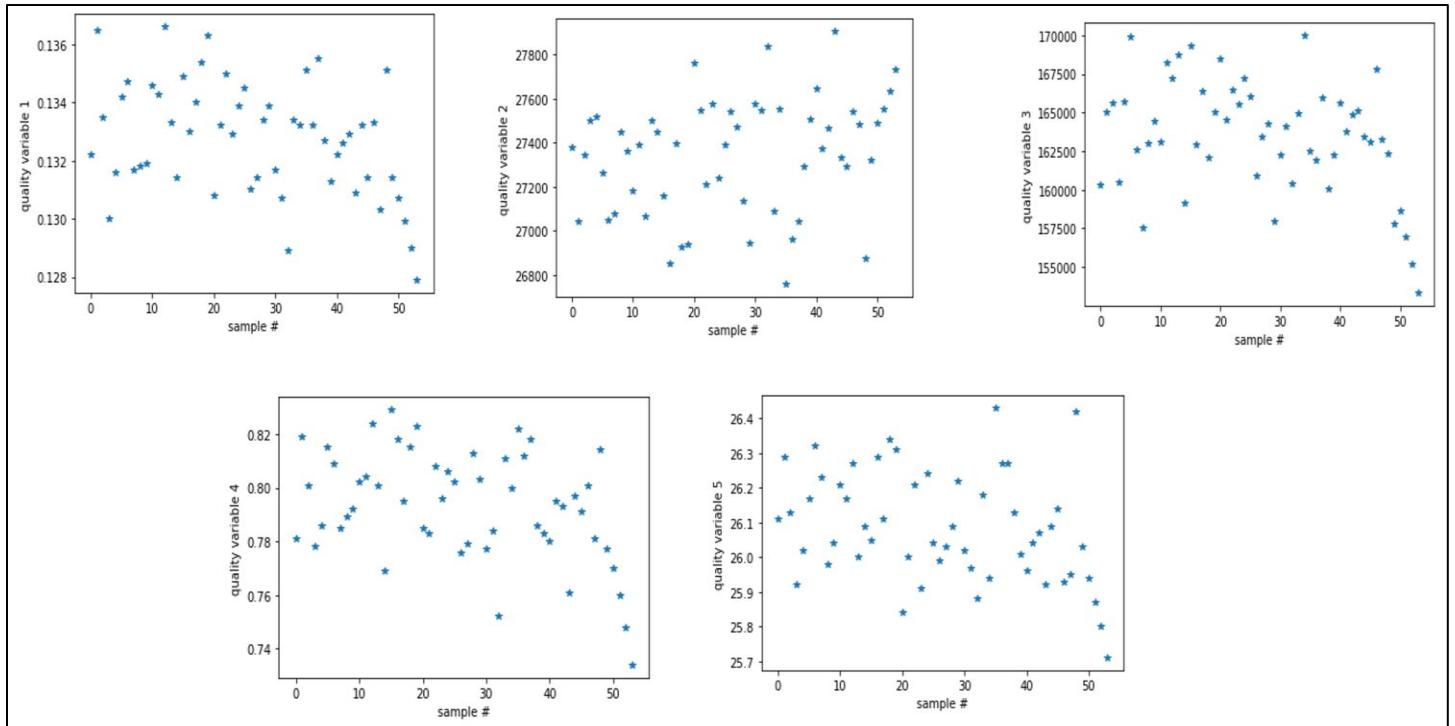


Figure 5.18: Plot of quality variables in LDPE dataset

We begin by building a PLS model using 3 latent components²³.

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

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

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

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

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

Computation of captured variances reveal that just 56% of the information in \mathbf{X} can explain almost 90% of the variation in \mathbf{Y} ; this implies that there are variations in \mathbf{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 \mathbf{t} vs \mathbf{u} score plots further confirms that linear correlation was a good assumption for this dataset. We can also see how the correlation becomes poor for higher components.

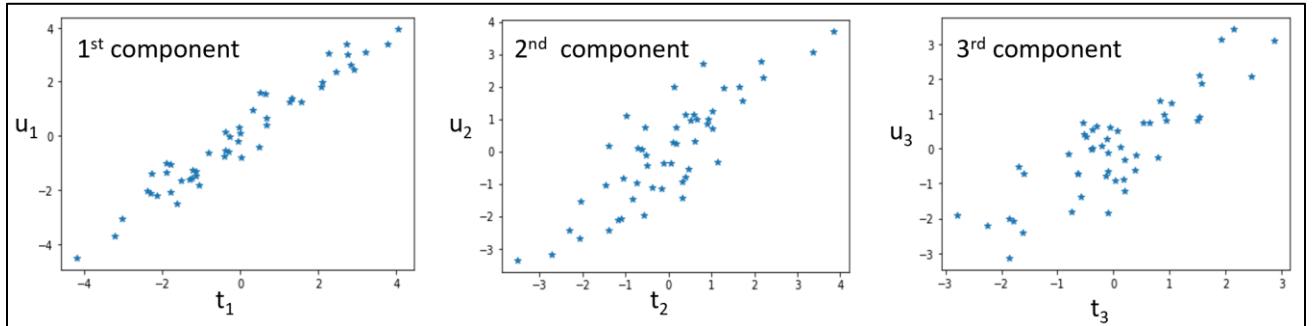


Figure 5.19: X-scores vs Y-scores. Here \mathbf{t}_j and \mathbf{u}_j refer to the j^{th} columns of \mathbf{T} and \mathbf{U} matrices, respectively.

Fault detection indices

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

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

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

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

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

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

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

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

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

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

χ_α^2 is the $(1-\alpha)$ percentile of a chi-squared distribution²⁴ with h degrees of freedom; μ denotes the mean value and σ denotes the variance of the SPE statistic. Note that this expression could be used for PCA SPE statistic as well.

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

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

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

# SPEy
```

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

```

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

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

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

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

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

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

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

```

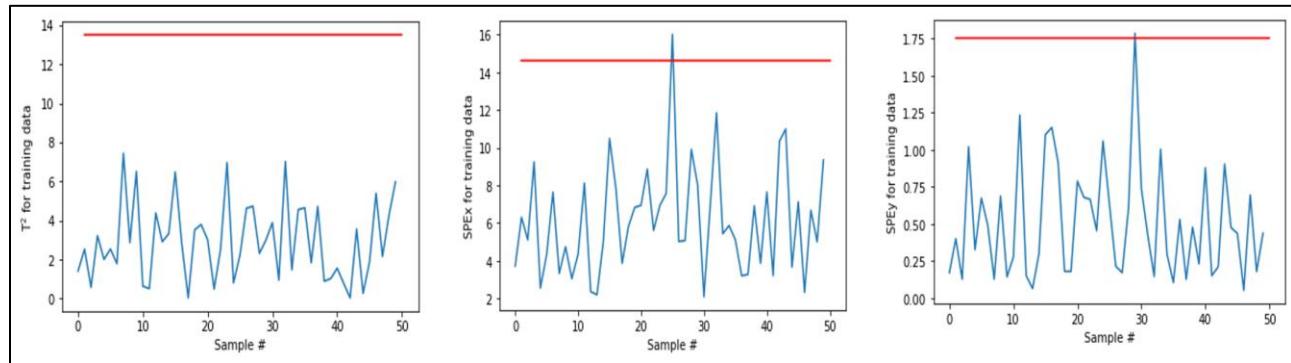


Figure 5.20: Monitoring charts for LDPE training data

Fault detection on test data

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

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

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

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

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

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

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

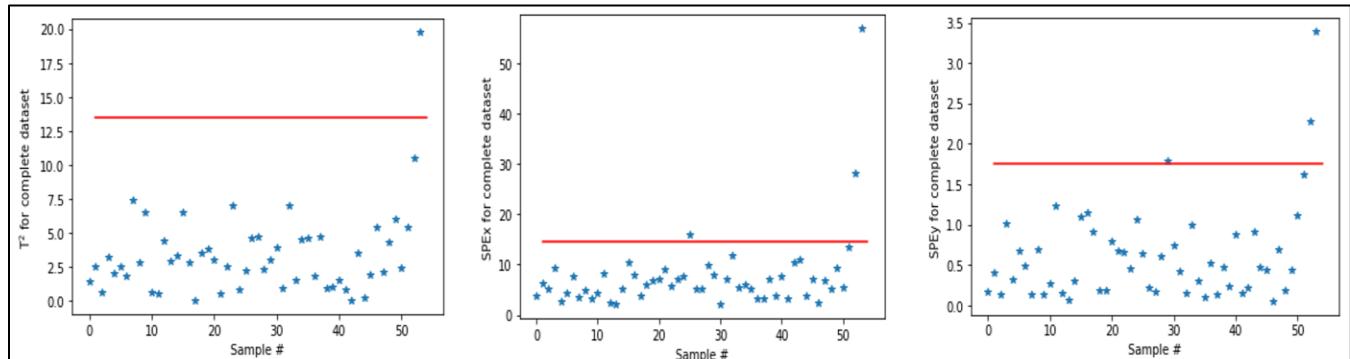


Figure 5.21: Monitoring charts for the complete LDPE dataset

5.7 Variants of Classical PLS

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

therefore including past measurements improve predictive accuracy²⁵. For dynamic PLS, an augmented input data matrix is built (using the same procedure as that for dynamic PCA) using lagged values of only input variables or both input and output variables. Using only lagged inputs leads to a FIR (finite impulse response) model while using both lagged inputs and outputs leads to an ARX (autoregressive with exogenous variables) model. The value of lag period can be determined using cross-validation or time-constant of the process.

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

Model maintenance for time-varying process

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

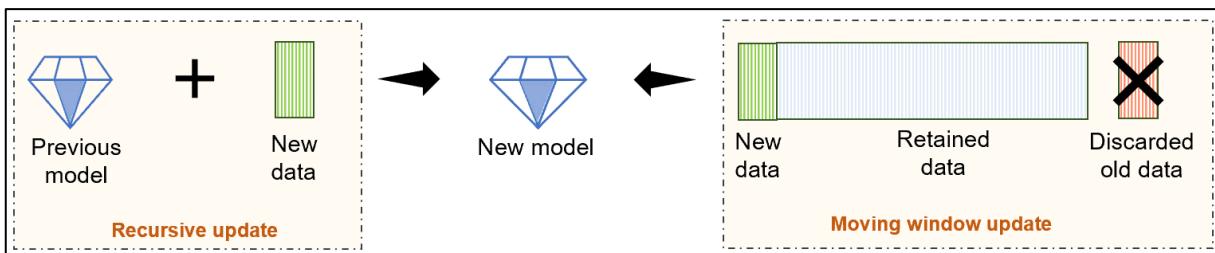


Figure 5.22: Model-maintenance schemes

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

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

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

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

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

Summary

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

Chapter 6

Dimension Reduction and Latent Variable Methods (Part 2)

By now you must be very impressed with the powerful capabilities of PCA and PLS techniques. These methods allowed us to extract uncorrelated latent variables which maximized the captured variances. However, you may ask, “Are these the best dimensionality reduction techniques to use for all problems?”. We are glad that you asked! Other powerful methods do exist which may provide better performance. For example, independent component analysis (ICA) 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. If you are reducing dimensionality with the end goal of classifying 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) would be better suited for this task.

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
-

6.1 ICA: An Introduction

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

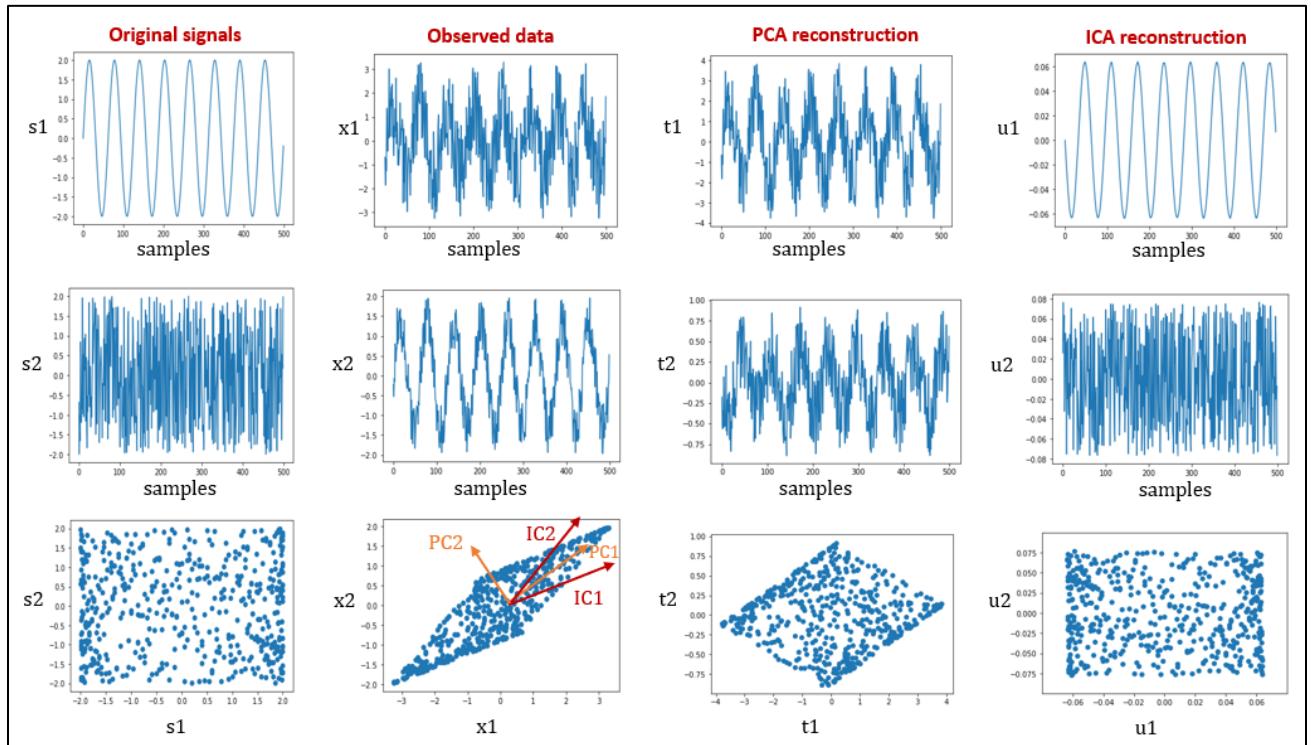


Figure 6.1: Simple illustration of ICA vs PCA. The arrows in the x_1 vs x_2 plot show the direction vectors of corresponding components. Note that the signals t_1 and t_2 are not independent as value of one variable influences the range of values of the other variable.

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

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

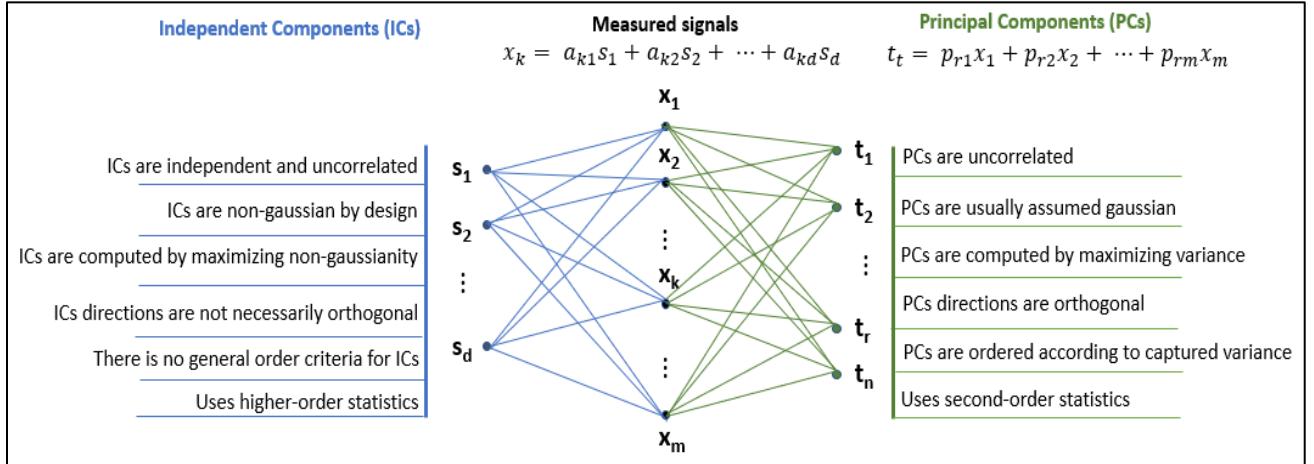


Figure 6.2: Differences between ICA and PCA

Independence vs Uncorrelatedness

Before jumping into the mathematics behind ICA, let us take a few seconds to ensure that we understand the concepts behind independence and uncorrelatedness. Two random variables 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 $\mathbf{X} \in \mathbb{R}^{m \times N}$ 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 .

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

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

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

Before estimating \mathbf{W} , the initial step in ICA involves removing correlations between the variables in the data matrix \mathbf{X} . The step, called as whitening or spherling, is accomplished via PCA.

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

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

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

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,

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

Therefore, once \mathbf{B} is found, \mathbf{W} can be generated. Matrix \mathbf{B} is estimated by maximizing the non-Gaussianity of the estimated IC values, i.e., each row of \mathbf{S} (\mathbf{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 components²⁷ that we need.

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

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

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

Complex chemical process: Tennessee Eastman Process (TEP)

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

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

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

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

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

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

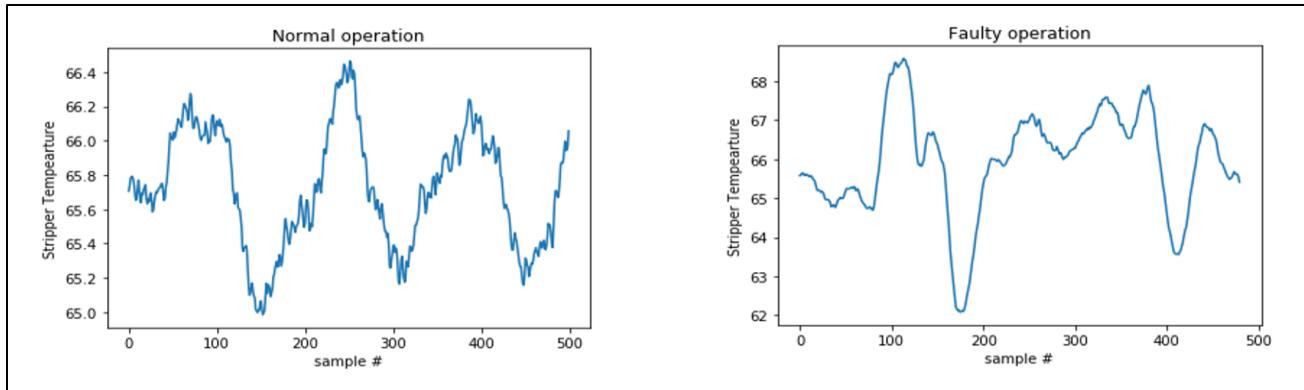


Figure 6.3 (a): Normal vs faulty process profile in Tennessee Eastman dataset

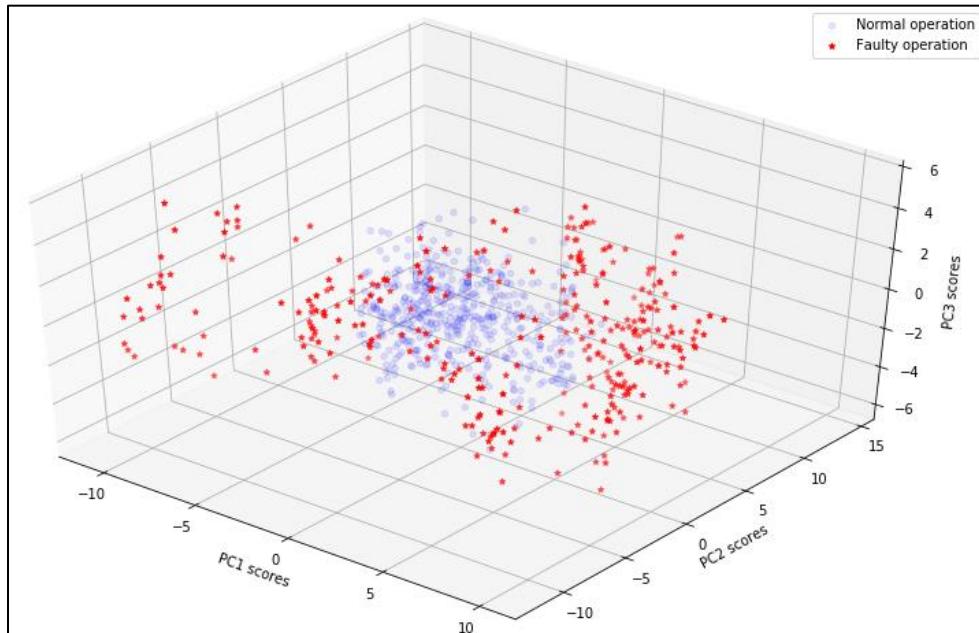


Figure 6.3 (b): Normal vs faulty (Fault 10) TE process data in PC space

Deciding number of ICs

The number of ICs returned by FastICA equals the dimension of the measured space. Previously, we saw in PCA that not all extracted latent variables are important; only a few dominant/important components contain majority of the information while the rest contain noise or trivial details. Moreover, the extracted PCs were ordered according to their variance. However, there is no standard criterion for quantifying the importance of ICs and selecting the optimal number of ICs automatically. One popular approach (described below) for quantifying component importances was suggested by Lee et. al²⁸.

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

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

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

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

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

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

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

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

# re-arrange rows of W matrix
W_sorted = W[sort_order,:]
```

row 1 now corresponds to the most important IC and so on

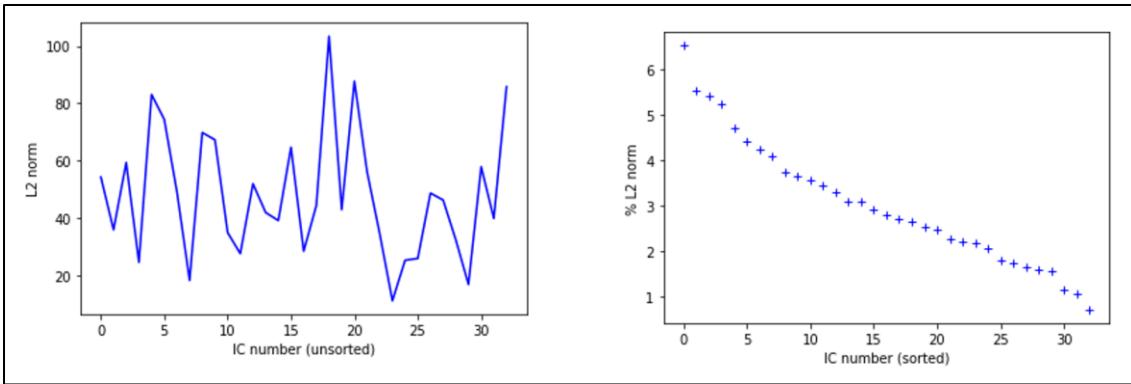


Figure 6.4: (Unsorted) L_2 norm of each row of \mathbf{W} and (sorted) percentage of the L_2 norms

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

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

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

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

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

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

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

6.2 Process Monitoring via ICA for Tennessee Eastman Process

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

Fault detection indices

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

$$I^2 = \mathbf{s}_i^T \mathbf{s}_i \quad \text{eq. 6}$$

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.

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

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

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

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

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

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 6.5 shows the monitoring charts along with 99% control limits.

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

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

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

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

    # compute le2
    We = W_sorted[n_comp:,:]
    Se = np.dot(We, data.T)
    le2 = 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')

```

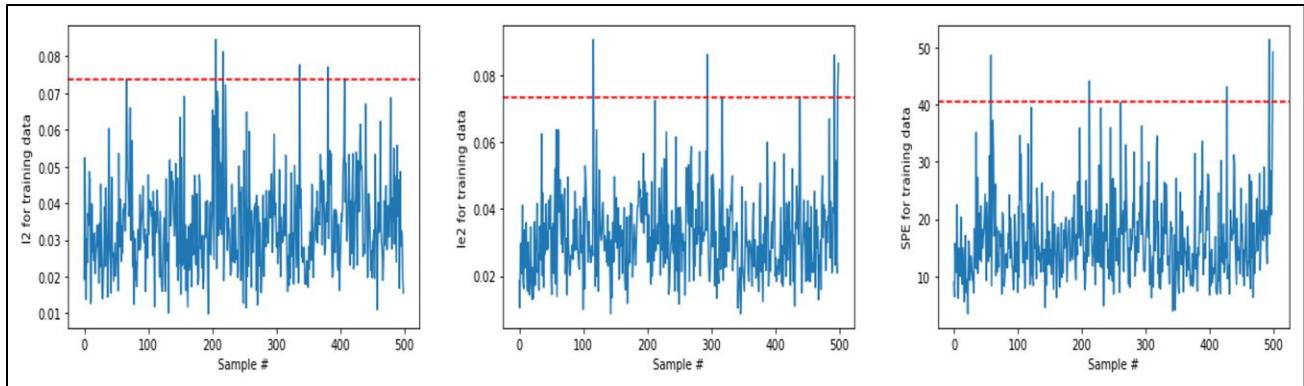


Figure 6.5: ICA monitoring charts for TEP training data

Fault detection for test data

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

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

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

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

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

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

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

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

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

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

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

    return alarmRate

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

xmeas = TEda_Fault_test[:,0:22]
xmv = TEda_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, le2_CL, SPE_CL], 'test')

# compute FAR or FDR
alarmRate = compute_alarmRate(ICA_statistics_test[160:,:], [I2_CL, le2_CL, SPE_CL])
                                # fault starts sample 160 onwards
```

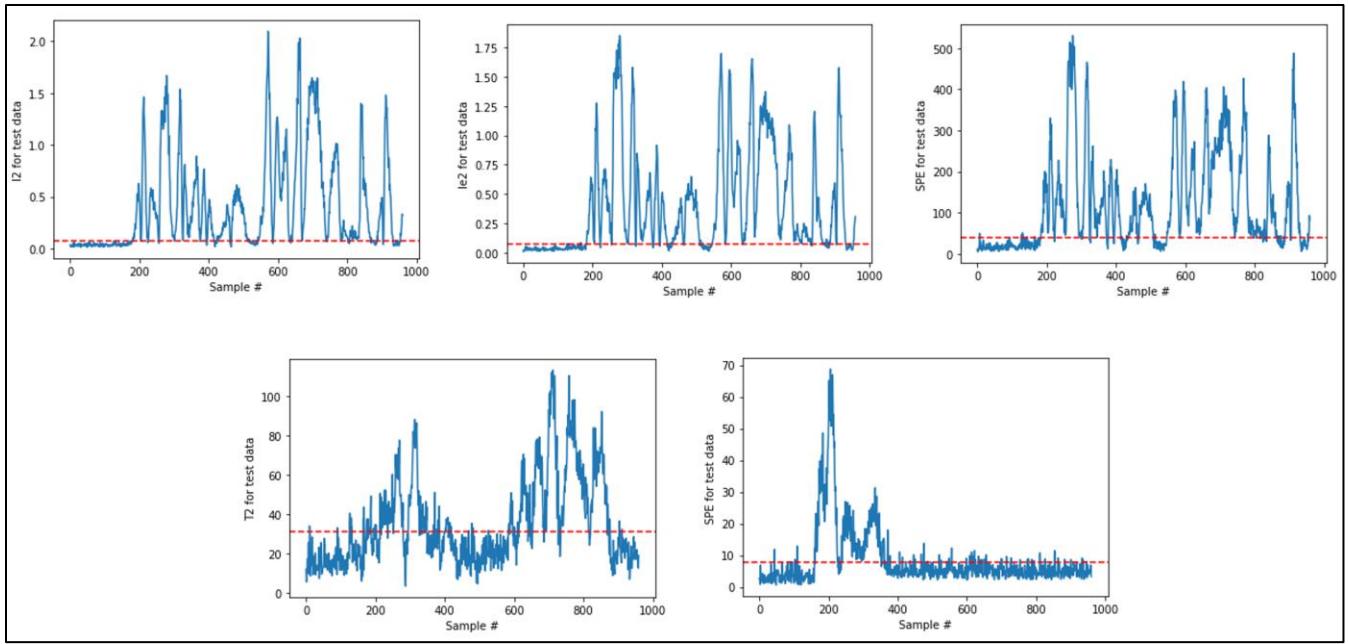


Figure 6.6: Monitoring charts for Fault 10 data with ICA (top, FDR = 90.8%) and PCA (bottom, FDR = 75.6%) metrics

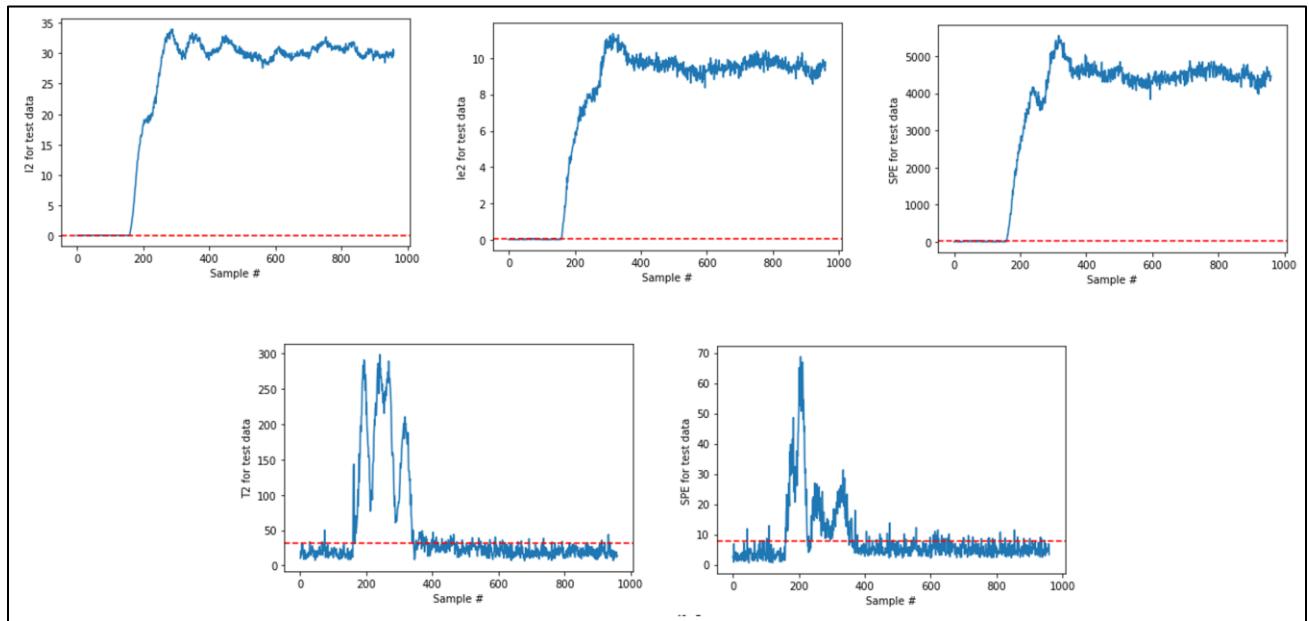


Figure 6.7: Monitoring charts for Fault 5 data with ICA (top, FDR = 100%) and PCA (bottom, FDR = 41.5%) metrics

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

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

6.3 FDA: An Introduction

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

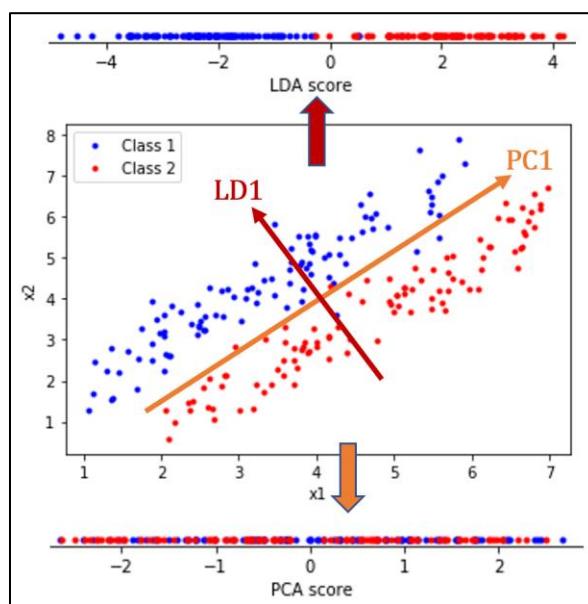


Figure 6.8: Simple illustration of FDA vs PCA. The arrows in the x_1 vs x_2 plot show the direction vectors of 1st components of the corresponding methods

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

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

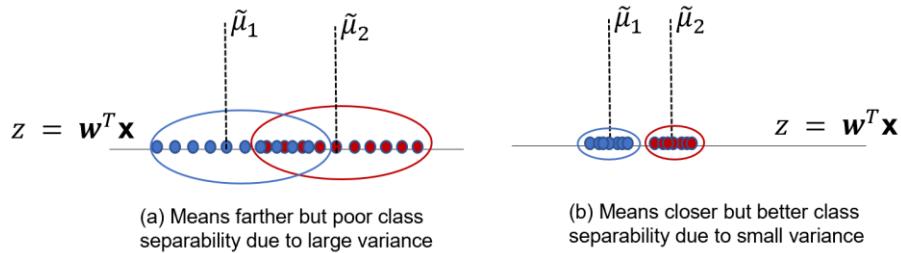
experience different kinds of process issues/faults and it is imperative to accurately identify the specific issue quickly during online operations to minimize economic losses. During model training, FDA learns from data collected from historical process failures (data from a specific process fault form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

Mathematical background

To facilitate data classification, FDA not only maximizes the separation between the classes but also minimizes the variation/scatter within each class. To see how this is achieved, let us first consider a dataset matrix $X \in \mathbb{R}^{N \times m}$ 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

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

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.

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

Using the base relation $y = \mathbf{w}^T \mathbf{x}$ and straightforward algebraic manipulations³², one can equivalently represent the objective $J(\mathbf{w})$ as follows which also holds for any (p) number of data classes

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

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

$$\begin{aligned} \mathbf{S}_b &= \sum_{j=1}^p N_j (\boldsymbol{\mu}_j - \boldsymbol{\mu})(\boldsymbol{\mu}_j - \boldsymbol{\mu})^T \\ \mathbf{S}_w &= \sum_{j=1}^p \mathbf{S}_j \\ \mathbf{S}_j &= \sum_{x_i \in \omega_1} (\mathbf{x}_i - \boldsymbol{\mu}_j)(\mathbf{x}_i - \boldsymbol{\mu}_j)^T \end{aligned}$$

where, $\boldsymbol{\mu} \in \mathbb{R}^m$ and $\boldsymbol{\mu}_j \in \mathbb{R}^m$ 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, \mathbf{w}_1 , is found by maximizing $J(\mathbf{w})$ and the subsequent vectors are found by solving the same problem with the added constraints of orthogonality to previously computed vectors. Note that there can be at most $p-1$ FDA vectors. Alternatively, like PCA, the vectors can also be computed as solutions of a generalized eigenvalue problem

$$\mathbf{S}_w^{-1} \mathbf{S}_b \mathbf{w} = \lambda \mathbf{w} \quad \text{eq. 10}$$

where $\lambda = J(\mathbf{w})$. 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

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

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

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

Dimensionality reduction for Tennessee Eastman Process

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

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

# select variables (discarding composition measurements)
xmeas = TEdata_Faulty_train[:,0:22] # 22 continuous process variables
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 6.9 shows the transformed samples in 2 dimensions after FDA and PCA. FDA is able to provide a clear separation of Fault 5 samples; however, it could not separate Faults 10 and 19 data. Infact, the 2nd discriminant (FD2) contributes little to the discrimination and therefore, only FD1 is needed to separate samples from Fault 5. To segregate Faults 10 and 19, kernel FDA can be explored³³. Linear PCA, on the other hand, fails to separate any of the classes.

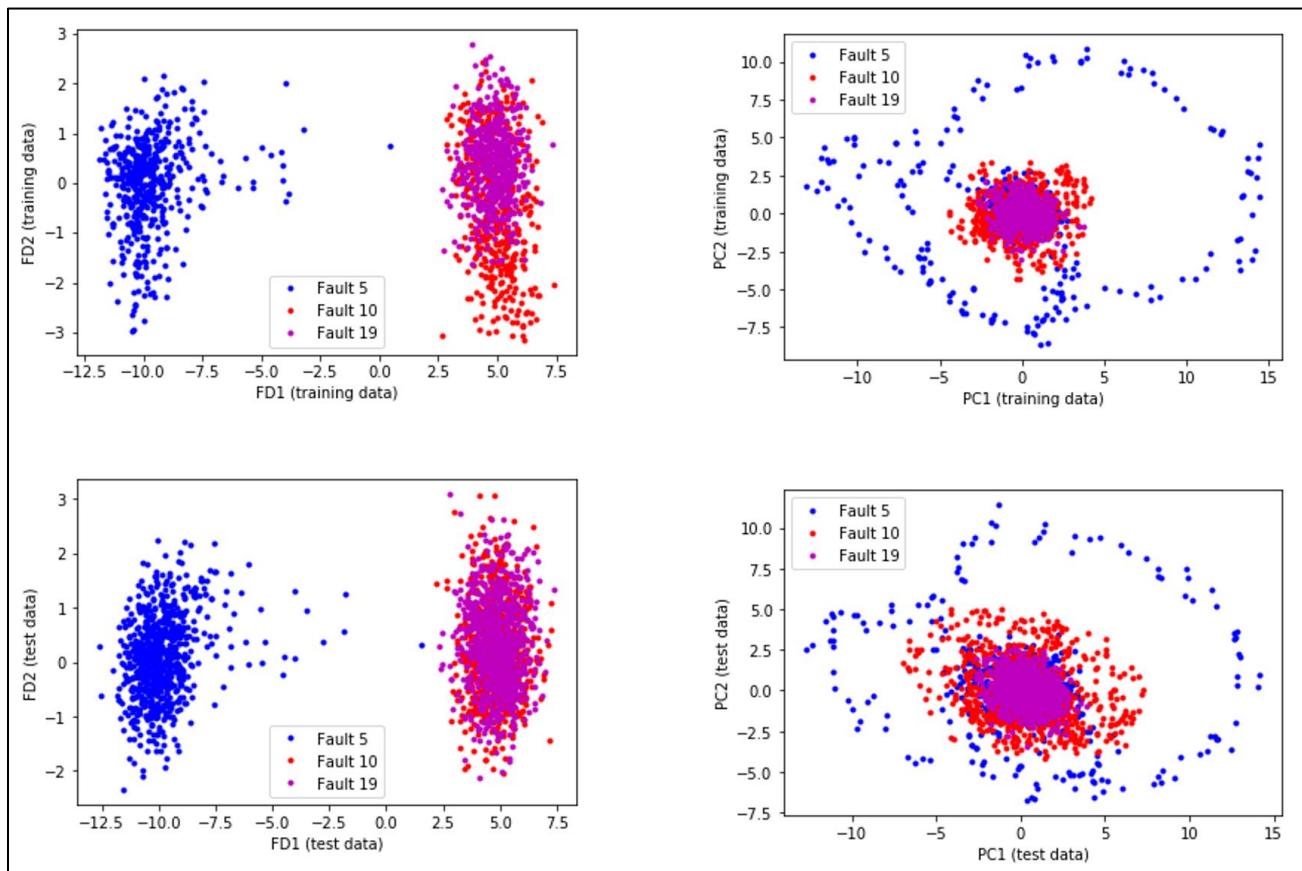


Figure 6.9: FDA and PCA scores in 2 dimensions with 3 fault classes from TEP training (top) and test (bottom) dataset

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

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

6.4 Fault Classification via FDA for Tennessee Eastman Process

After projecting the samples onto the FDA space, any classification technique can be chosen to classify or diagnose the specific fault. A popular T^2 statistic-based approach entails computing a T^2 control limit for each fault class using the training data. The T^2 control limit ($T_{CL,j}^2$) 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

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

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

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

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

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

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

```

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

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

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

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

```

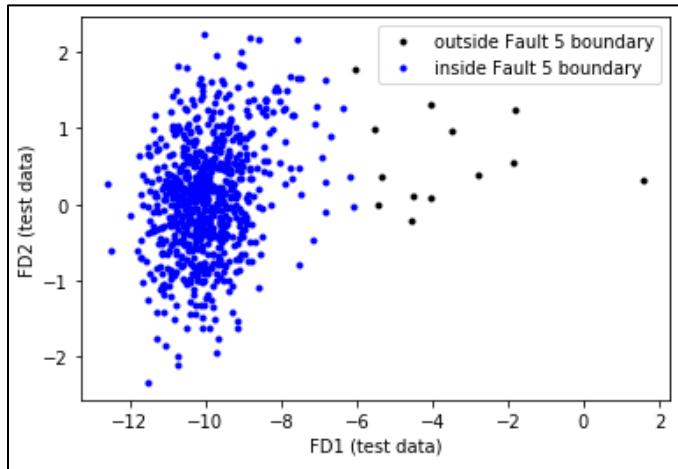


Figure 6.10: Classification result for TEP Fault 5 test samples

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

Process Monitoring Terminology

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

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

Summary

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

Chapter 7

Support Vector Machines & Kernel-based Learning

In previous chapters, we saw methods that help us overcome the ‘curse of dimensionality’. Wouldn’t it be great to have ML methods that do not ‘mind’ dealing with high-dimensional systems? SVM (support vector machine) is one such algorithm which excels in dealing with high-dimensional, nonlinear, and small or medium-sized data. SVMs, by design, minimize overfitting to provide excellent generalization performance. Another major quality of SVMs is that they are extremely flexible and can be employed for classification and regression tasks in both supervised and unsupervised settings. You may already be thinking, “Wow! SVMs seem to pack a lot of good features.” 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.

Above, we mentioned only some of the features of SVMs. You will find more as you work through this chapter. In terms of use in process industry, SVMs have been employed for fault classification, process monitoring, outlier detection, soft sensing, etc.

To understand different aspects of SVMs, we will cover the following topics in this chapter

- Fundamentals of SVMs
- The kernel trick for nonlinear modeling
- SVDD (support vector data description) for unsupervised classification
- Fault detection via SVDD for semiconductor manufacturing process
- Fundamentals of SVR (support vector regression)
- Soft sensing via SVR in a polymer plant and a petroleum refinery

7.1 SVMs: An Introduction

The classical SVM is a supervised linear technique for solving binary classification problems. For illustration, consider Figure 7.1a. 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.

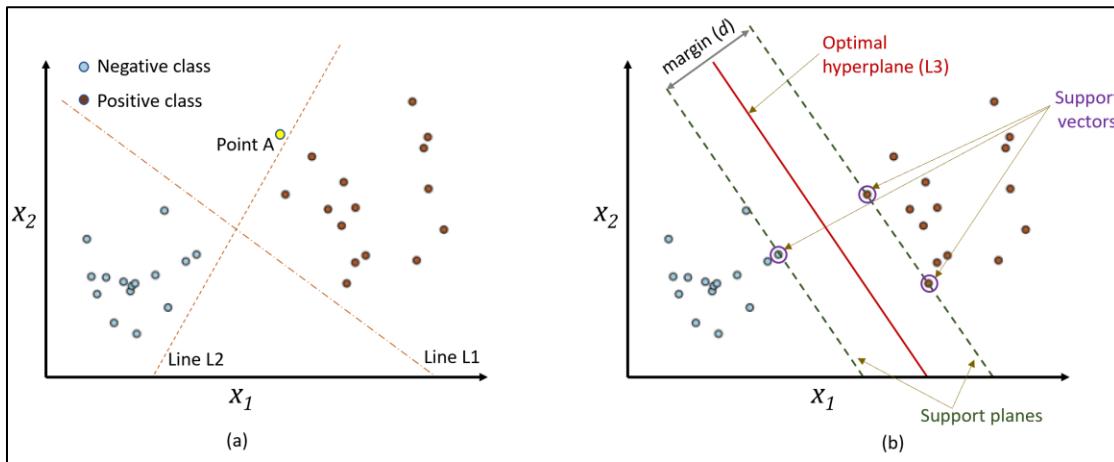


Figure 7.1: (a) Training data distribution with test sample A (b) Optimal separating boundary

The optimal separating line/decision boundary, line L3 in Figure 7.1b, 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 7.1, 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 vectors³⁴ and completely determine the optimal boundary, and hence the name, support vector machines. In Figure 7.1, 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.

³⁴ 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 (\mathbf{x}, y) where \mathbf{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 $\mathbf{w}^T \mathbf{x} + b = 0$ where the model parameters (\mathbf{w}, b) are found such that

$$\begin{aligned}\mathbf{w}^T \mathbf{x}_i + b &\geq 1 && \text{for positive samples } (y_i = 1) \\ \mathbf{w}^T \mathbf{x}_i + b &\leq -1 && \text{for negative samples } (y_i = -1)\end{aligned}$$

The supporting planes get represented as $\mathbf{w}^T \mathbf{x} + b = 1$ and $\mathbf{w}^T \mathbf{x} + b = -1$. 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 $\frac{2}{\|\mathbf{w}\|}$ where $\|\mathbf{w}\|$ is vector norm. Therefore, model parameters are found by maximizing $\frac{2}{\|\mathbf{w}\|}$ (or equivalently, minimizing $\|\mathbf{w}\|$) while meeting the above constraints. Specifically, the following optimization problem is solved

$$\begin{array}{ll}\min_{\mathbf{w}, b} & \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{s.t.} & y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1, \quad i = 1, \dots, N\end{array} \quad \text{eq. 1}$$

Once the model has been fitted, class predictions for test sample, \mathbf{x}_t , are made as follows.

$$\hat{y}_t = \text{sign}(\mathbf{w}^T \mathbf{x}_t + b) \rightarrow \mathbf{x}_t \text{ belongs to positive class if it lies on positive side of separating hyperplane}$$

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 7.1 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 7.1³⁵. 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 7.2. Our SVM formulation (in Eq. 1) will fail to find a solution in this case

³⁵ Check out the online code to see how the separating boundary and support planes are plotted

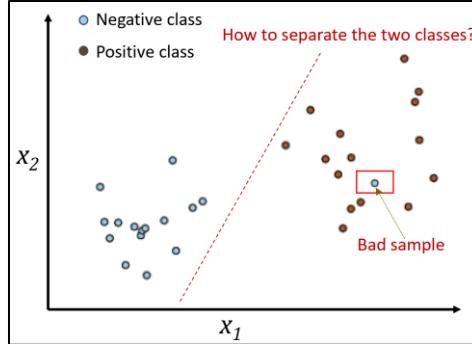


Figure 7.2: Presence of the shown bad sample makes perfect linear separation infeasible

To deal with such scenarios, we add a little flexibility into our SVM optimization program by modifying the constraints as shown below

$$\begin{aligned} \mathbf{w}^T \mathbf{x}_i + b &\geq 1 - \xi_i && \text{for } y_i = 1 \\ \mathbf{w}^T \mathbf{x}_i + b &\leq -1 + \xi_i && \text{for } y_i = -1 \end{aligned}$$

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

$$\begin{aligned} \min_{\mathbf{w}, b, \xi} \quad & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N \xi_i && \text{eq. 2} \\ \text{s.t.} \quad & y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i, \quad i = 1, \dots, N \\ & \xi_i \geq 0 \end{aligned}$$

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 7.2), with the previously shown code, we end up with the same separating boundary as shown in Figure 7.1. Class prediction expression remains the same as $\hat{y}_t = \text{sign}(\mathbf{w}^T \mathbf{x}_t + b)$.

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 7.3. 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.

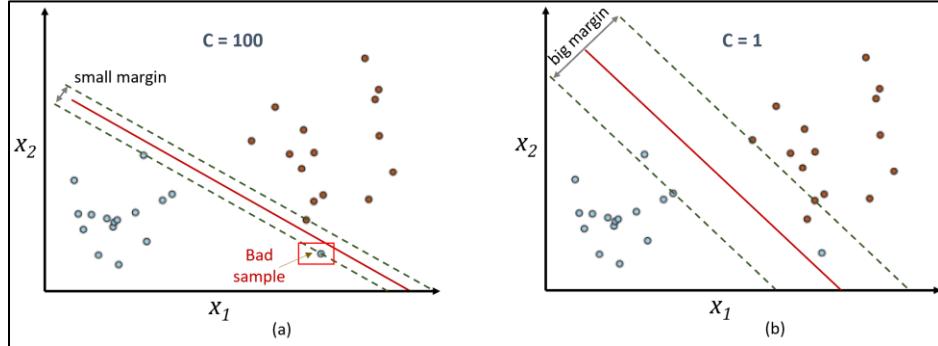


Figure 7.3: (a) Overfitted unregularized boundary (b) Regularized fitting

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.

7.2 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 7.4. It is clear that a linear boundary is inappropriate here.

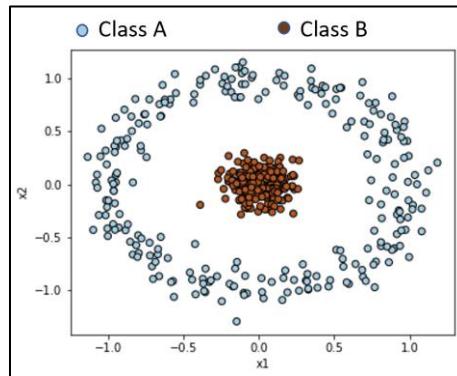


Figure 7.4: Nonlinearly distributed samples

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 7.4, the following transformation would work quite well

$$\varphi(\mathbf{x}) = \varphi(x_1, x_2) = [z_1, z_2, z_3] = [x_1, x_2, x_1^2 + x_2^2]$$

As we can see in Figure 7.5, in the 3D space, the data is easily linearly separable!

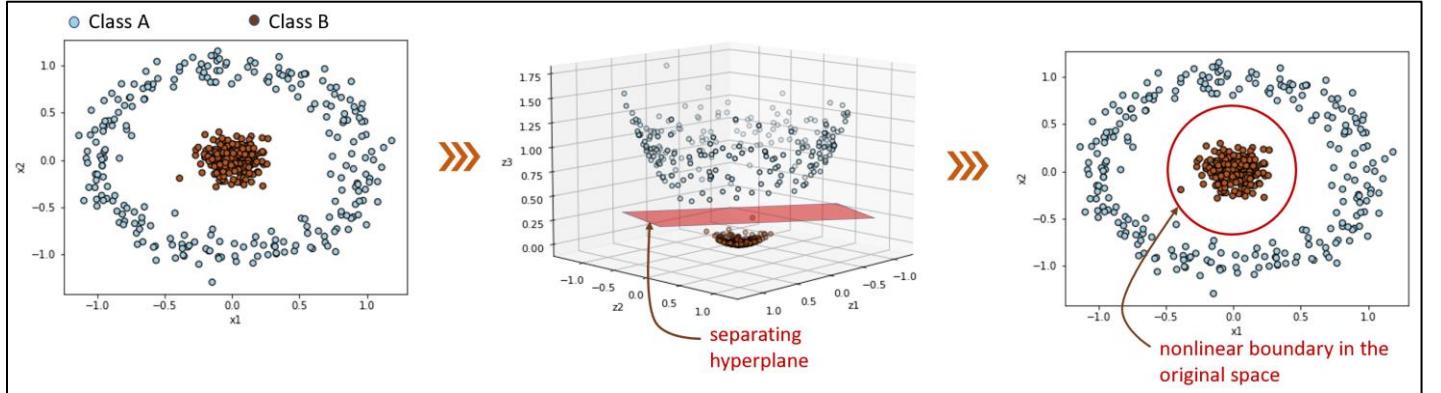


Figure 7.5: Nonlinear mapping to higher dimensional enabling linear segregation

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'. Let's learn what is this trick and how it is used. For this 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

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j - \sum_{i=1}^N \alpha_i \\ \text{s.t.} \quad & \sum_{i=1}^N y_i \alpha_i = 0 \\ & 0 \leq \alpha_i \leq C \quad i = 1, \dots, N \end{aligned} \quad \text{eq. 3}$$

Here, the optimization parameters \mathbf{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, \mathbf{w} and b can be computed via the following

$$\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i, \quad \mathbf{b} = \frac{1}{N_s} \sum_{i \in \{SV\}} 1 - y_i \mathbf{w}^T \mathbf{x}_i$$

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

$$\hat{y}_t = \text{sign}(\sum_{i=1}^N \alpha_i y_i \mathbf{x}_i^T \mathbf{x}_t + b) \quad \text{eq. 4}$$

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

$$\hat{y}_t = \text{sign}(\sum_{i \in \{SV\}} \alpha_i y_i \mathbf{x}_i^T \mathbf{x}_t + b) \quad \text{eq. 5}$$



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

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \varphi(\mathbf{x}_i)^T \varphi(\mathbf{x}_j) - \sum_{i=1}^N \alpha_i \\ \text{s.t.} \quad & \sum_{i=1}^N y_i \alpha_i = 0 \\ & 0 \leq \alpha_i \leq C \quad i = 1, \dots, N \end{aligned} \quad \text{eq. 6}$$

and Eq. 5 becomes

$$\hat{y}_t = \text{sign}(\sum_{i \in \{SV\}} \alpha_i y_i \varphi(\mathbf{x}_i)^T \varphi(\mathbf{x}_t) + b) \quad \text{eq. 7}$$

The most crucial observation here is that the transformed variables ($\varphi(\mathbf{x})$) appear only as inner (dot) products. Now, if somehow, we knew the values of these dot products then we would not need to know the exact form of $\varphi(\mathbf{x})$ or the mapping at all and we can fit our SVM model. This is made possible via kernel functions, $K(\mathbf{x}_i, \mathbf{x}_j)$, which provide the relationship

$$K(\mathbf{x}_i, \mathbf{x}_j) = \varphi(\mathbf{x}_i)^T \varphi(\mathbf{x}_j)$$

As you can see, kernel functions allow us to compute dot products in the ‘unknown’ transformed space as a function of vectors in the original space! There are several forms of kernel functions to choose from and thus this choice becomes a model hyperparameter. Once a kernel function is chosen, Eq. 6 becomes

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^N \alpha_i \\ \text{s.t.} \quad & \sum_{i=1}^N y_i \alpha_i = 0 \\ & 0 \leq \alpha_i \leq C \quad i = 1, \dots, N \end{aligned} \quad \text{eq. 8}$$

Above is the form in which a SVM model is fitted and predictions are made as follows

$$\hat{y}_t = \text{sign}(\sum_{i \in \{SV\}} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_t) + b) \quad \text{eq. 9}$$

These kernel functions allow us to obtain powerful nonlinear classifiers while retaining all the benefits of the original linear SVM method!

Types of kernel functions

Table below lists some commonly used kernel functions. The first one is simply the familiar dot product of two vectors, while the 2nd one, RBF (radial basis function) or Gaussian kernel is the most popular (and usually the default choice) kernel for nonlinear problems.

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

$$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z} + 1)^2$$

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

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

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

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

Therefore, if you use the above polynomial kernel, you are implicitly assuming that your dataset is linearly separable in the shown 6th dimensional feature space! In general, you will have to experiment with the kernel hyperparameters to determine what exact form of polynomial kernel works best for your problem.

If you were amazed by the previous 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.

Sklearn Implementation

Let's try to find the nonlinear classifier boundary for the toy dataset in Figure 7.4 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 $1/2\sigma^2$. Optimal C and gamma come out to be 0.1 and 1, respectively, with the classifier solution shown in Figure 7.6. 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 gamma is concerned, large value (or small σ) also leads to overfitting.

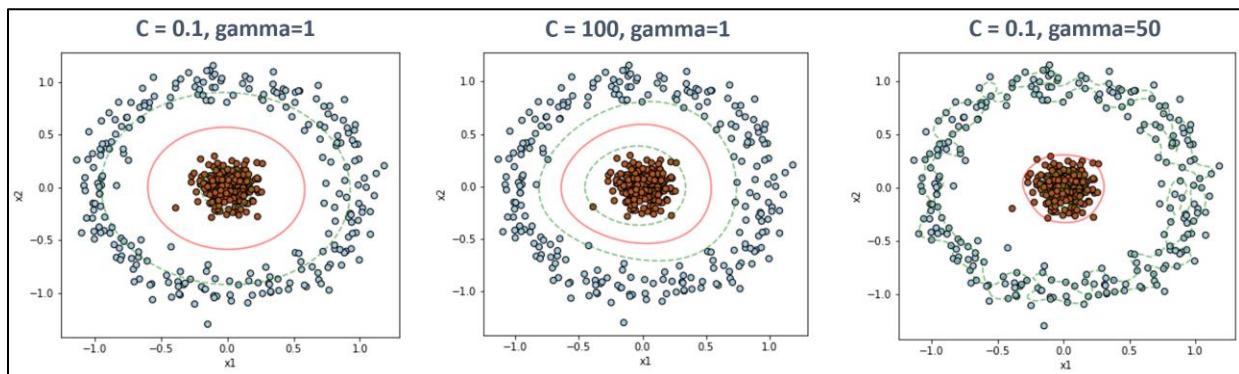
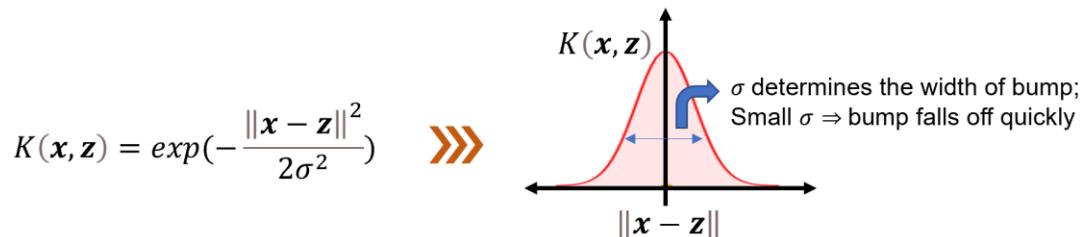


Figure 7.6: Nonlinear binary classification via kernel SVM and impact of hyperparameters.
[Code for plotting the boundaries is provided online]

Closer look on Gaussian kernels

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

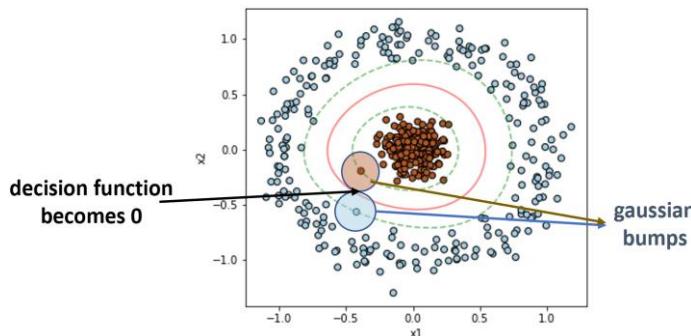


Here, if two points (\mathbf{x} , \mathbf{z}) are close to each-other in the original space, then their similarity (or kernel value) in the mapped space will be higher, compared to when \mathbf{x} and \mathbf{z} are far away from each-other. Now, let's look at the classifier prediction formula for a sample \mathbf{z}

$$\hat{y} = \text{sign}\left(\sum_{i \in \{SV\}} \alpha_i y_i K(\mathbf{x}_i, \mathbf{z}) + b\right)$$

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.

7.3 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 7.7. Here, how do we obtain a model of the training data to be able to call sample X 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.

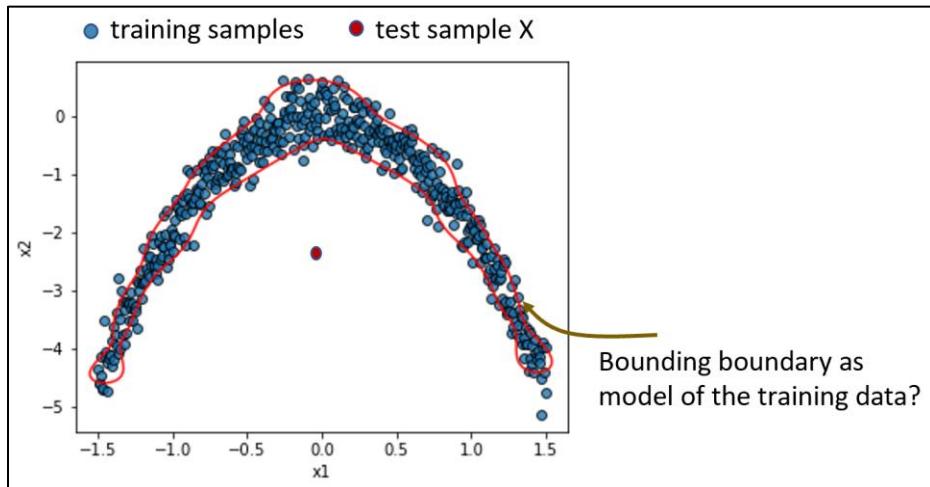


Figure 7.7: 2D training dataset with only one class. Boundary in red shows a potential description model of the dataset that can be used to distinguish the test sample X from training samples.

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 7.7 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

$$\begin{aligned}
Dist(\varphi(x_t), \mathbf{a})^2 &= \|\varphi(x_t) - \mathbf{a}\|^2 \\
&= K(x_t, x_t) - 2 \sum_{i \in \{SV\}} \alpha_i K(x_t, x_i) + \sum_{i \in \{SV\}} \sum_{j \in \{SV\}} \alpha_i \alpha_j K(x_i, x_j)
\end{aligned}$$

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

$$\begin{aligned}
\min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j K(x_i, x_j) \\
\text{s.t.} \quad & \sum_{i=1}^N \alpha_i = 1 \\
& 0 \leq \alpha_i \leq C \quad i = 1, \dots, N
\end{aligned}$$

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 7.7. 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 criterion³⁶, gives bandwidth as follows

$$\begin{aligned}\sigma &= \sqrt{\frac{\bar{D}^2}{\ln\left(\frac{N-1}{\delta^2}\right)}} \\ \bar{D}^2 &= \frac{\sum_{i < j} \|x_i - x_j\|^2}{\frac{N(N-1)}{2}} \\ \delta &= -0.14818008\emptyset^4 + 0.2846623624\emptyset^3 - 0.252853808\emptyset^2 + 0.159059498\emptyset - 0.001381145 \\ \emptyset &= \frac{1}{\ln(N-1)}\end{aligned}$$

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 7.7.

```
# read data
import numpy as np
X = np.loadtxt('SVDD_toyDataset.csv', delimiter=',')
```

³⁶ Kalde & Sadek, The mean and median criterion for kernel bandwidth selection for support vector data description, IEEE 2017

```

# 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, 'sqeuclidean'))/(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 7.8 shows the bounding boundary for different values of gamma with f (or nu 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.

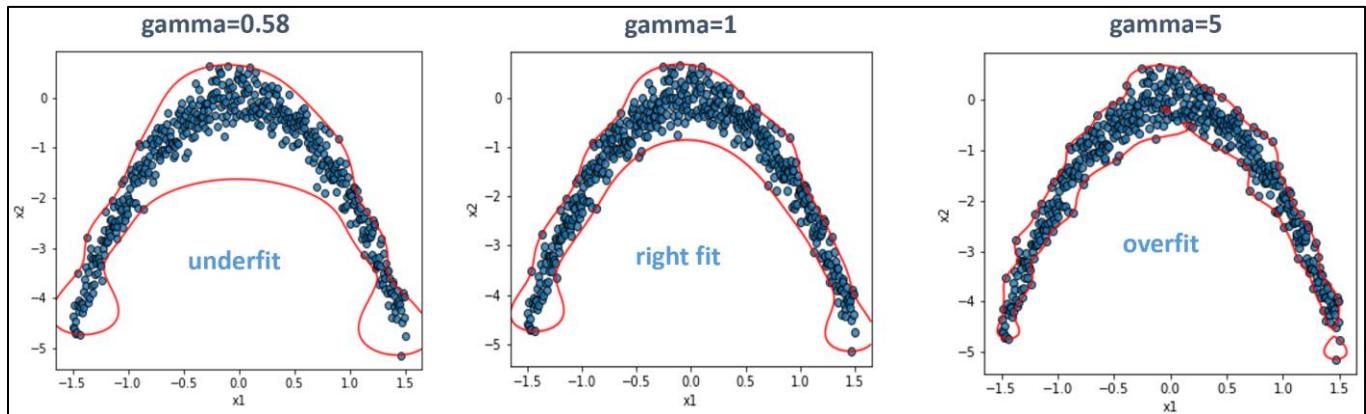


Figure 7.8: SVDD application for data description and impact of model hyperparameter.

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 and then move onto another variant of SVM utilized for regression applications.

PCA vs SVDD for process monitoring



In the previous chapters, we studied dimensionality reduction-based methods (PCA, ICA) for process monitoring. We have now added SVDD to our toolkit. Although it is difficult to say beforehand which method will work best for given scenarios, some rough guidelines do exist. If variables in your dataset are linearly (or only slightly nonlinearly) related and measurements are noisy, PCA/ICA would work best. PCA, however, assumes unimodal distribution of data. SVDD makes no assumption on underlying data distribution and is very-well suited for very nonlinear dataset. However, relationships between different process variables are difficult to interpret in SVM/SVDD models compared to PCA/ICA models.

7.4 Process Fault Detection via SVDD

To illustrate a practical application of SVDD for process monitoring, we will use data from a semiconductor manufacturing process. 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 (see appendix for more details). Figure 7.9 shows the training samples and the faulty test samples in the principal component space.

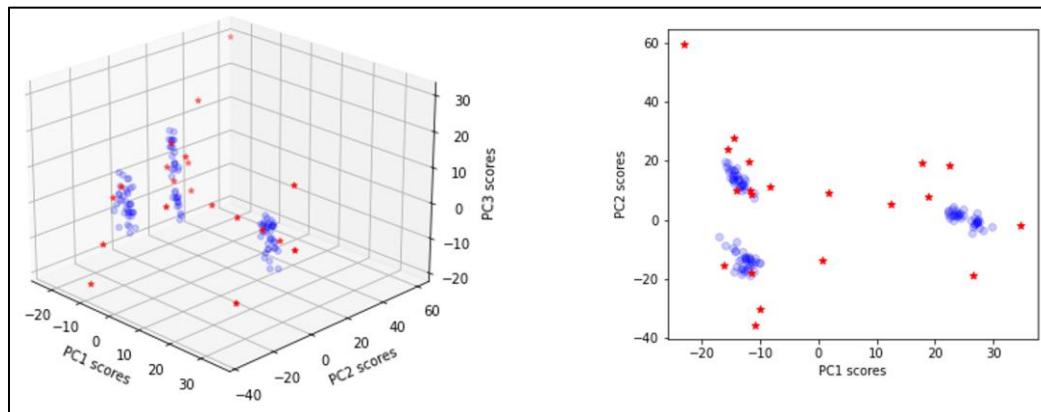


Figure 7.9: Normal (in blue) and faulty (in red) batches in PCA score space

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 7.10 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.

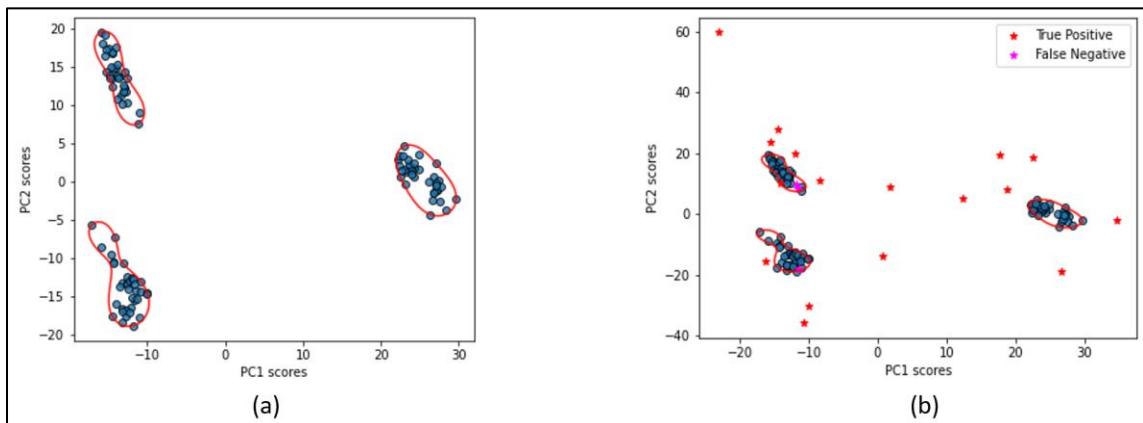


Figure 7.10: (a) SVDD / OC-SVM boundary (in red) around metal-etch training dataset in 2D PC space (b) Position of correctly and incorrectly diagnosed faulty samples

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 GitHub³⁷.

³⁷ <https://github.com/iqiukp/SVDD>

7.5 SVR: An Introduction

Support vector regression (SVR) is another variant of SVM used for linear and nonlinear regressions. SVR attempts to find a regression curve that is as close as possible to the training observations. Geometrically, as shown in Figure 7.11 for kernelized-SVR, a tube of pre-specified width (ε) is fitted to the data and any sample lying outside the tube is penalized. You will see later that SVR's optimization program is designed to obtain a good balance between model generalization capabilities and the ability to fit the training data.

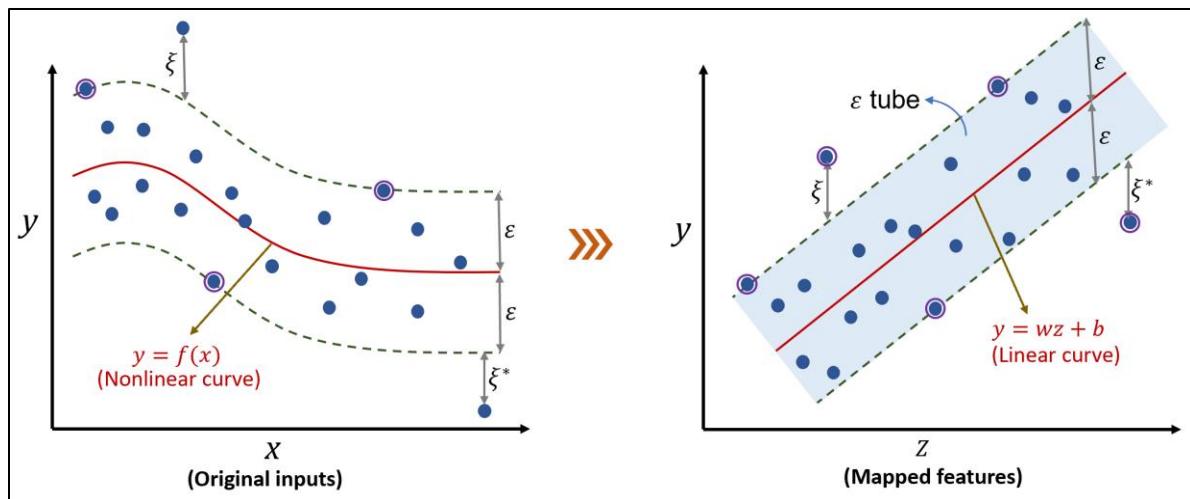


Figure 7.11: Kernelized-SVR illustration for nonlinear curve fitting

SVR generates a linear hyperplane that best describes the output as a function of the inputs. For kernelized SVR, the hyperplane translates into an appropriate nonlinear curve in the original measurement space. As shown in Figure 7.11, in feature space, most of the data-points lie within ε distance to the optimal hyperplane. Slack variables, ξ , allow some samples to fall outside the ε tube; however, such violations are penalized. Samples lying on the edge or outside the tube are the support vectors which completely determine the optimal solution for a given ε . Adding or removing any non-support vector (or samples inside the tube), does not affect the solution.

SVR has been found to provide performance superior to ANNs for small and medium-sized high-dimensional datasets. We will see a couple of applications later in the chapter.

Mathematical Background

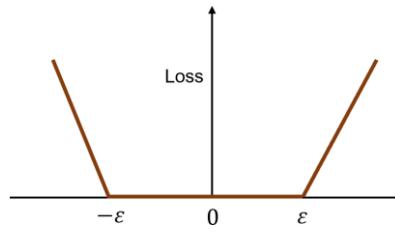
SVR's optimization program for parameter estimation takes the following form in the feature space

$$\begin{aligned}
 \text{Min}_{\mathbf{w}, b, \xi, \xi^*} \quad & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N (\xi_i + \xi_i^*) \\
 \text{s.t.} \quad & y_i - \mathbf{w}^T \varphi(x_i) - b \leq \varepsilon + \xi_i \\
 & \mathbf{w}^T \varphi(x_i) + b - y_i \leq \varepsilon + \xi_i^* \\
 & \xi_i, \xi_i^* \geq 0, \quad i = 1, \dots, N
 \end{aligned}$$

The first part of the objective ($\frac{1}{2} \|\mathbf{w}\|^2$) attempts to keep the model as less complex as possible or the model output as flat as possible. For example, for $y = \mathbf{w}^T \mathbf{x} + b$, $\mathbf{w} = 0$ gives the simplest model $y = b$ with least dependency on inputs. The 2nd part tries to keep the number of samples outside of the ε tube as low as possible. The hyperparameter C controls the trade-off between the two.



In the SVR formulations, there are no errors associated with samples that lie within the ε tube. In SVM literature, this is referred to as implementing ε -insensitive loss function as shown below.



Compare this to an ordinary least squares regression where error from the regression curve for every sample is penalized.

The dual form is given by

$$\begin{aligned}
 \max_{\alpha, \alpha^*} \quad & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N (\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*) K(x_i, x_j) - \\
 & \varepsilon \sum_{i=1}^N (\alpha_i + \alpha_i^*) + \sum_{i=1}^N y_i (\alpha_i - \alpha_i^*) \\
 \text{s.t.} \quad & \sum_{i=1}^N (\alpha_i - \alpha_i^*) = 0 \\
 & 0 \leq \alpha_i, \alpha_i^* \leq C \quad i = 1, \dots, N
 \end{aligned}$$

If you notice carefully, here we assign two multipliers (α, α^*) for each data-point. Both the multipliers end up being 0 for non-support vectors (training samples lying within the ε tube). Once multipliers have been found, predictions can be made by

$$\hat{y}_t = \sum_{i \in \{SV\}} (\alpha_i - \alpha_i^*) K(x_i, x_t) + b$$

The specification of the kernel function and hyperparameters allow us to fit the model (find multipliers) and make predictions. Figure 7.12 shows the impact of the SVR hyperparameters (C , γ , ε) with RBF kernel for a SISO system.

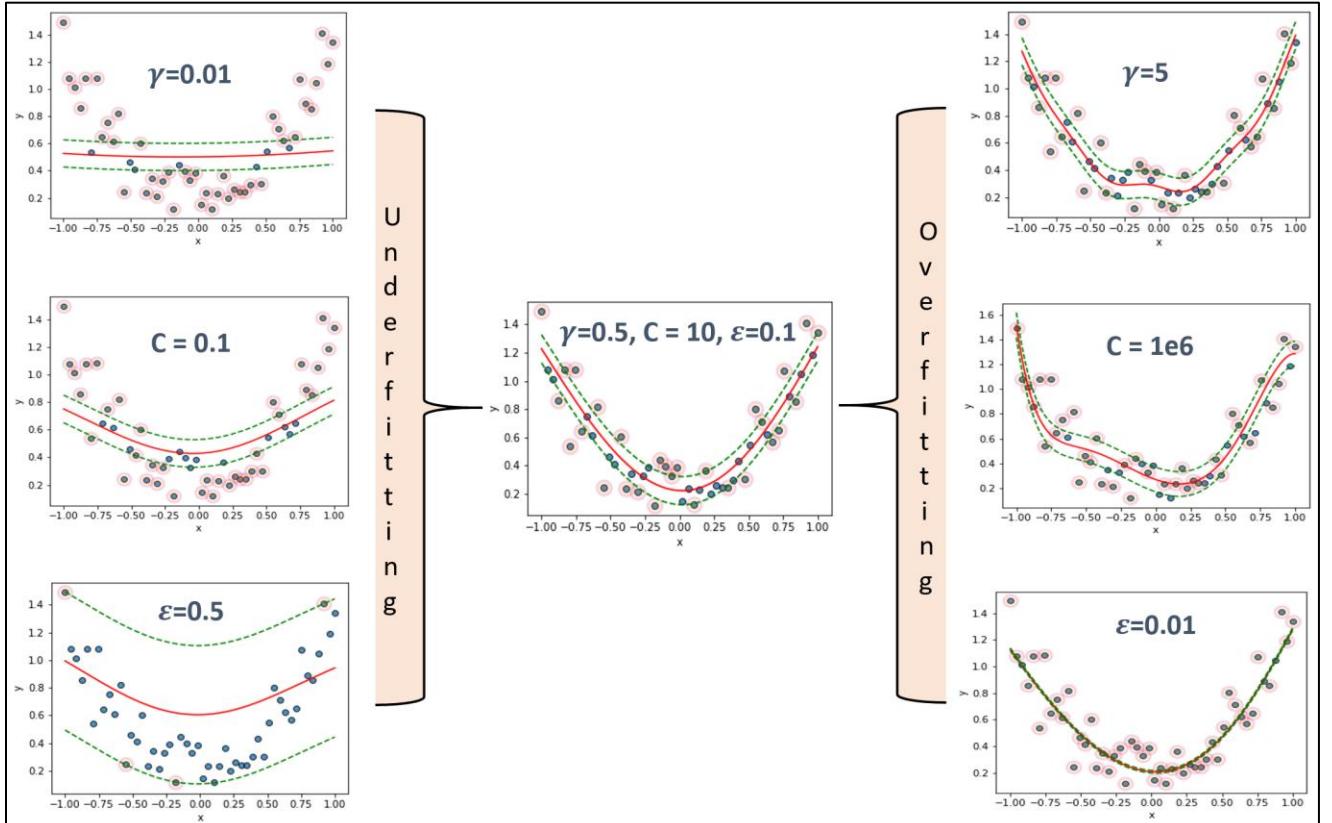


Figure 7.12: Impact of SVR hyperparameters on model fitting quality. Encircled training samples denote the support vectors selected by the respective models.

Figure 7.12 is along the expected lines w.r.t. gamma and C. Let's understand the impact of ε . Large ε makes the ε tube big allowing SVR to keep the model 'flat' leading to underfitting. On the other hand, very small ε makes the tube very small which results in high number of support vectors making the fit prone to overfitting. Nonetheless, it is not uncommon to specify small ε and control overfitting via C. To see SVR implementations using Sklearn, let's check out a couple of industrial applications in the next sections.

SVR vs ANN

Due to the powerful nonlinear modeling capabilities of SVR, it is natural to compare it to artificial neural networks (ANNs). The equation for SVR model prediction can be represented in a network form as shown in figure below. This representation makes the similarities between the two regression methods evident. However, unlike ANNs, the hidden units are automatically determined without any explicit specification of the network structure in SVR. This makes SVR easier to train as there are fewer hyperparameters.

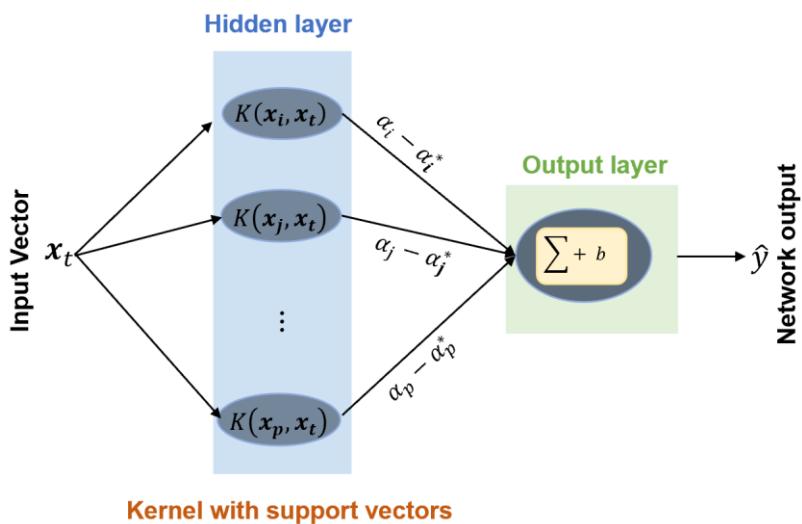


Fig: Network representation of SVR prediction

[We will learn about ANNs in detail in Chapter 11. If you are not familiar with them, you can re-visit this note later. But we figured that if you already know, this comparison could throw some more insights into SVR algorithm.]

In ANNs, the number of fitted model parameters (and therefore the tendency to overfit) increases with increase in dimensionality of input vectors. But in SVR, number of parameters depend on the number of support vectors and not on input dimensionality. Adding more input variables in a SVR model doesn't increase model complexity if the number of support vectors don't change. This greatly helps SVRs overcome overfitting issue arising from high input dimensionality. Moreover, through support vectors, SVR models provide an explicit knowledge of training data-points which are important in defining the regression prediction; this can often help in rationalizing model predictions. Overall, SVRs are as powerful as ANNs for nonlinear modeling with some unique advantageous features.

7.6 Soft Sensing via SVR in a Polymer Processing Plant

For our first illustration, we will apply SVR for soft sensing for predicting plant outputs in a polymer plant. The dataset obtained from Dupont consists of 61 samples of 14 variables (10 predictors, 4 plant outputs). We chose this dataset to illustrate the utility of SVRs for small-sized datasets. Figure 7.13 provides a quick glimpse into the distribution of some of the process inputs and a process output. The sparse nature of data distribution is immediately evident.

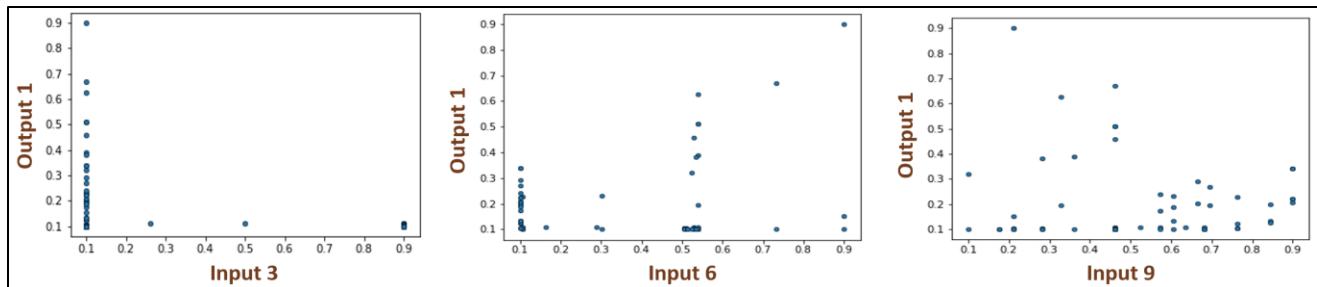


Figure 7.13: Distribution of polymer process inputs and output 1

Sparse data modeling is known to be a difficult task in machine learning. Nonetheless, let's see how well our SVR model can handle this challenge. The code below generates an SVR-based soft sensor model to predict each of the process output one at a time. Hyperparameters are determined via grid-search and k-fold CV.

```
# read data
import numpy as np
data = np.loadtxt('polymer.dat')

X, Y = data[:, 0:10], data[:, 10:]
y = Y[:, 0] # modeling the 1st output

# fit SVR model using grid-search
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV

model = SVR(epsilon=0.01) # default epsilon = 0.1
param_grid = [{"gamma": np.linspace(0.1e-05, 5, 100), 'C': np.linspace(0.01, 5000, 100)}]
gs = GridSearchCV(model, param_grid, scoring='neg_mean_squared_error', cv=10)
```

```

gs.fit(X, y)
print('Optimal hyperparameter:', gs.best_params_)

>>> Optimal hyperparameter: {'C': 50.51, 'gamma': 0.1}

# predict using the best model
y_predicted_SVR = gs.predict(X)

# plots of raw and predicted data
import matplotlib.pyplot as plt

plt.figure()
plt.plot(y, y_predicted_SVR, '.', markeredgecolor='k', markeredgewidth=0.5, ms=9)
plt.plot(y, y, '-r', linewidth=0.5)
plt.xlabel('measured data'), plt.ylabel('predicted data ')

```

Note that all the variables in the raw dataset are already scaled between 0.1 and 0.9 and therefore no further scaling is done before model fitting. Figure below shows the model predictions. Predictions from PLS models are provided for comparison. The first 2 outputs proved difficult for the linear PLS models, but SVR models could provide good fit. Outputs 3 and 4 seem to be more linearly related to the inputs with both PLS and SVR models providing similar fits.

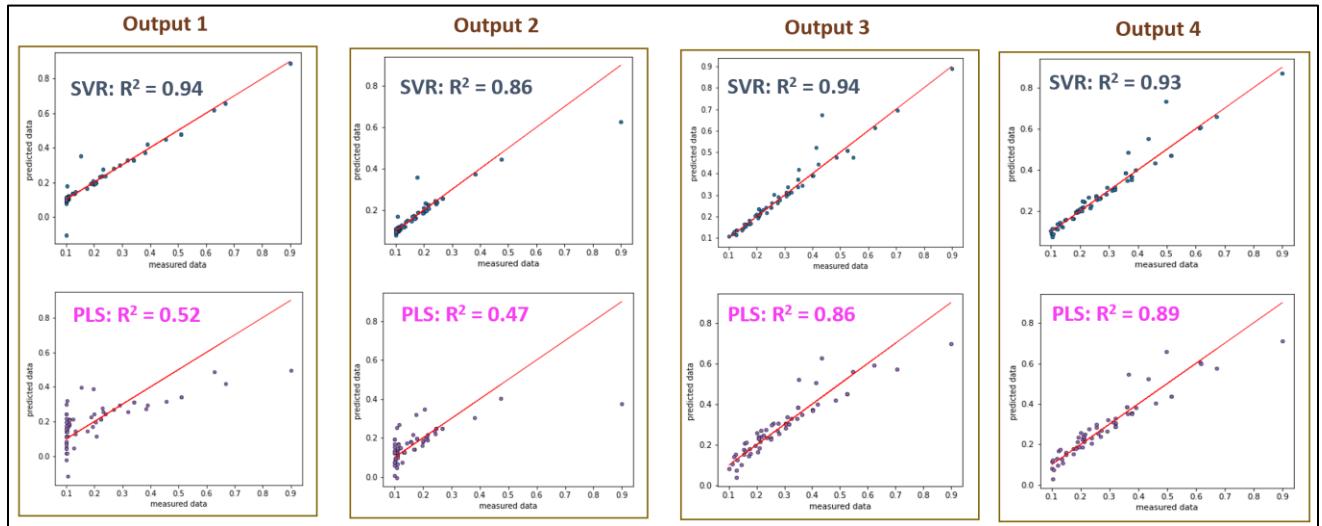


Figure 7.14: SVR and PLS predictions for polymer plant dataset. The red line denotes the ideal $y_{\text{prediction}} = y_{\text{measured}}$ reference.

7.7 Soft Sensing via SVR for Debutanizer Column in a Petroleum Refinery

In this illustration we will use a medium-sized dataset that comes from a debutanizer column operation in a petroleum refinery (see Appendix for system details). The butane (C4) content in gasoline bottoms product of the debutanizer column is not available in real-time and therefore, is required to be predicted using other process data around the column. The dataset contains 2394 samples of input-output process values. Seven process (pressures, temperatures, flows around the column) variables are used as predictors. Figure 7.15 shows that dataset has decent process variability. Note that data have been provided in normalized form.

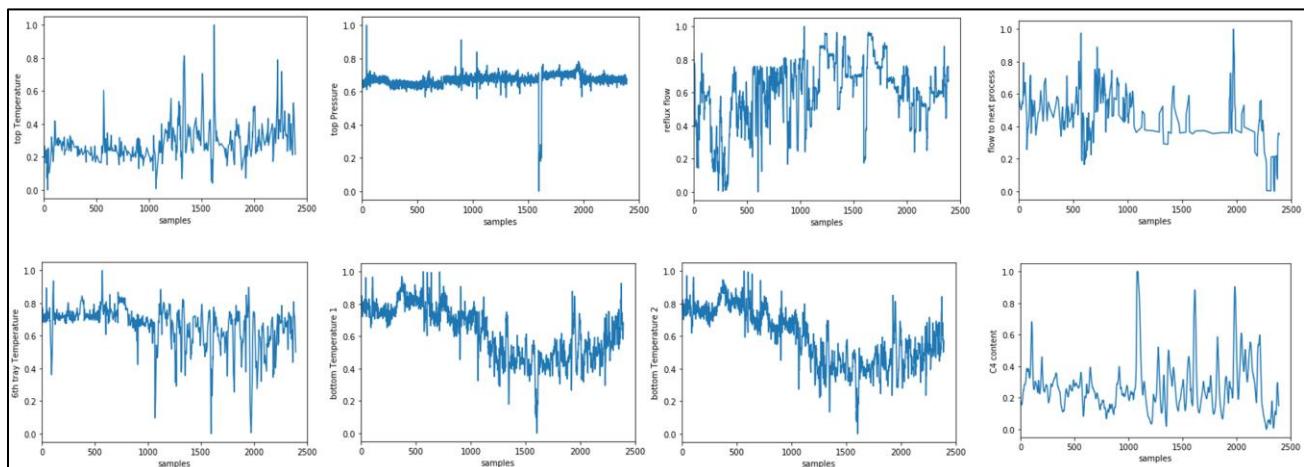


Figure 7.15: Plots of input and output (last plot) variables for the debutanizer column dataset

The output variable shows strong nonlinearity and therefore PLS model fails miserably as shown in Figure 7.16. Let's build an SVR-based soft sensor.

```
# read data
import numpy as np
data = np.loadtxt('debutanizer_data.txt', skiprows=5)

# separate train and test data
from sklearn.model_selection import train_test_split
X, y = data[:,0:-1], data[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_state = 100)

# fit SVR model via grid-search
from sklearn.svm import SVR
```

```

from sklearn.model_selection import GridSearchCV

model = SVR(epsilon=0.05)
param_grid = [{"gamma": np.linspace(1,10,10), 'C': np.linspace(0.01,500,10)}]
gs = GridSearchCV(model, param_grid, scoring='neg_mean_squared_error', cv=10)

gs.fit(X_train, y_train)

```

As shown in Figure 7.16, with a very coarse hyperparameter grid, we are able to obtain a SVR model that provides reasonable accuracy on test data. Difference in training and test accuracies suggest that some overfitting, which may be overcome with more exhaustive hyperparameter search.

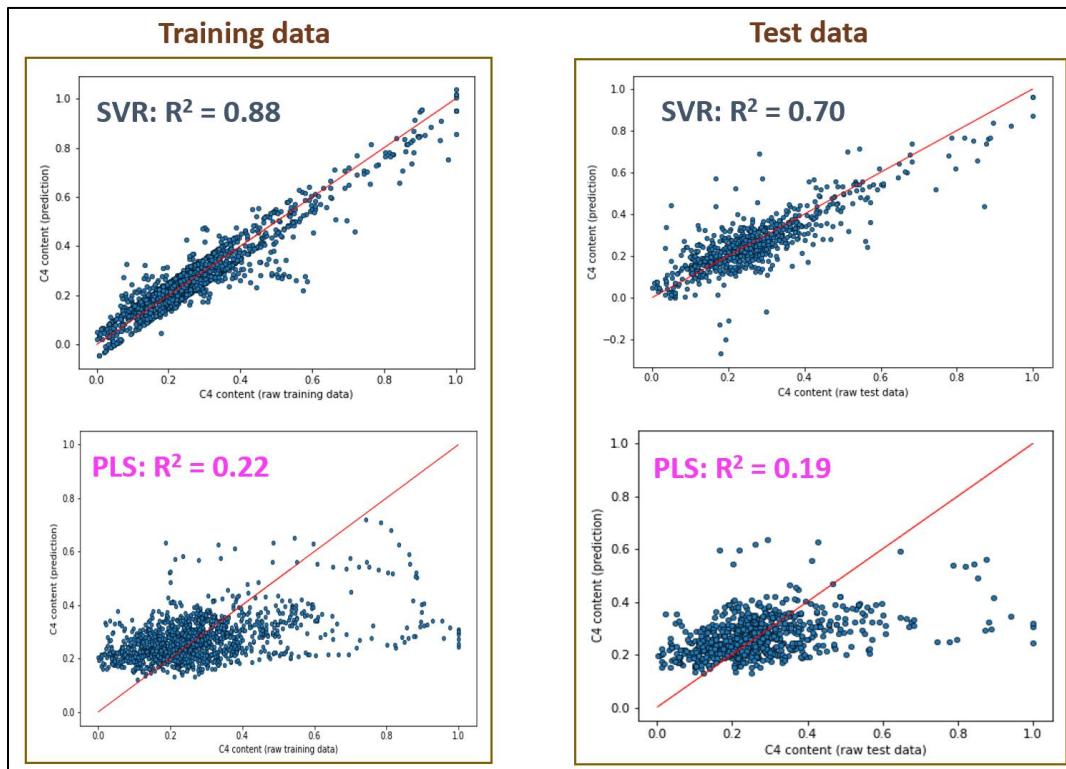


Figure 7.16: SVR and PLS model performance comparison for debutanizer dataset

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 classification and regression. We saw its applications for binary classification, process monitoring, fault detection, and soft sensing. Through kernelized learning, we learned the art of nonlinear modeling. In summary, we have added a powerful tool to your data science toolkit. Next, we will continue building our toolkit and learn how to find clusters/groups in process dataset.

Chapter 8

Finding Groups in Process Data: Clustering & Mixture Modeling

When exploring a dataset, one of the things that you should always do is to check if data shows presence of distinct clusters. Most industrial datasets 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. Whether you are building a soft sensor or 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 results.

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 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

- Introduction to clustering
- Finding groups using classical k-means clustering
- Grouping arbitrarily shaped clusters using DBSCAN
- Probabilistic clustering via Gaussian mixture modeling
- Process monitoring of multimode processes

8.1 Clustering: An Introduction

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

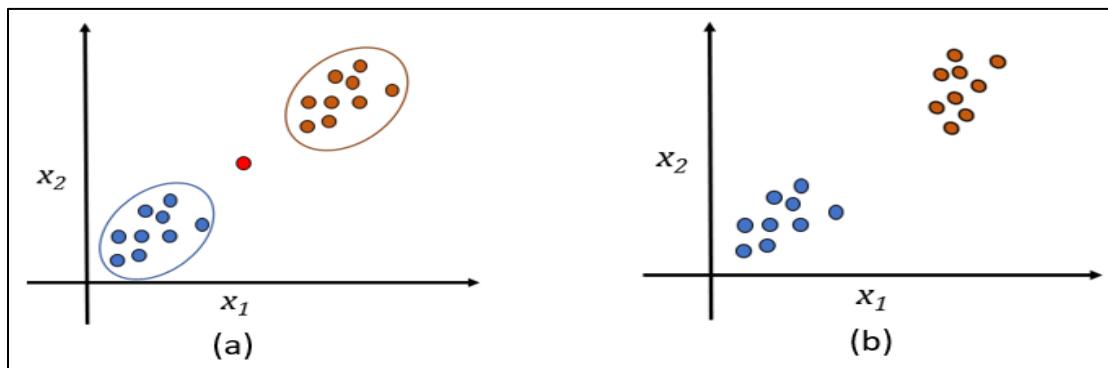


Figure 8.1: Illustrative scenarios for which conventional ML techniques are ill-suited

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

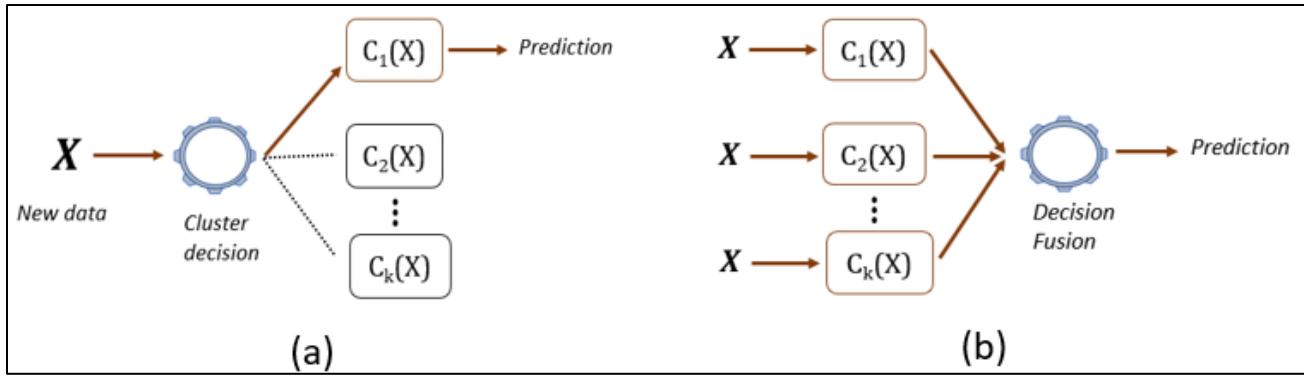


Figure 8.2: ML approaches for multimode processes

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

- *Centroid-based algorithms*: In these algorithms, the similarity between data-points is quantified by the distance of the data-points from the centroid of the clusters. K-Means, Fuzzy C-Means models belong to this category.
- *Distribution-based algorithms*: Here, similarity between data-points is quantified by computing the probabilities of the data-points belonging to the same underlying distribution. Gaussian mixture models (GMMs) are popular in this category.
- *Density-based algorithms*: Density models find areas of high density of data in the multivariable space and assign data-points to different clusters/regions. DBSCAN and OPTICS are popular examples of density models.
- *Connectivity-based algorithms*: In these models, similarity is quantified by directly measuring the distance between the data-points. Hierarchical clustering models belong to this category.



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

Multimode semiconductor manufacturing operation

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

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

# fetch data
import scipy.io

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

variable_names = Etch_data[0,0].variables

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

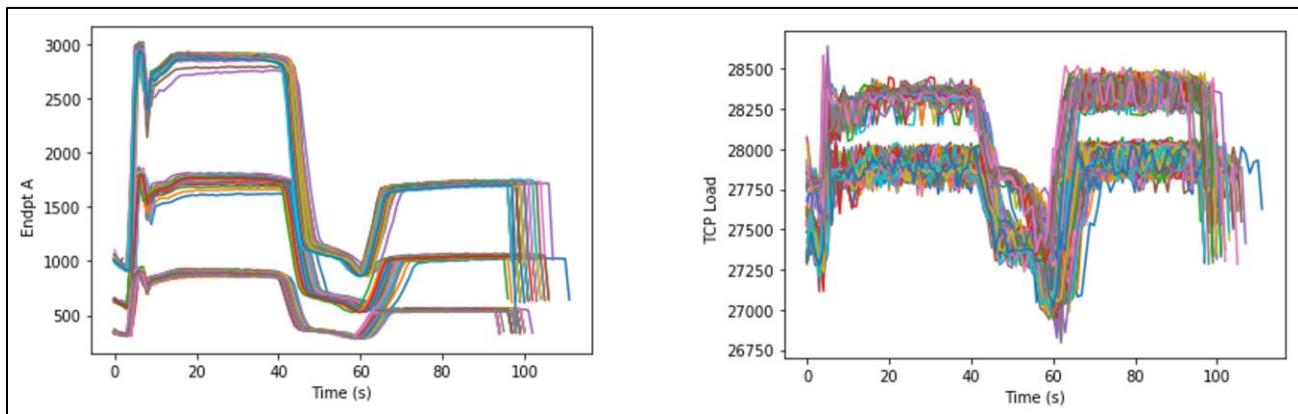


Figure 8.3: Select variable plots for all batches in metal etch dataset. Each colored curve corresponds to a batch.

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

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

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

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

    if calibration_expt.shape[0] < 85:
        continue

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

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

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

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

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

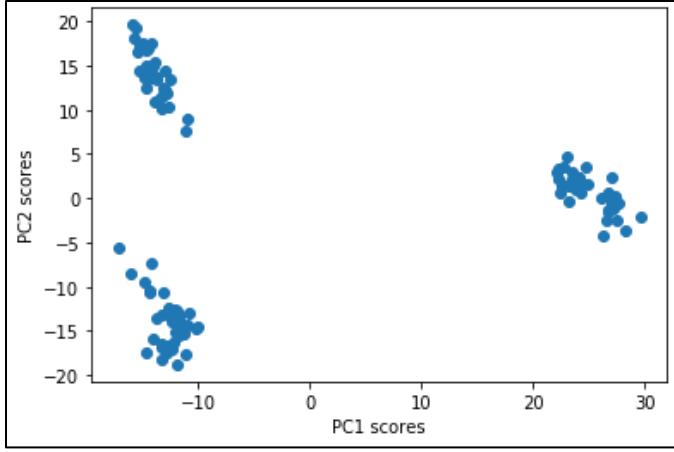


Figure 8.4: Score plot of PC1 and PC2 for calibration batches in metal etch dataset

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

8.2 Centroid-based Clustering: K-Means

K-Means is one of the most popular clustering algorithms due to its simple concept, ease of implementation, and computational efficiency. Let K denote the number of clusters and $\{\mathbf{x}_i\}$, $i = 1, \dots, N$ 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

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

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

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

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

```

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

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

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

```

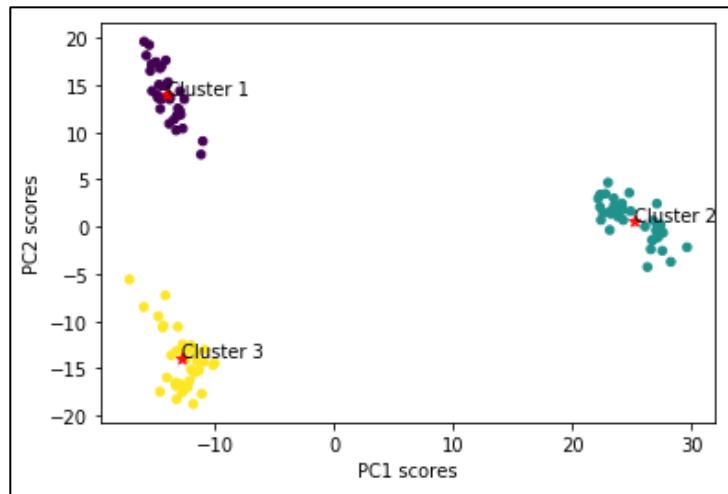


Figure 8.5: Clustering via k-means

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

Determining the number of clusters via elbow method

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

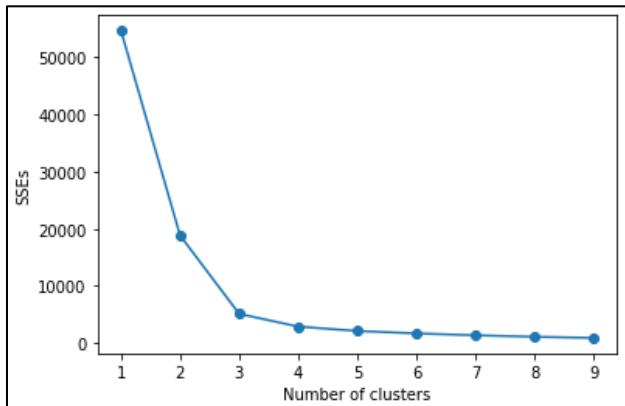


Figure 8.6: Cluster inertias for different number of clusters for metal-etch dataset



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

Silhouette analysis for quantifying clusters quality

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

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

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

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

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

>>> Average silhouette score is : 0.7444602567351603

# silhouette plot
from matplotlib import cm

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

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

```

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

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

```

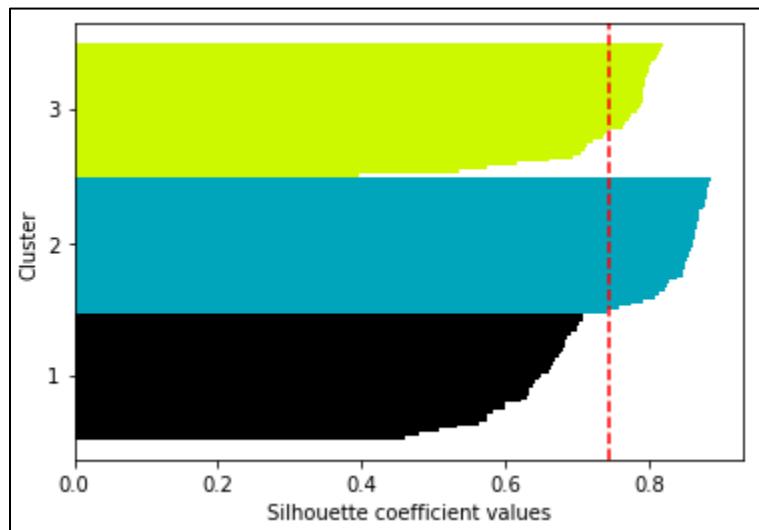


Figure 8.7: Silhouette plot for metal etch data with 3 clusters determined via k-means. Red dashed line denotes the average silhouette value.

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

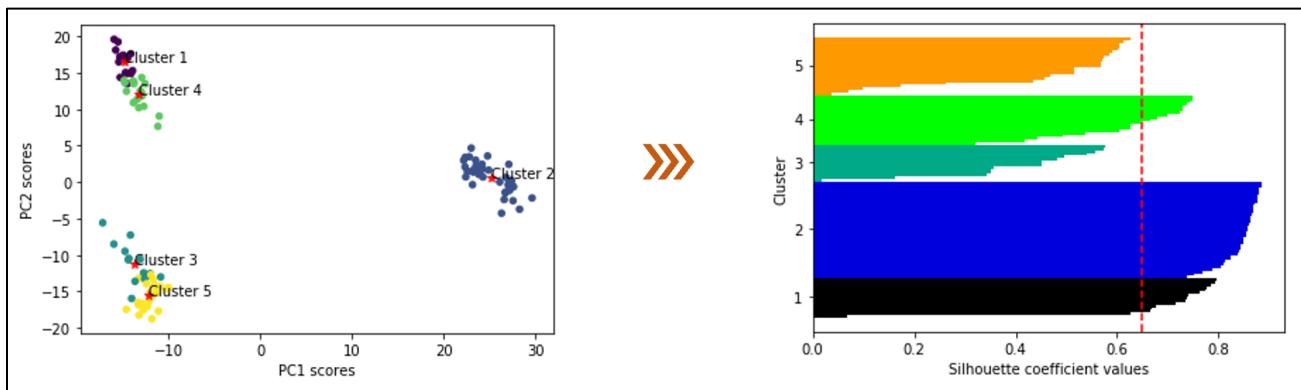


Figure 8.8: Silhouette plot for a sub-optimal clustering example

Pros and cons

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

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

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

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

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

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

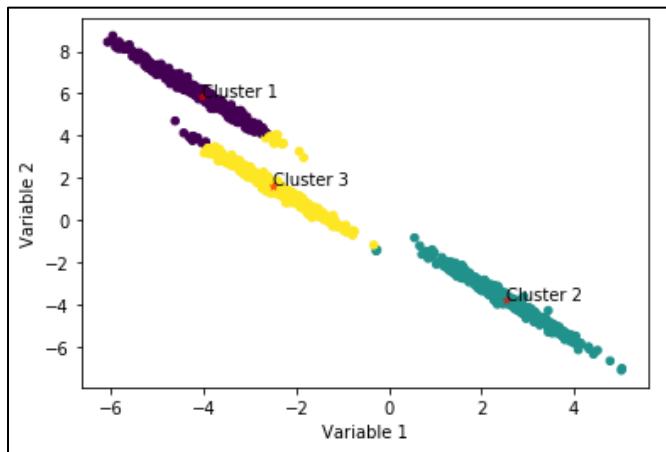


Figure 8.9: k-means clustering failure

8.3 Density-based Clustering: DBSCAN

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

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

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

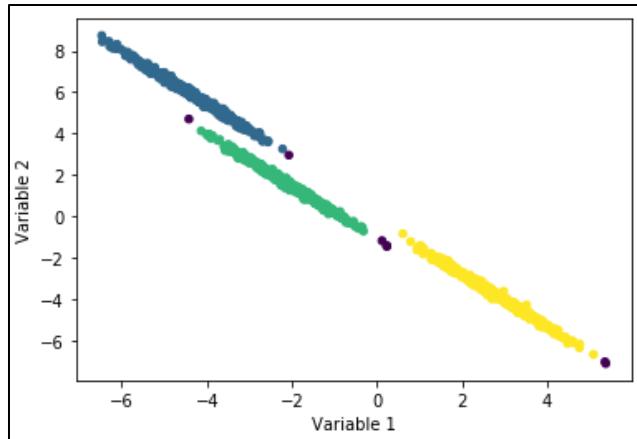


Figure 8.10: DBSCAN succeeds in handling elliptical data distribution

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

```

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

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

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

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

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

```

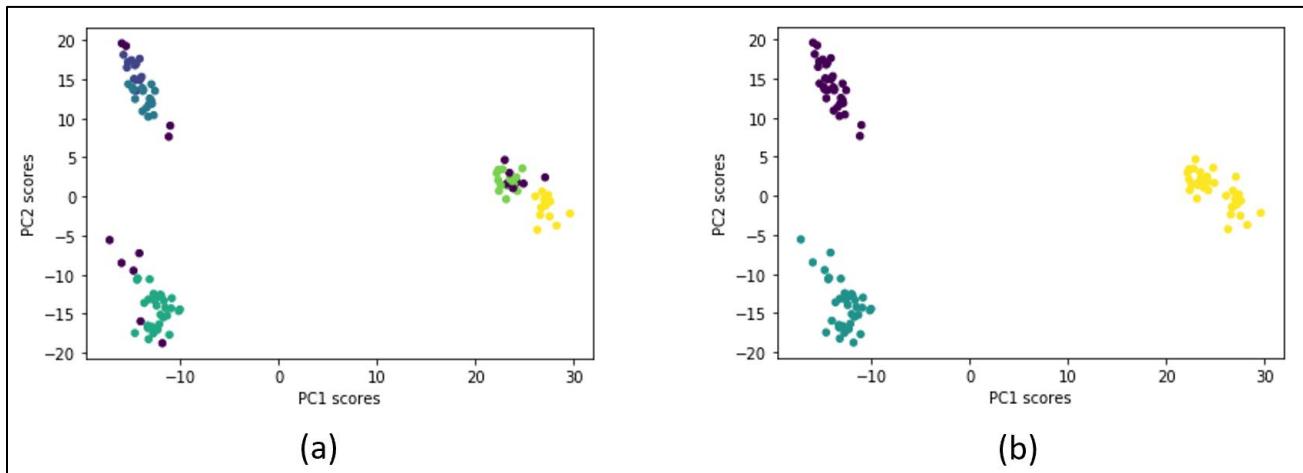


Figure 8.11: Clustering via DBSCAN for metal etch data with (a) improper hyperparameters and (b) proper hyperparameters

Pros and cons

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

8.4 Probabilistic Clustering: Gaussian mixtures

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

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

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

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

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

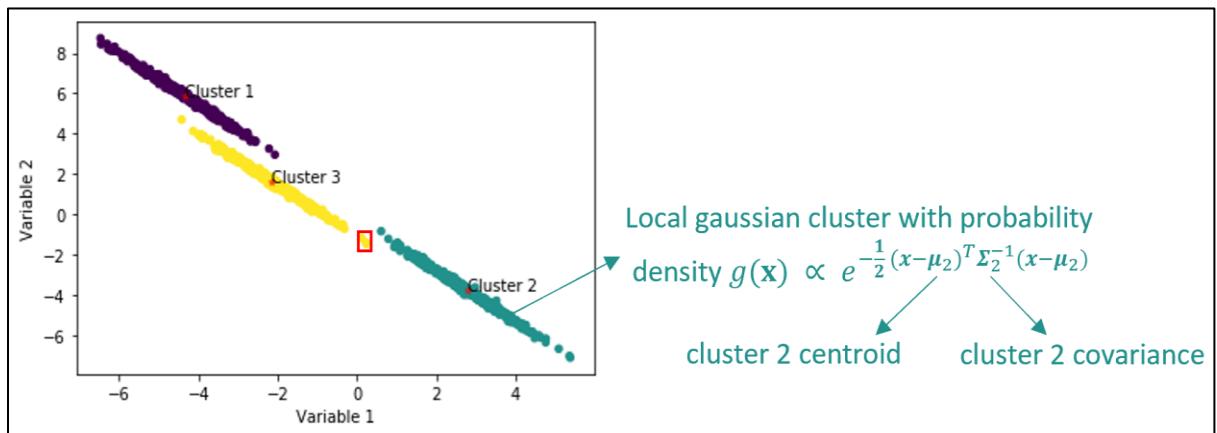


Figure 8.12: GMM based clustering of ellipsoidal data distribution

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

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

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

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

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

Mathematical background

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

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

The overall density at any spatial location is given by

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

where, ω_i represents the prior probability that a new sample comes from the i^{th} Gaussian component and $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_k\}$. The GMM model is constructed by estimating the parameters $\boldsymbol{\theta}_i$, ω_i for all the clusters using the training samples $\mathbf{X} \in \mathbb{R}^{N \times m}$. The parameters are estimated by optimizing the log-likelihood of the training dataset given as below

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

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

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

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

$P^{(s)}(C_i|x_j)$ 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

$$\begin{aligned}\boldsymbol{\mu}_i^{(s+1)} &= \frac{\sum_{j=1}^N P^{(s)}(C_i|x_j) \mathbf{x}_j}{\sum_{j=1}^N P^{(s)}(C_i|x_j)} \\ \boldsymbol{\Sigma}_i^{(s+1)} &= \frac{\sum_{j=1}^N P^{(s)}(C_i|x_j) (\mathbf{x}_j - \boldsymbol{\mu}_i^{(s+1)}) (\mathbf{x}_j - \boldsymbol{\mu}_i^{(s+1)})^T}{\sum_{j=1}^N P^{(s)}(C_i|x_j)} \\ \omega_i^{(s+1)} &= \frac{\sum_{j=1}^N P^{(s)}(C_i|x_j)}{N}\end{aligned}$$

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. While in k-means, we plotted SSEs for different number of clusters, in GMM we will utilize the Bayesian Information Criterion (BIC) metric (introduced in Chapter 3). Unlike SSEs, BICs increase after a while and optimal Gaussian components is the value that minimizes BIC.

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

if BIC < lowestBIC:
    optimal_n_cluster = n_cluster
    lowestBIC = BIC

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

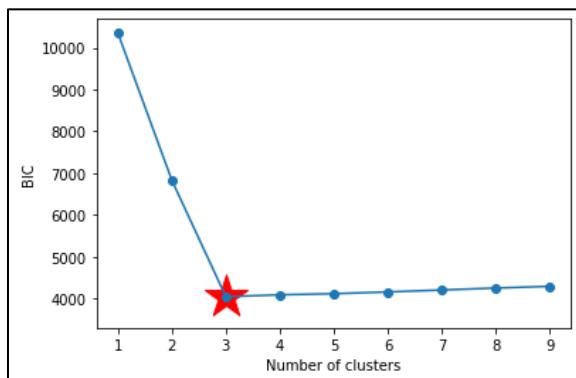


Figure 8.13: BIC plot for determining number of GMM components

There is another method, called F-J algorithm³⁹, which can be used to find the optimal number of GMM components⁴⁰ and model parameters in an integrated manner. Also, the number of components does not need to be specified beforehand in F-J method. Internally, the method initializes with a large number of components and adaptively adjusts this number by

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

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

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

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

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

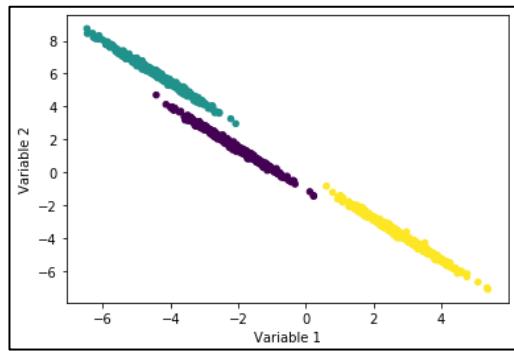


Figure 8.14: GMM based clustering of ellipsoidal data distribution via F-J method

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

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

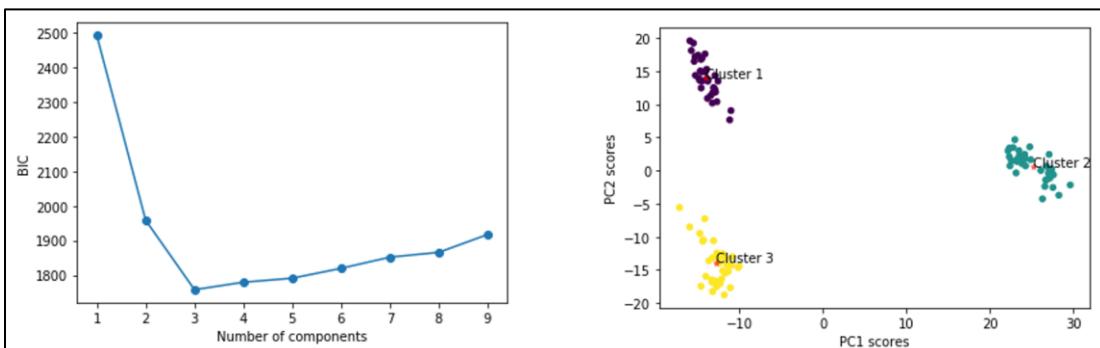


Figure 8.15: BIC plot and GMM clustering of metal etch data

8.5 Multimode Process Monitoring via GMM for Semiconductor Manufacturing Process

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

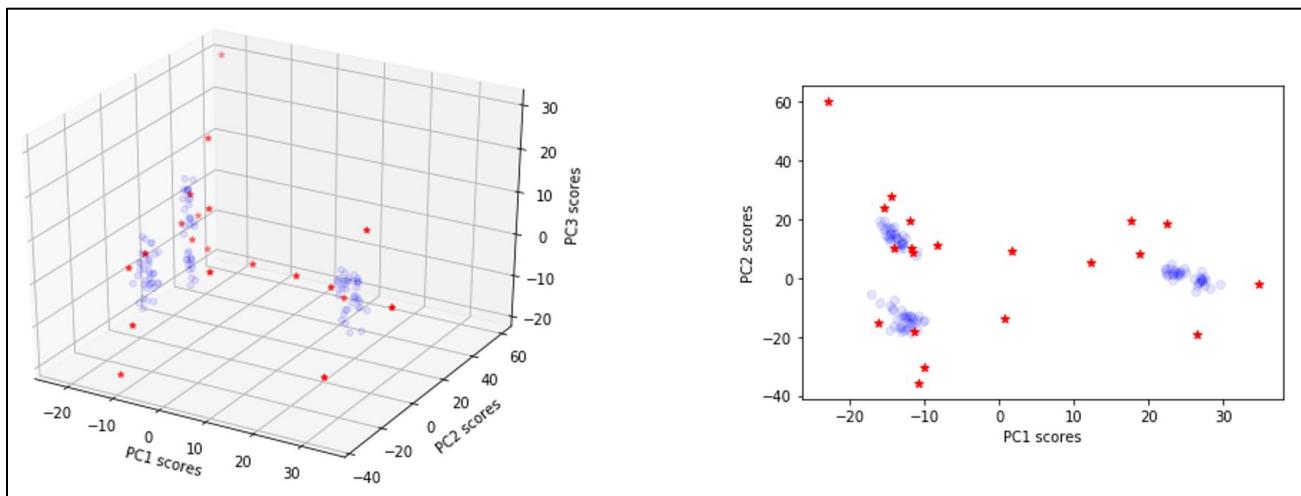


Figure 8.16: Calibration (in blue) and faulty (in red) batches in PCA score space

Process monitoring/Fault detection index

A straightforward approach to detect a faulty batch would be to build separate ellipsoidal boundary or T^2 control chart for each cluster/mode and check if the T^2 metric for the faulty batch falls outside the control limits for all the clusters. However, monitoring many control charts can become tedious when the number of Gaussian components is high. For convenience, the local metrics/charts can be combined to generate a single global metric/chart. We will follow the approach of Xie & Shi⁴¹ (an alternative approach is provided by Yu & Qin³⁹). For a test data point, \mathbf{x}_t , local Mahalanobis distance is computed from each cluster

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

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

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

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

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

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

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

    for component in range(3):
        Dlocal = np.dot(np.dot((x-gmm.means_[component,:,:np.newaxis]).T,
                               np.linalg.inv(gmm.covariances_[component,:,:])),
                       (x-gmm.means_[component,:,:np.newaxis]))
        Dglobal_train[i] = Dglobal_train[i] + probs[0,component]*Dlocal

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

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

Figure 8.17 shows the control chart for the training data.

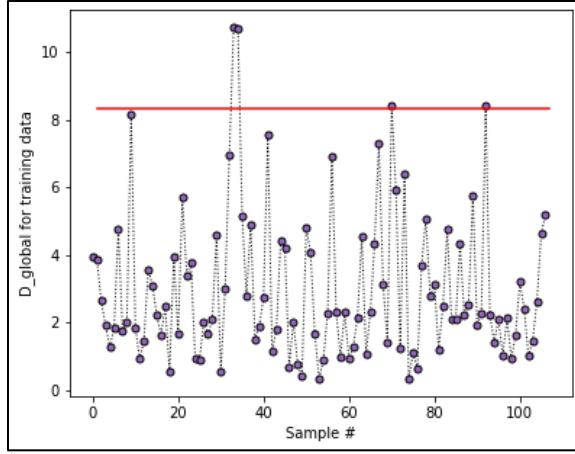


Figure 8.17: Global monitoring chart for metal etch calibration data

Fault detection for test data

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

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

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

    if test_expt.shape[0] < 85:
        continue

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

unfolded_TestdataMatrix = unfolded_TestdataMatrix[1,:,:]

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

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

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

```

probs = gmm.predict_proba(x.T)

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

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

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

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

```

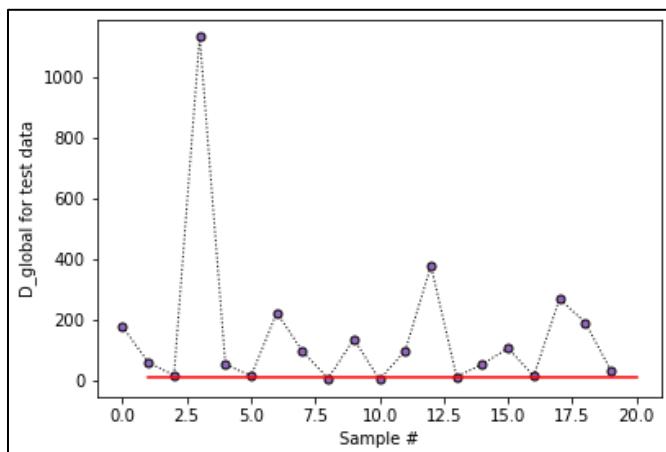


Figure 8.18: Global monitoring chart for test data

Summary

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

Chapter 9

Decision Trees & Ensemble Learning

Imagine that you are in a situation where even after your best attempts your model could not provide satisfactory performance (due to high bias or variance). 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
 - Soft sensing application of random forests in concrete construction industry
 - Introduction to ensemble learning techniques (bagging, Adaboost, gradient boosting)
 - Effluent quality prediction using XGBoost in wastewater treatment plant
-

9.1 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 9.1 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.

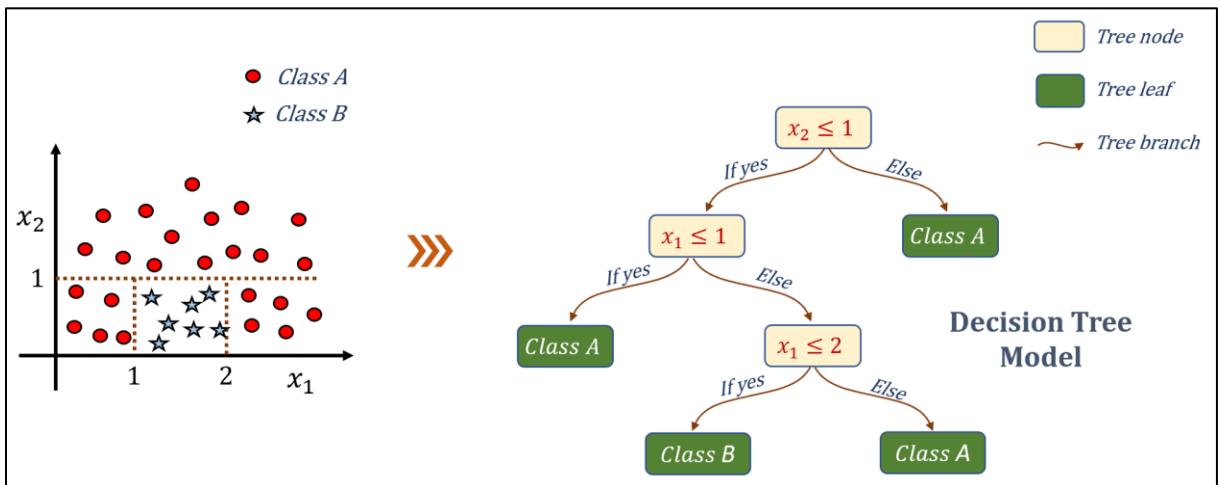


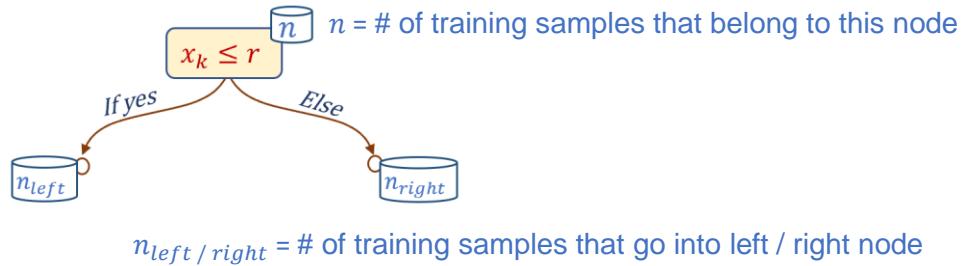
Figure 9.1: A decision tree with constant model used for binary classification in a 2D space

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 node⁴² 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

$$I = 1 - \sum_{q=1}^2 p_q^2 \quad \text{eq. 1}$$

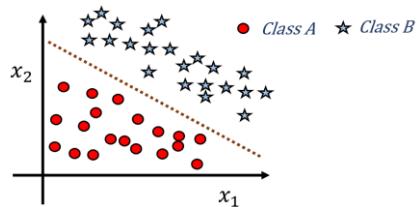
where p_q is the ratio of samples of the corresponding data subset belonging to class q . For example, if all the samples in a subset belong to class 1, then $p_1 = 1$ and $I = 0$. Therefore, if CART could find k and r such that the n samples get perfectly divided class-wise into the left and right subsets, 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 .

⁴² 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 package⁴³ called 'linear-tree' that allows building model trees with linear models at the leaves.

⁴³ <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 9.2a plots the predicted vs actual output values. It is apparent that with no restriction on tree depth overfitting has occurred. Subplot 9.2b 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)
```

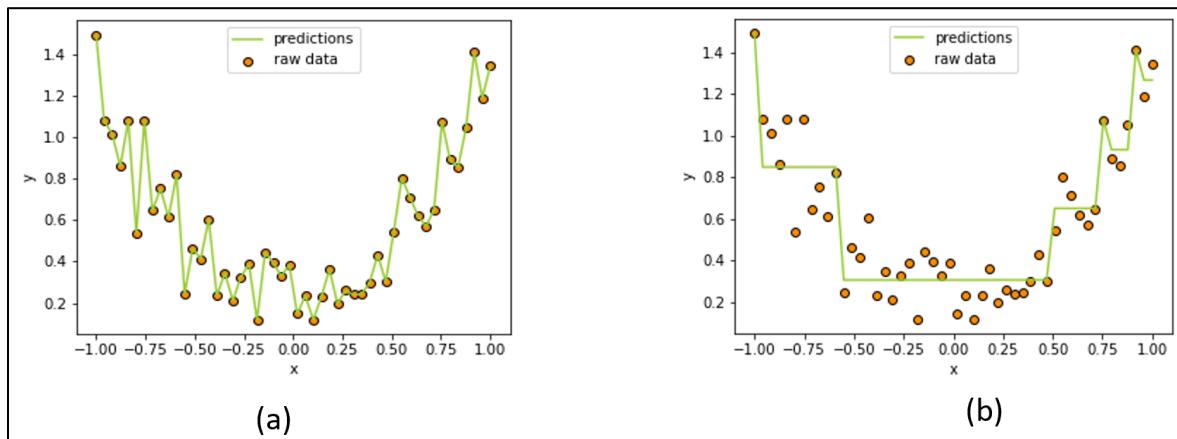
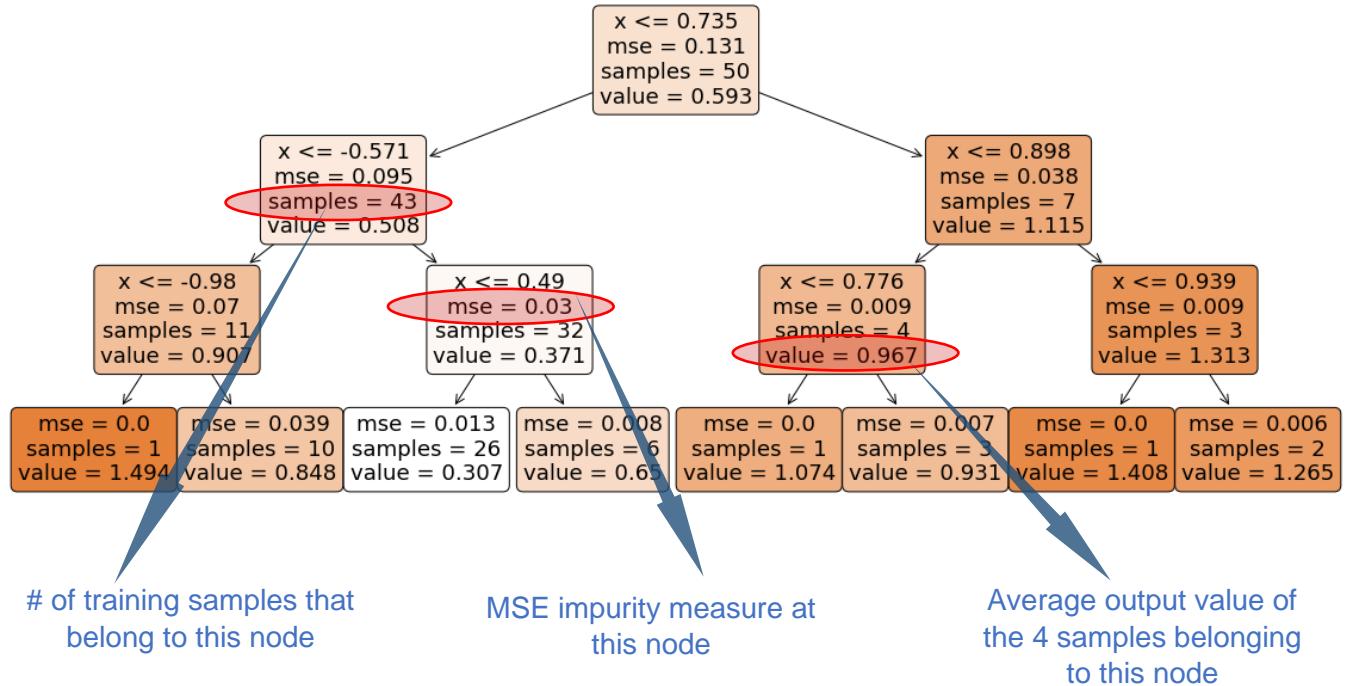


Figure 9.2: Decision tree regression predictions using unregularized and regularized models.

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 9.2, 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.

9.2 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 9.3 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).

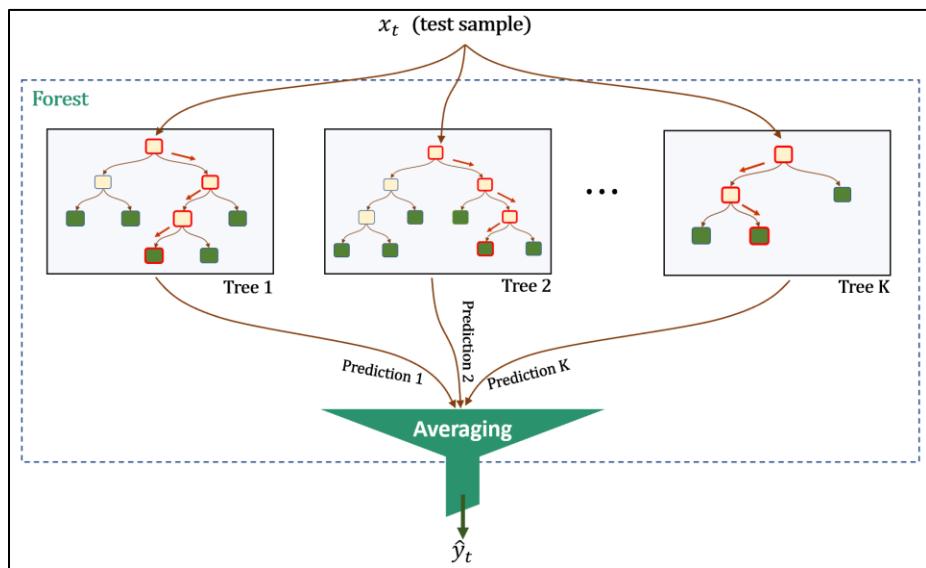


Figure 9.3: A random forest prediction is combination of predictions from multiple decision trees. [For a classification problem, Sklearn returns the class corresponding to the highest average class probability]

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 RF training 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 $N=10$.

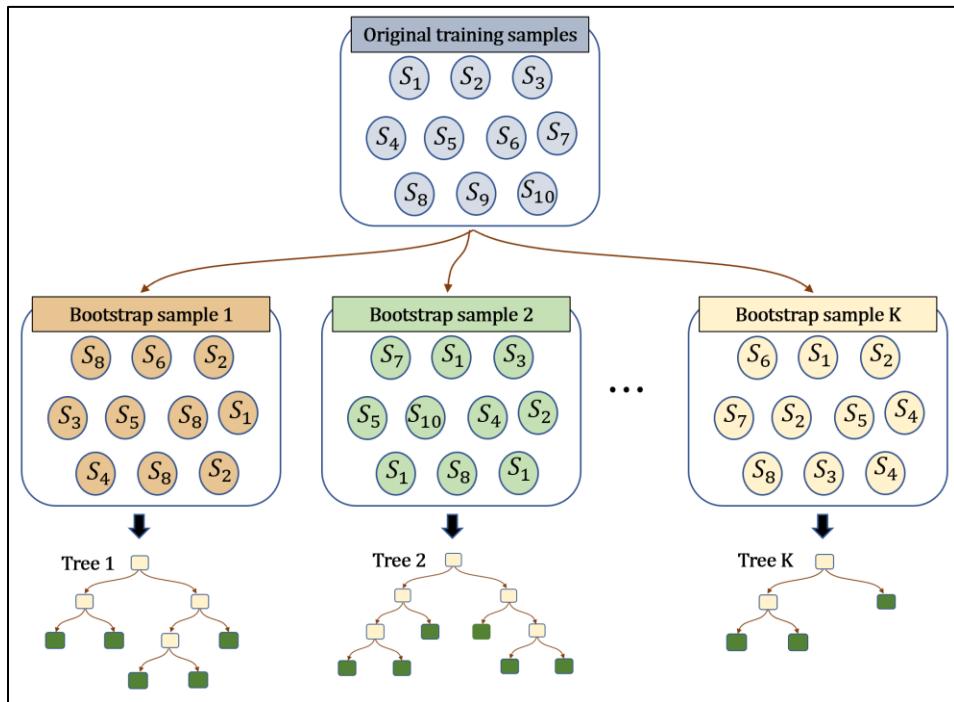


Figure 9.4 Creation of separate DT models using bootstrap samples. S_i denotes the i^{th} training sample.

- 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 $M=3$.

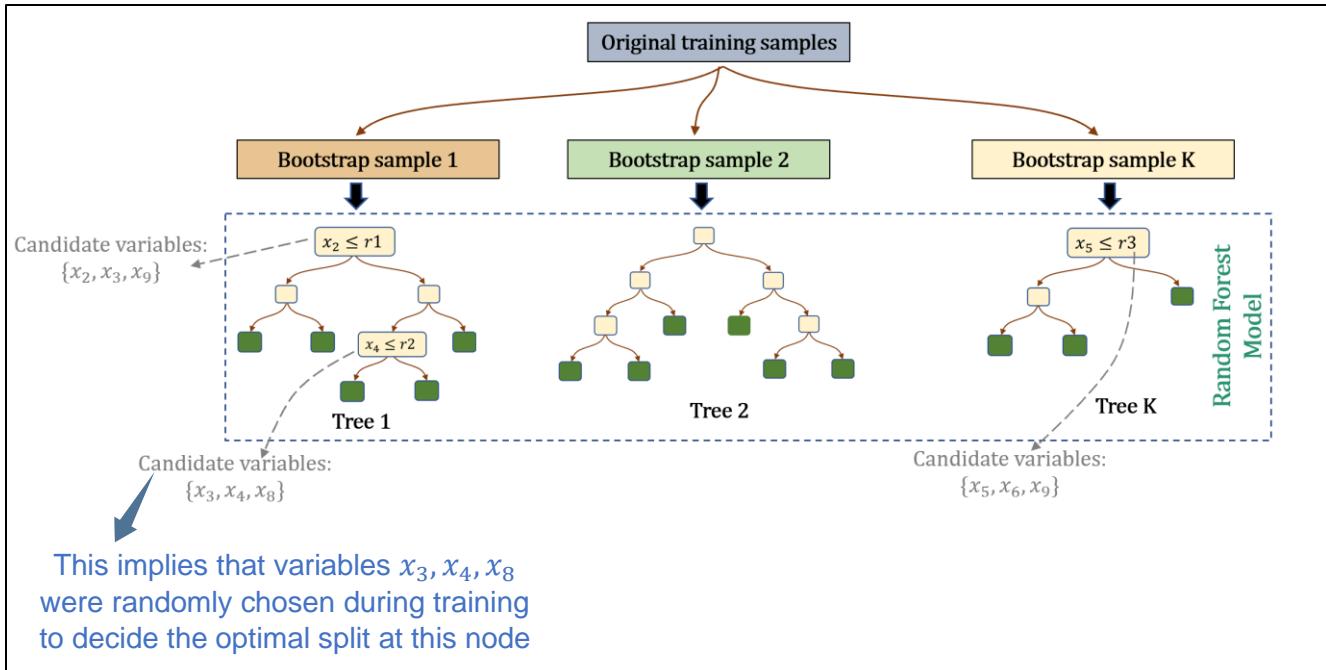


Figure 9.5: Illustration of RF training mechanism

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 9.2. Figure 9.6 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)
```

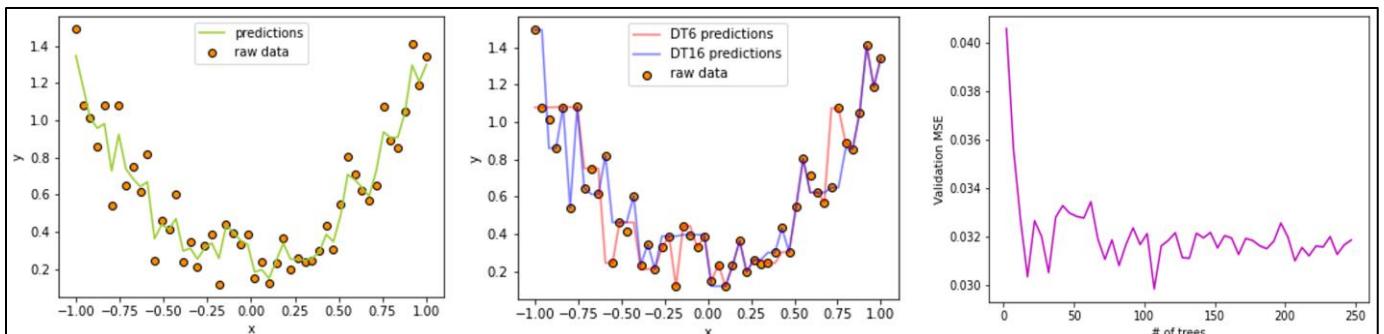


Figure 9.6: (Left) Random forest regression predictions, (middle) predictions from a couple of constituent decision trees, (right) Impact of number of trees in RF model on validation error

Figure 9.6 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.

9.3 Soft Sensing via Random Forest in Concrete Construction Industry

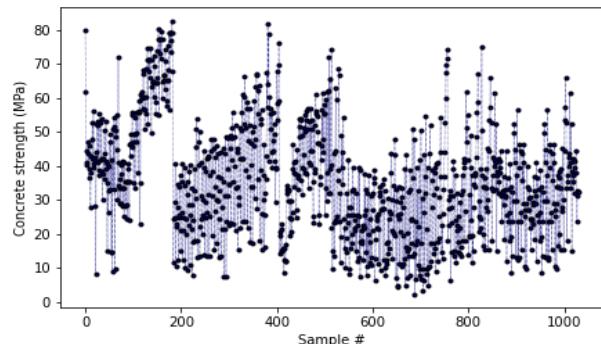
Let us see an application of RF for prediction of compressive strength of concrete as a function of its constituent ingredients and concrete age. The dataset contains 1030 concrete samples and has data on the following

Inputs

- Cement: kg/m³ mixture
- Blast furnace slag: kg/m³ mixture
- Fly ash: kg/m³ mixture
- Water: kg/m³ mixture
- Superplasticizer: kg/m³ mixture
- Coarse aggregate: kg/m³ mixture
- Fine aggregate: kg/m³ mixture
- Age: Day (1 to 365)

Output

- Concrete compressive strength: MPa



We will keep 33% of data aside for testing with the number of trees set to a large value 200 and M to 3.

```
# read data
Import numpy as np
data = np.loadtxt('cement_strength.txt', delimiter=',', skiprows=1)
X = data[:,0:-1]
y = data[:,-1]

# separate training and test data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_state = 100)

# fit RF model
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(n_estimators=200, max_features=3, oob_score=True,
random_state=1).fit(X_train, y_train)
```

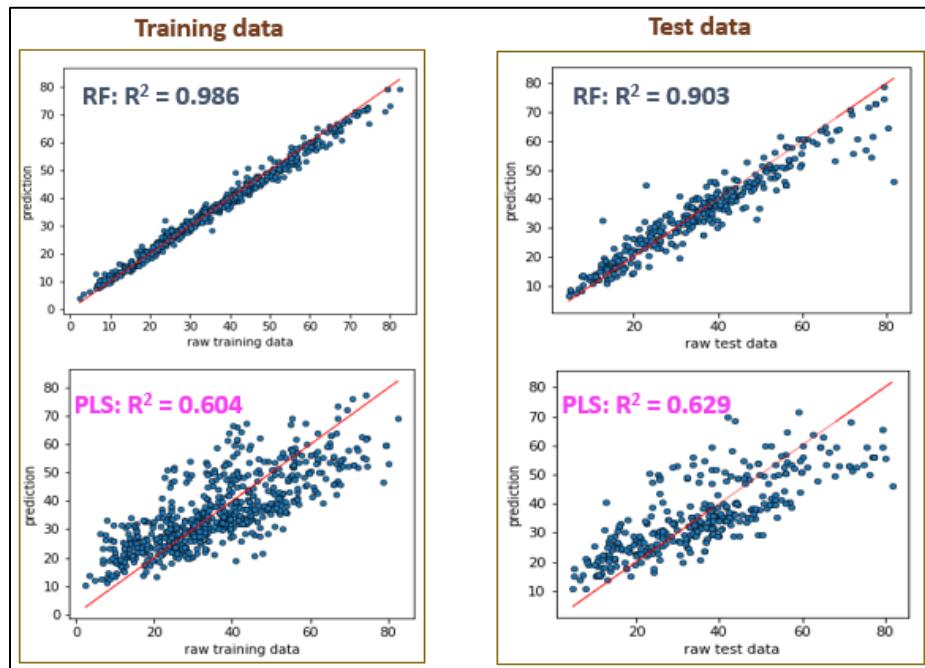


Figure 9.7: RF and PLS predictions for concrete compressive strength dataset. The red line denotes the ideal $y_{\text{prediction}} = y_{\text{measured}}$ reference

Two noteworthy mentions here. First, we didn't scale the variables before fitting. It's not a mistake! RFs (and DTs) are one of the rare ML methods whose solutions are invariant to data-scaling. Second, the superior performance of RF over PLS shows the nonlinear modeling capabilities of RF.

OOB accuracy

In previous illustration, we specified `oob_score=True` during RF training. This specification instructs RF to compute a generalization score using training data during model fitting. This almost sounds like an incorrect statement, but it makes sense when you realize that any single bootstrap dataset does not contain all the samples of the original dataset⁴⁴ (as seen in Figure 9.4). Samples that do not belong to the bootstrap dataset of a tree are called out-of-bag (OOB) samples for that tree. Any given sample would be part of OOB samples of several trees of the forest (let's say DT1, DT5, DT7). A prediction can be made for this sample using the ensemble of DT1, DT5, DT7 and an OOB error can be computed. The OOB errors for all the samples are used to compute an overall `oob_score`.

```
# print OOB score
print('OOB score: ', model.oob_score_)
>>> OOB score: 0.903
```

The `oob_score` for the concrete dataset is same (up to 3 decimal places) as the test score! An OOB score may not be the perfect generalization assessment, but it comes in handy for small-sized dataset where keeping aside a test dataset may not be a luxury.

Feature importances

Let's now look at another handy feature of RF that allows us to rank the model's input/feature variables in terms of their importances for output predictions. Sklearn computes feature importances by computing the average reduction in node impurities (or MSE for regression) upon usage of a feature for node splitting. First, importance of a feature j in a tree is calculated as follows

$$FI_j^{tree} = \frac{\sum_{\text{nodes; node splitting on feature } j} \Delta I_{node}^{tree}}{\sum_{\text{nodes}} \Delta I_{node}^{tree}}$$

The average of FI_j^{tree} over all the trees in the forest gives the desired feature importances

$$FI_j = \frac{\sum_{\text{trees}} FI_j^{tree}}{\text{number of trees}}$$

For our concrete strength dataset, FI_s can be obtained as follows

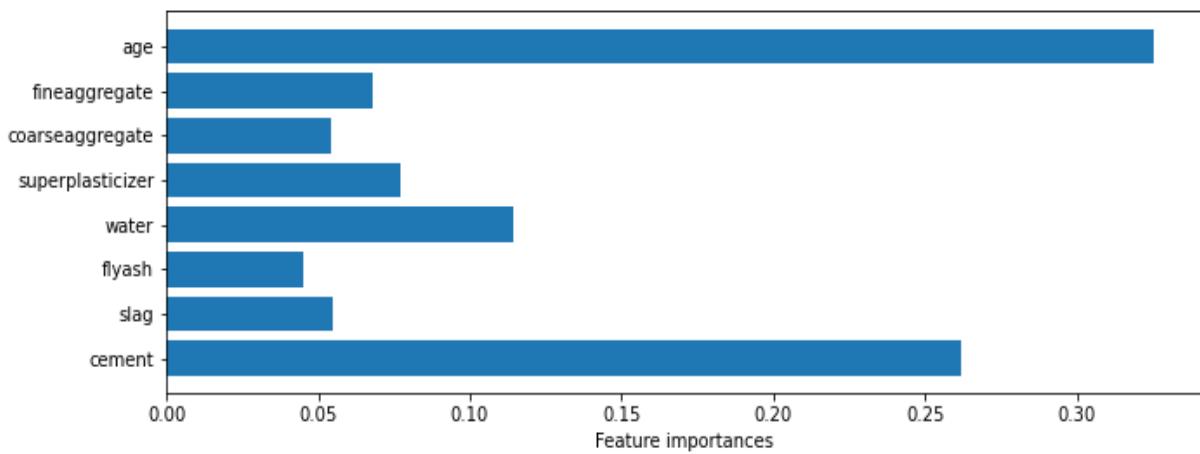
⁴⁴ For large N, it can be shown that any bootstrap dataset in RF theoretically contains approximately 63% of the samples from the original dataset

```

# feature importances
var_names = ['cement','slag','flyash','water','superplasticizer','coarseaggregate','fineaggregate','age']
importances = model.feature_importances_

plt.barh(var_names, importances)
plt.xlabel('Feature importances')

```



Above results show that cement content and age of concrete are the two most important variables for prediction of concrete strength. This result is along the expected lines and further highlights the usefulness of RFs for analyzing nonlinear dependencies in process data.

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.

9.4 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 9.8 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.

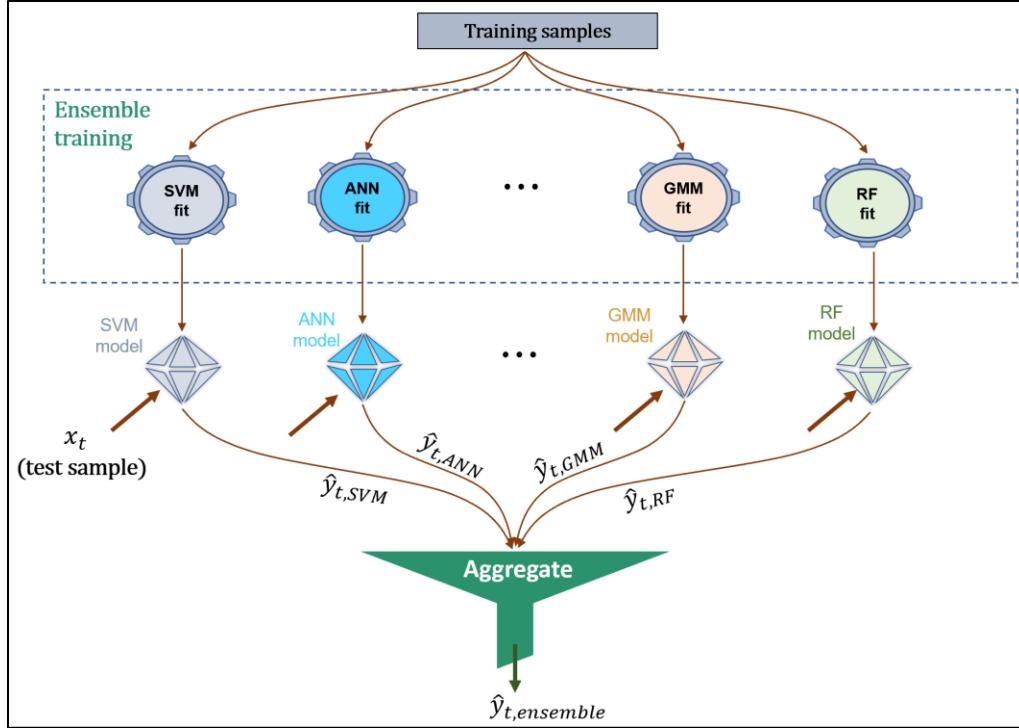
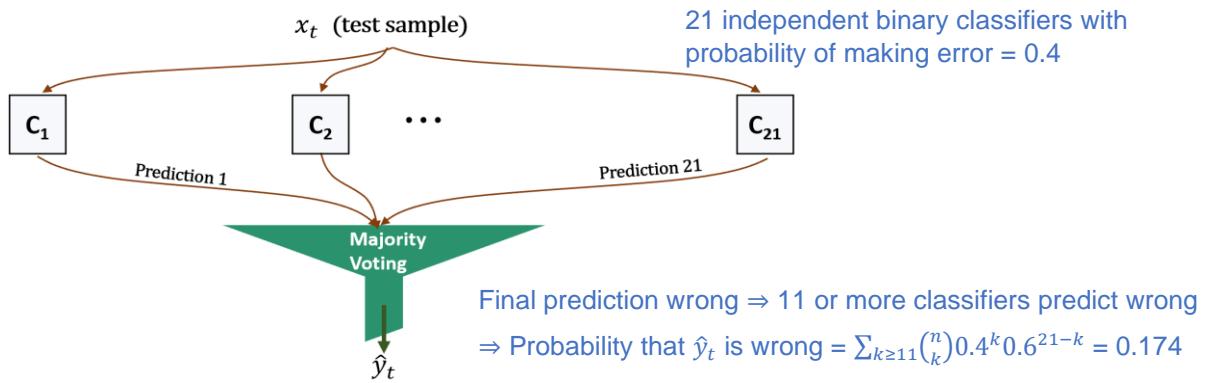


Figure 9.8: Heterogeneous ensemble learning scheme⁴⁵. RF, which itself is an ensemble method, is used as a base model here.

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, 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.

⁴⁵ Another popular heterogeneous ensemble technique is stacking where base models' predictions serve as inputs to another meta-model which is trained separately



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 9.8, 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.

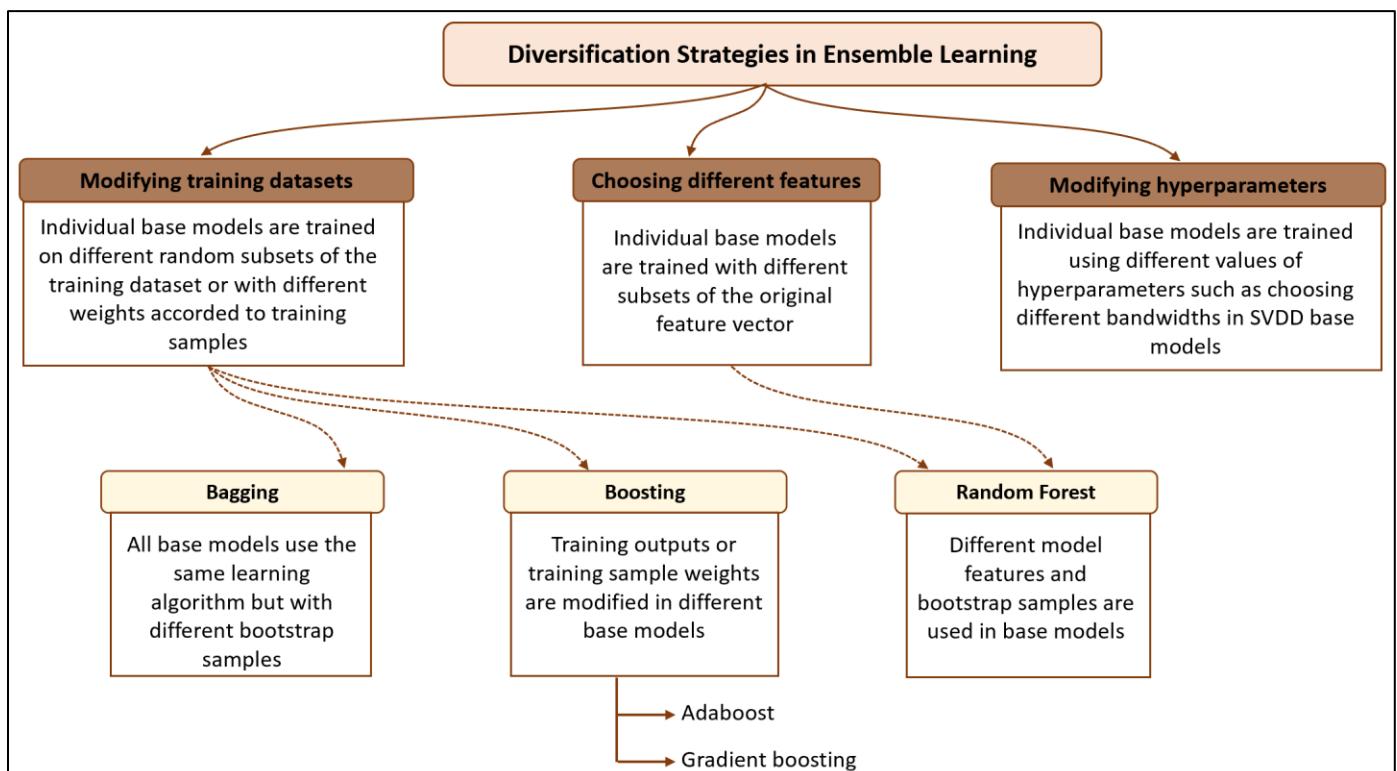


Figure 9.9: An overview of popular ensemble modeling techniques with homogeneous base models



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.

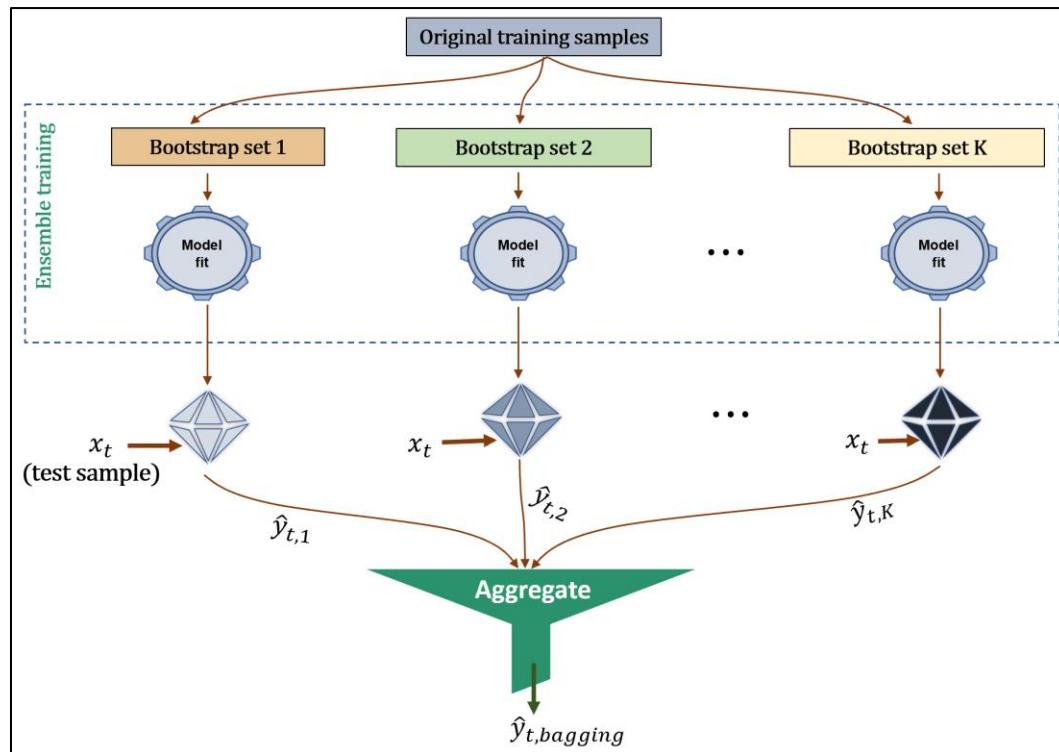


Figure 9.10: Bagging ensemble technique. Note that if you have multiple independently sampled original training datasets, then bootstrapping is not needed.

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
```

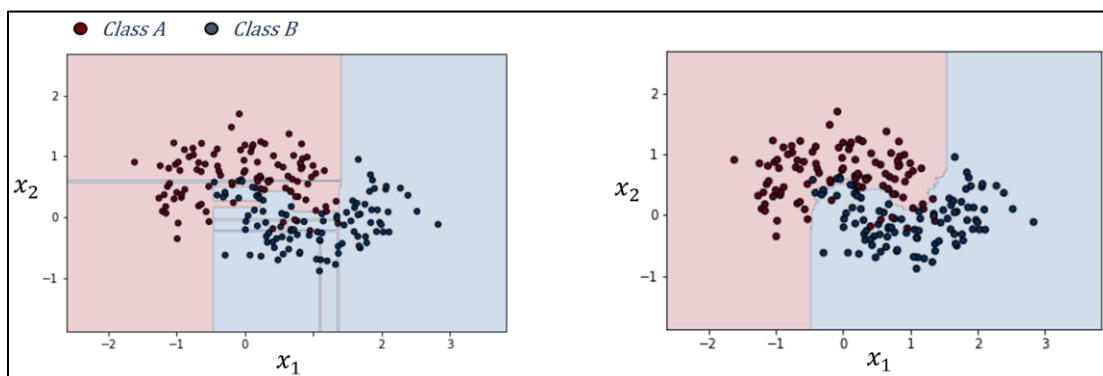


Figure 9.11: Classification boundaries obtained using (left) fully grown decision tree and (right) bagging with fully grown decision trees

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 bias⁴⁶ than the base models.

⁴⁶ 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 9.12, 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).

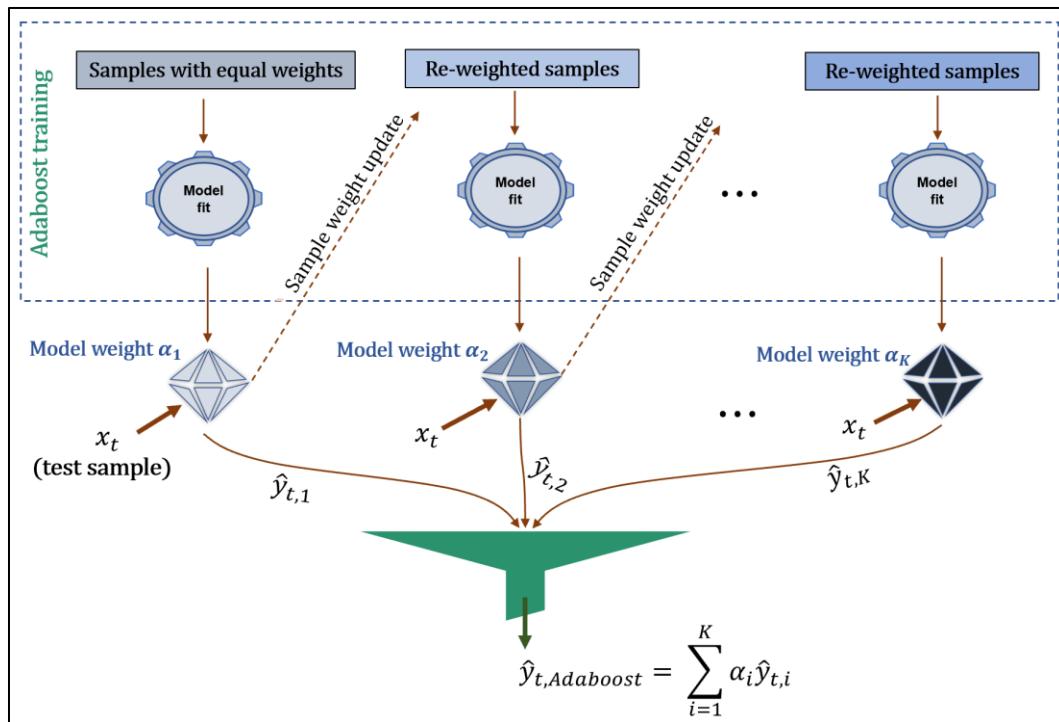


Figure 9.12: Adaboost ensemble scheme

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 9.13 shows the algorithm's scheme.

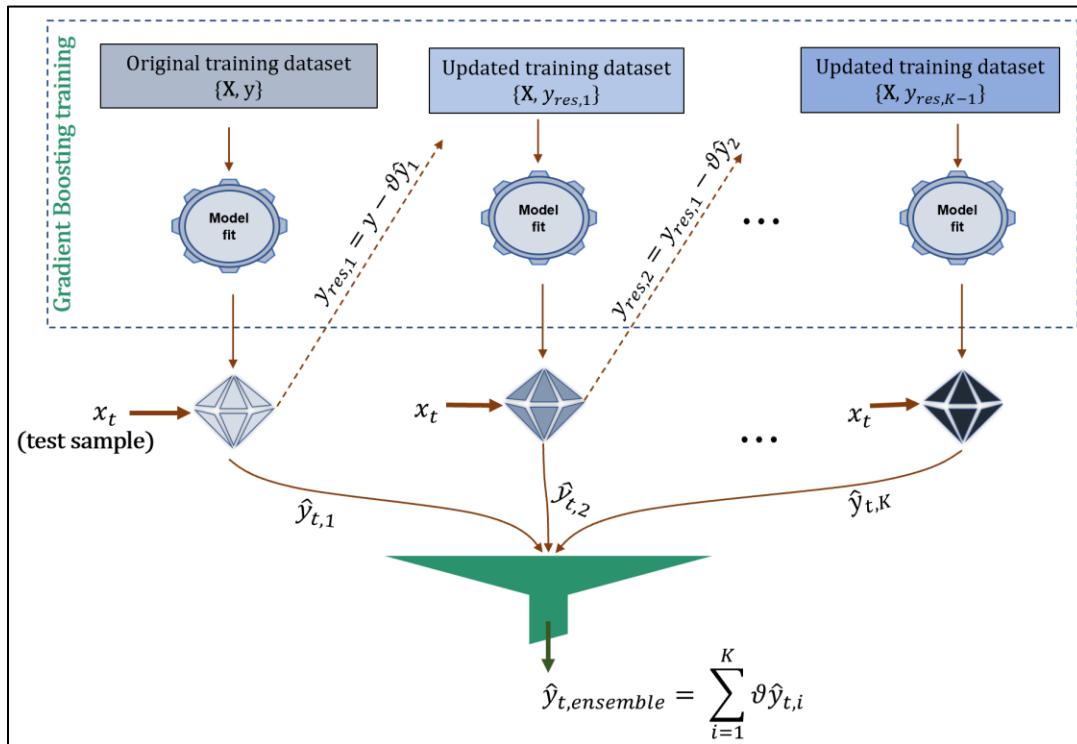


Figure 9.13: Gradient Boosting ensemble scheme. Here, \hat{y}_i denote prediction for training samples from the i^{th} base model. Note that the hyperparameter ϑ need not be constant (as used here) for the different base models.

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 problems, 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).

9.5 Effluent Quality Prediction in Wastewater Treatment Plant via XGBoost

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 see an application of XGBoost for a soft-sensing problem. The dataset comes from an activated sludge process which treats wastewater from domestic and industrial sewers. Data for 38 variables over each of the 527 days of operation are provided. 7 out of the 38 variables characterize the process output or effluent quality of the plant. In this case study, we will create a predictive model for one of the 7 output variables, output conductivity (COND-S) of the output stream. First 22 measurements taken at different stages of the process are used as model inputs (detailed process description is in Appendix). Our XGBoost model will act as a soft sensor to predict the output.

Let's quickly explore the dataset.

```
# read data
import pandas as pd
data_raw = pd.read_csv('water-treatment.data', header=None, na_values="?" )
X_raw = data_raw.iloc[:,1:23]
y_raw = data_raw.iloc[:,29]
data = pd.concat([X_raw, y_raw], axis=1)

# check for presence of missing values
print(data.info())

RangeIndex: 527 entries, 0 to 526
Data columns (total 23 columns):
 #   Column  Non-Null Count  Dtype  
 ---  -- 
 0    1        509 non-null   float64
 1    2        524 non-null   float64
 2    3        527 non-null   float64
 3    4        521 non-null   float64
```

We notice that there are missing values in different columns of the dataset. We will adopt the naïve approach of removing rows with any missing data.

```

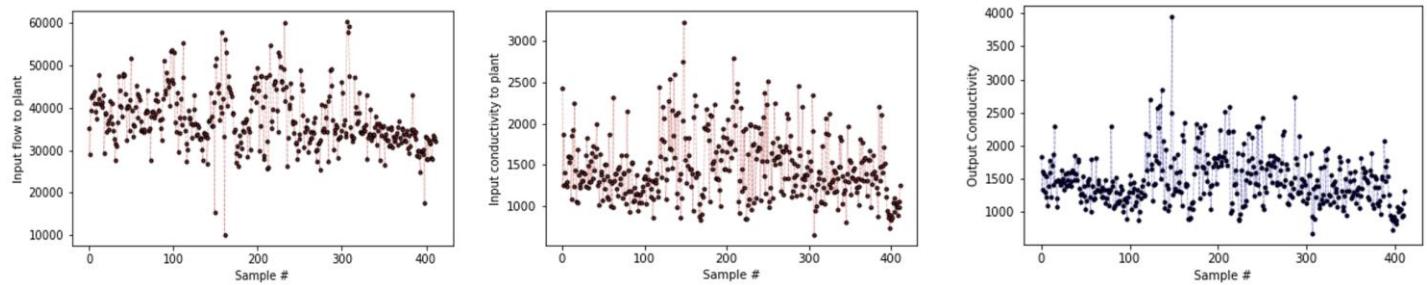
# remove rows with missing data
data.dropna(axis=0, how='any', inplace=True)

print('Number of samples remaining:', data.shape[0])
>>> Number of samples remaining: 412

# separate inputs and output
X = data.iloc[:, :-1]
y = data.iloc[:, -1]

```

Figure below shows the data for a couple of inputs and the output. There seems to be adequate variability in the dataset. For model training and assessment, we will split the dataset into fitting, validation, and test datasets.



```

# separate fitting, validation, 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)
X_fit, X_val, y_fit, y_val = train_test_split(X_train, y_train, test_size=0.3, random_state=100)

# fit XGBoost model
import xgboost
model = xgboost.XGBRegressor(max_depth=3, learning_rate=0.1, random_state=100)
model.fit(X_fit, y_fit, eval_set=[(X_val, y_val)], early_stopping_rounds=2)

```

Because shallower trees are preferred with boosting, `max_depth` parameter was used to restrict tree depth to 3. If desired, this hyperparameter can be tuned via grid search. The `early_stopping_rounds` parameter is used to instruct XGBoost to stop iterating (or adding more base models) if validation accuracy does not improve in any round of 2 consecutive iterations. During model fitting, validation accuracy at each iteration is reported as shown below. Here, `rmse` is used as evaluation metric; any other metric can be specified through `eval_metric` parameter in the `fit` method.

```

[0] validation_0-rmse:1374.03943
[1] validation_0-rmse:1242.94312
[2] validation_0-rmse:1123.29248
[3] validation_0-rmse:1017.08002
:
[45] validation_0-rmse:115.25280
[46] validation_0-rmse:115.51093
[47] validation_0-rmse:115.25786
[48]

```

XGBoost has several other settings that can be used to tune the model. You can check them out in the official documentation⁴⁷. Figure 9.14 shows the XGBoost regressor model predictions. Model accuracies indicate good model performance without much overfitting.

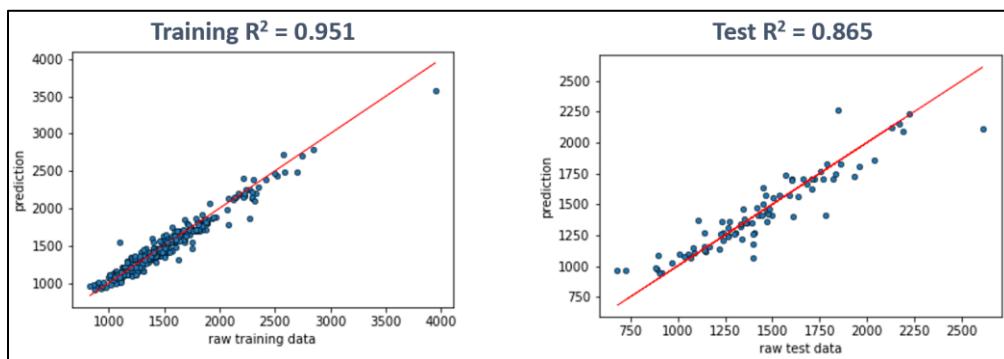


Figure 9.14: XGBoost Model predictions on wastewater treatment plant dataset.

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 XGBoost (a popular implementation of gradient boosting trees) for prediction of effluent quality in a wastewater treatment plant. This chapter has added several powerful techniques to your ML arsenal and you will find yourself using these quite often.

⁴⁷ https://xgboost.readthedocs.io/en/stable/python/python_api.html#module-xgboost.sklearn

Chapter 10

Other Useful Classical ML Techniques

In this last chapter on classical ML methods, we will study two popular non-parametric techniques, namely, KDE and k-NN. These methods are conceptually and mathematically simple but have proven to be very useful in practice. While KDE allows us to compute probability density functions using process data, k-NN facilitates characterizing new data sample using (geometrically) neighboring samples. You might recollect that KDE was often mentioned in the earlier chapters as the technique of choice to estimate monitoring metrics thresholds for complex process systems; we will see how to do that in this chapter. In multivariate settings, KDE is also employed for outlier detection. The most common usage of k-NN is classification, but several other interesting adaptations of traditional k-NN have also been reported and we will study one such application of k-NN for fault detection.

Most of the illustrations in this book till now have involved employment of a single ML technique to solve a given problem. However, it is not necessary to use different ML algorithms in silos. Combining different ML methods (such as PLS with PCA, or ANN with PLS) can sometimes provide better performance. Before we end this part of the book on classical ML techniques, we will look at several examples involving smart combinations of ML techniques.

Specifically, we will cover the following topics in this chapter

- Introduction to KDE
- Finding process monitoring control limits using KDE
- Introduction to k-NN
- Fault detection using k-NN
- Combination of techniques as a useful strategy

10.1 KDE: An Introduction

Kernel density estimation (KDE) is a technique to estimate (probability) density of a variable or joint probability density of a group of variables from data without any prior knowledge about the form of the underlying distribution. Figure 10.1 shows a 1D example where the estimated KDE density curve is superimposed on the histogram built using observed data. As is evident, KDE is a robust way of estimating probability densities. Using the KDE curve, we can estimate the likelihood of new variable measurement falling in any specified range.

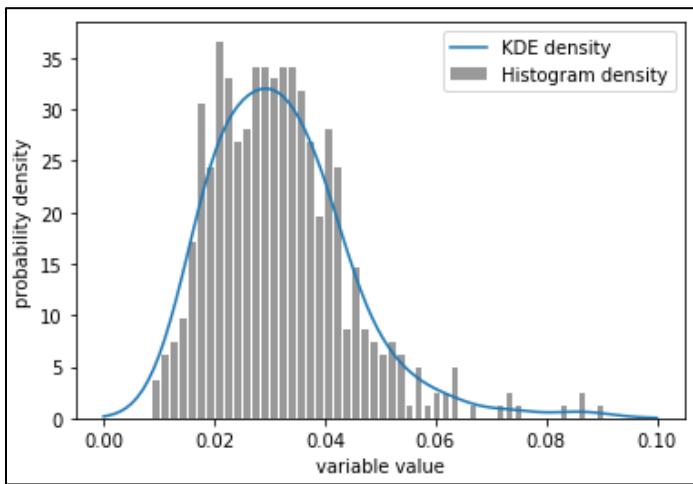


Figure 10.1: Probability density via histogram and KDE

To see one of the utilities of KDE, recollect the two ways we have employed to estimate the threshold of a process monitoring metric till now. We computed the percentile of either the samples (for ICA) or an assumed distribution (such as a F-distribution for PCA T^2 metric). While the first approach is inappropriate when number of samples is low, the second approach is impractical for complex systems (or systems where process variables do not follow gaussian distribution) where the metric distribution is not known beforehand. In these scenarios, KDE can be employed for control limit estimation.

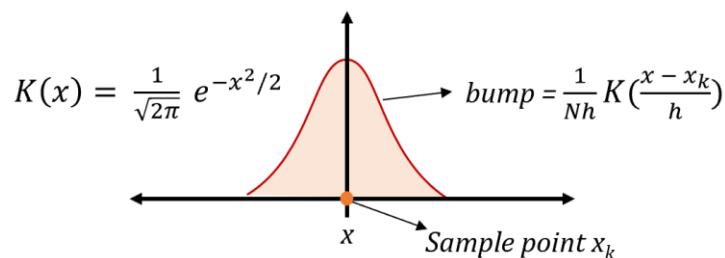
One can use histogram as well for estimating probability densities. However, as seen in Figure 10.1, non-smooth values are obtained, i.e., densities of neighboring points have sharp jumps. These sharp jumps are not due to the underlying true distribution but are just an artifact of the choice of bin locations. The density estimates from histogram strongly depend on the choice of the number of bins and bin locations. These problems become more serious for high-dimensional datasets. Therefore, KDE is preferred over histograms for density estimation.



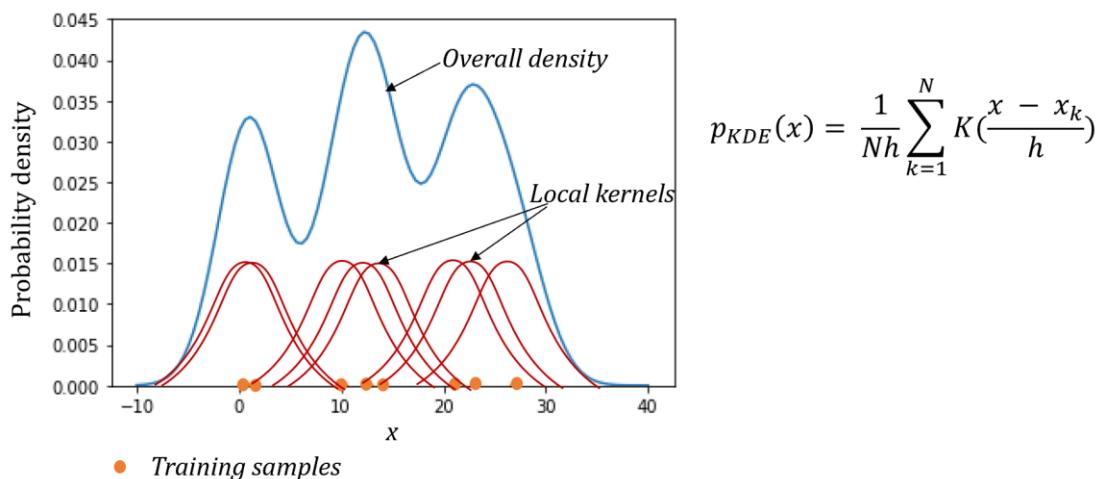
In the beginning of the chapter, we classified KDE and k-NN as non-parametric methods. Parametric vs non-parametric is an important machine learning classification. Parametric methods assume a fixed form of the underlying model where only the model parameters are estimated using data; non-parametric methods do not make any such assumptions. A parametric approach for density estimation would be to assume a gaussian density function and then estimate mean and standard deviation from data. Other popular non-parametric methods that you have already studied are decision trees and SVMs.

Mathematical background

In Gaussian mixture modeling, the overall density was approximated as a sum of local gaussians centered at different clusters. KDE takes this idea to its extreme and puts a local density function (also called kernel) at each data point as shown below for a 1D case, where a popular choice for kernel, K , Gaussian kernel is used, and N is number of training samples.



The parameter h in the above illustration is called bandwidth or smoothing parameter and helps determine the width of local bumps. The overall density at any point ($p_{KDE}(x)$) is obtained by simply summing up the local bumps as shown below.



The kernels are chosen such that $\int_{-\infty}^{\infty} K(x) dx$, the total area under the KDE curve, is 1. For multidimensional case with D dimensions, $p_{KDE}(\mathbf{x})$ is given by

$$p_{KDE}(\mathbf{x}) = \frac{1}{Nh^D} \sum_{k=1}^N K\left(\frac{\mathbf{x} - \mathbf{x}_k}{h}\right)$$

The above multivariate KDE uses the same bandwidth for all axes; different bandwidths are used to deal with non-uniform spread along different axes. We will deal primarily with univariate problems in this chapter.

Deciding KDE hyperparameters

The kernel K and the bandwidth are the two hyperparameters needed by KDE algorithm. Figure 10.2 shows the kernels available with scikit-learn. The choice of kernel is typically not very crucial and Gaussian kernel is usually the default choice.

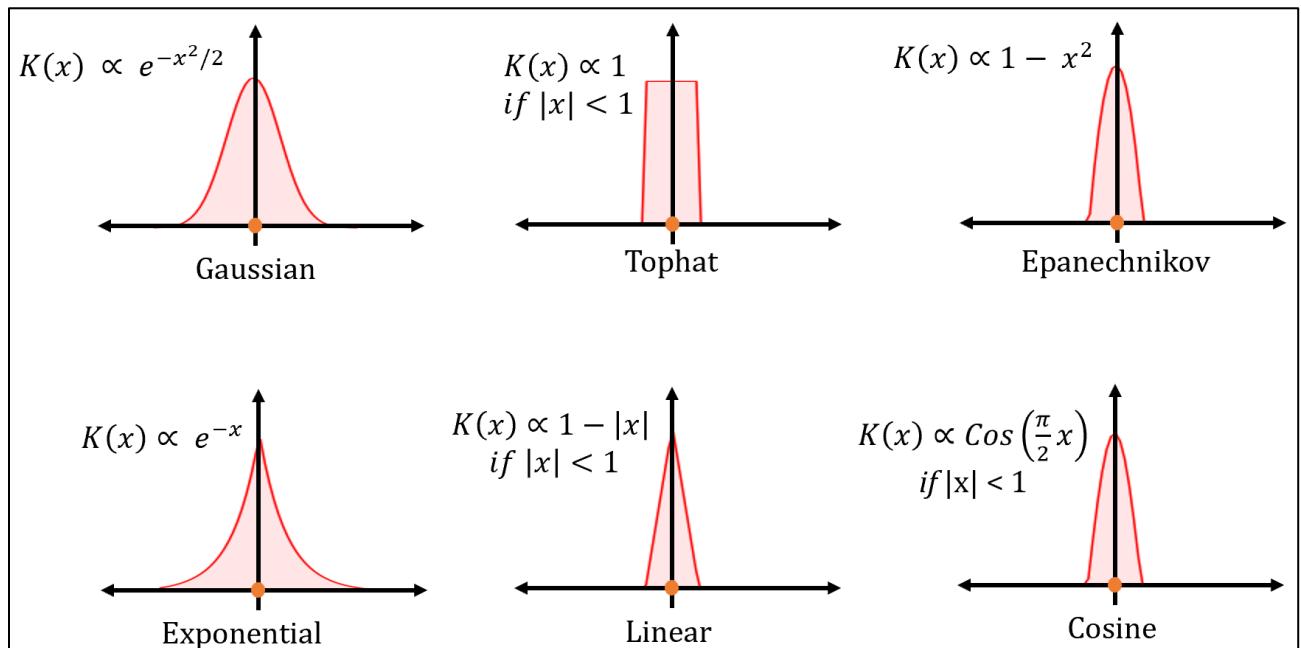


Figure 10.2: Available KDE kernels with scikit-learn

The bandwidth, however, is a crucial parameter and should be chosen carefully. Too small bandwidth leads to spikes in density estimates due to overfitting; absence or presence of a single data-point can significantly impact the density estimate resulting in high variance. Too large bandwidth over-smooths the density curve potentially masking the critical structure in data; the discrepancy between the true and estimated density becomes large resulting in high bias. Figure 10.3 illustrates this bias-variance trade-off.

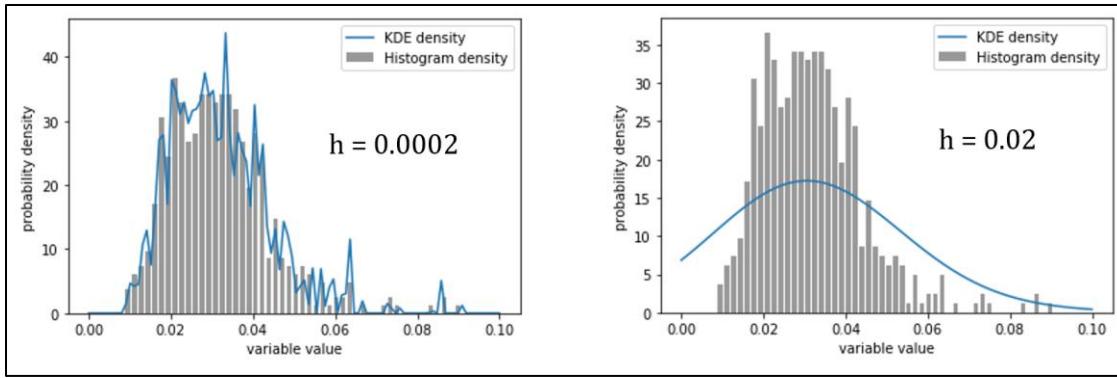


Figure 10.3: Overfitted KDE (left) and underfitted KDE (right)

An empirical approach to choosing h is to assume some underlying standard distribution and then minimize the error between KDE estimate and the assumed distribution. With gaussian assumption, bandwidth is given by

$$h = 1.06\sigma N^{-1/5}$$

where σ is sample standard deviation. A more robust modified version is given by

$$h = 0.9AN^{-1/5} ; A = \min(\sigma, \frac{IQR}{1.34})$$

where IQR is the interquartile range. These empirical approaches, however, may not give good results. A more systematic approach is to use cross validation to choose the bandwidth value that maximizes the probability or likelihood of validation data. For this purpose, GridSearchCV can be used which automatically returns the optimal bandwidth among supplied bandwidth values. GridSearchCV uses score method of KernelDensity estimator which returns the log-likelihood (logarithm of density) at the validation data-points.

To apply KDE, let us revisit our ICA example from Chapter 6. We will determine the monitoring control limits for the TEP process using KDE. Let us first see how to fit an optimal KDE curve to the monitoring metric values. For illustration, we will use I_e^2 values from the training samples. Note that we use the empirical expression of h to decide the list of bandwidth values for grid search.

```
# finding optimal KDE bandwidth by GridSearchCV
from sklearn.neighbors import KernelDensity
from sklearn.model_selection import GridSearchCV

N = len(ICA_statistics_train[:,1])
empirical_h = 1.06*np.std(ICA_statistics_train[:,1])*N**(-1/5)
h_grid = np.linspace(0.5,50)*empirical_h # 50 candidate values
```

```

grid = GridSearchCV(KernelDensity(), {'bandwidth': h_grid}, cv=N) # leave-one-out CV
grid.fit(ICA_statistics_train[:,1][:,None]) # 2D array is expected

print('Empirical h {:.4f}: optimal h {:.4f}'.format(empirical_h, grid.best_params_['bandwidth']))

>>> Empirical h 0.0461: optimal h 0.0047

# optimal KDE and histogram
kde = grid.best_estimator_
metric_grid = np.linspace(0,0.1,100)[:,None]
density = np.exp(kde.score_samples(metric_grid)) # exponential taken because score_samples
                                                 method returns logarithm of density

plt.hist(ICA_statistics_train[:,1], bins=50, color='grey', histtype='bar', alpha=0.8, density=True,
rwidth=0.7, label='Histogram density')
plt.plot(metric_grid, density, alpha=1, label='KDE density')
plt.ylabel('probability density'), plt.xlabel('le2 values')

```

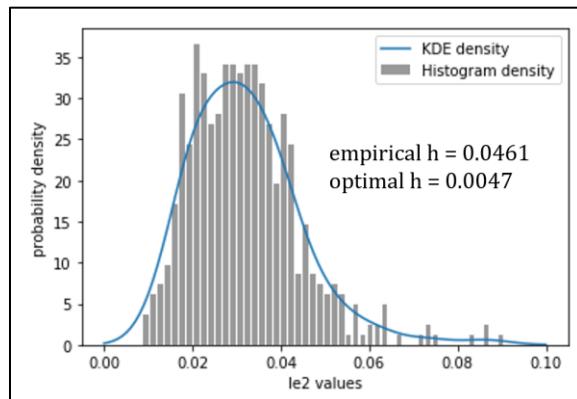


Figure 10.4: Optimal KDE curve fitted to the I_e^2 monitoring metric

We can see that GridSearchCV returns an optimal KDE that correctly reflects the distribution of the underlying data without overfitting the spurious details.

10.2 Determining Monitoring Metric Control Limit via KDE

Let's say we want to find the 99% control limit for our monitoring metric. With an optimal KDE curve on hand, the required control limit would be the metric value such that the area under

the KDE curve to the right of this value is 0.01 as illustrated in Figure 10.5. By definition, there would be only 1% probability that metric value for test sample (which comes from the same distribution from which the training samples were obtained) is higher than the 99% control limit. In process monitoring parlance, this implies that there is a 1% probability that a fault alarm is a false alarm.

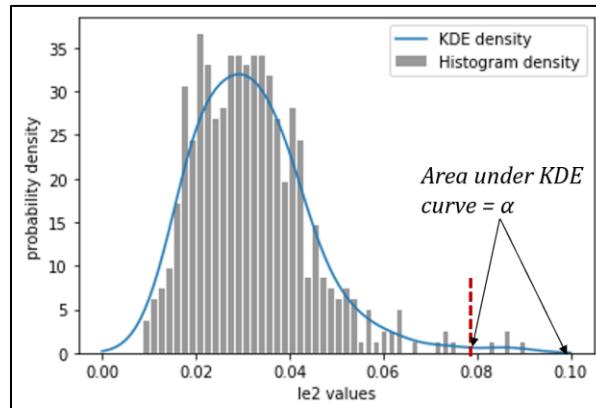


Figure 10.5: $100(1-\alpha)$ percentile or control limit of a monitoring metric. $\alpha = 0.01$ for a 99% control limit or 99th percentile.

Numerical integration is needed to find the area under KDE curve between any two metric values. Let us define a function that takes in metric values and returns a control limit corresponding to a specified percentile. As you can see, we divided the metric axis into a grid (metric_grid) with 100 intervals and computed the area under the KDE curve to the left of each of these grid points. The grid point with the required minimum area (as specified by the percentile parameter) becomes our control limit.

```
def get_CL_via_KDE(metric_values, percentile, metricLabel):
    """ calculate control limit for a monitoring statistic via KDE

    parameters
    -----
    metric_values: numpy array of shape = [n_samples,]
    """

    # find optimal KDE bandwidth via CV
    N = len(metric_values)
    empirical_h = 1.06*np.std(metric_values)*N***(1/5)
    h_grid = np.linspace(0.5,50)*empirical_h

    CV_grid = GridSearchCV(KernelDensity(),{'bandwidth': h_grid}, cv=N) # leave-one-out CV
    CV_grid.fit(metric_values[:,None]) # 2D array is expected
    optimal_h = CV_grid.best_params_['bandwidth']
```

```

# compute density values via KDE
kde = CV_grid.best_estimator_
metric_grid = np.linspace(0, np.max(metric_values), 100)[:,None]
metric_pdf_values = np.exp(kde.score_samples(metric_grid)) # pdf = probability density function

# Find control limit via numerical integration
metric_cdf_values = [np.trapz(metric_pdf_values[:i], None, metric_grid[:i,:], axis=0)[0] for i in range(100)] # cdf = cumulative density function
metric_CL = metric_grid[np.argmax(np.array(metric_cdf_values)) > percentile/100,0]

return metric_CL

```

We can now utilize this function to compute the 99% control limits for our ICA monitoring metrics. We can see that the control limits are similar to what we had obtained earlier in Chapter 6.

```

I2_CL = get_CL_via_KDE(ICA_statistics_train[:,0], 99, 'I2')
le2_CL = get_CL_via_KDE(ICA_statistics_train[:,1], 99, 'le2')
SPE_CL = get_CL_via_KDE(ICA_statistics_train[:,2], 99, 'SPE')

draw_ICA_monitoring_charts(ICA_statistics_train, [I2_CL, le2_CL, SPE_CL], 'training')

```

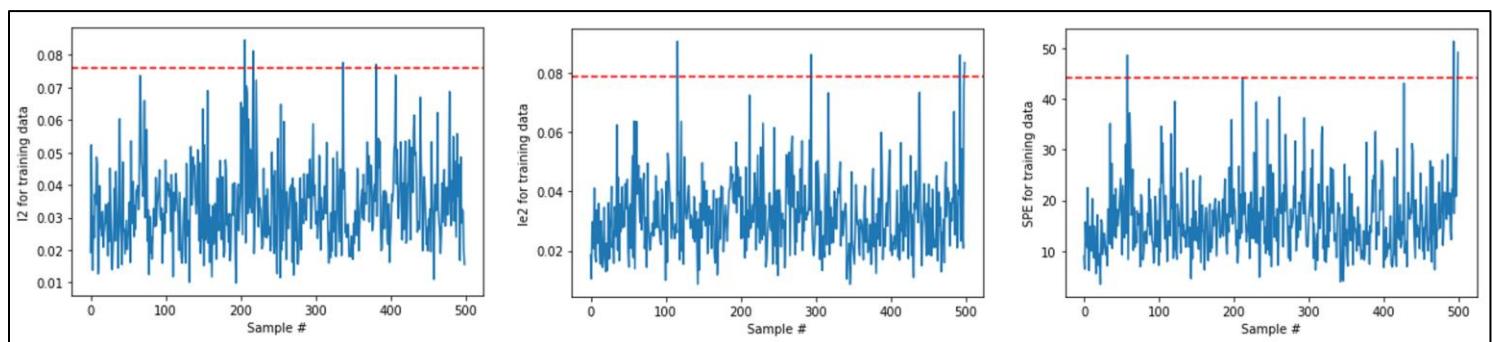


Figure 10.6: ICA monitoring charts for training data with control limits determined via KDE

Data density estimation via KDE has other uses as well. Samples in low-density regions can be segregated from high density regions and discarded as outliers. You can use KDE for the scenario shown in Figure 4.15 for your data pre-processing. Additionally, a process monitoring tool can be built using KDE as well by employing the logic that a test sample falling in low density regions (as estimated from training data) would be an abnormal sample. For all these applications, the procedure for building KDE models remains the same.

10.3 KNN: An Introduction

The k-nearest neighbor (k-NN or KNN) 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 10.7, k-NN can be used for both classification and regression. For classification, KNN 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.⁴⁸

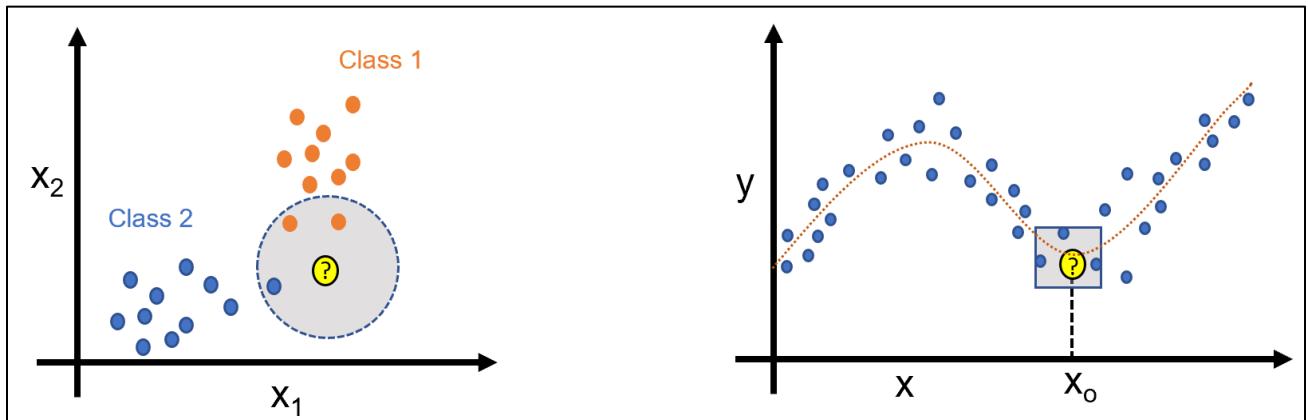


Figure 10.7: KNN illustration for classification (left) and regression (right). Yellow data point denotes unknown test sample. The grey-shaded region represents the neighborhood with 3 nearest samples.



KNN 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, KNN 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.

⁴⁸ 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

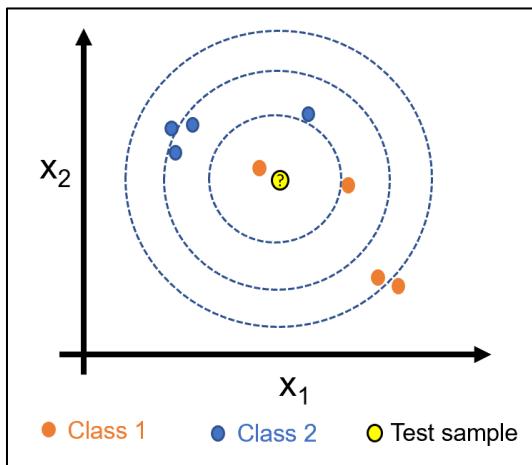


Figure 10.8: Illustration on impact of weight-scheme on k-NN output. Dashed circles are for distance references.

In Figure 10.8, 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 monitoring⁴⁹. 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 data⁵⁰. 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 sensors⁵¹, Borghesan et al. on forecasting of process disturbances⁵², Cecilio et al. on detecting transient process disturbances⁵³, and Zhou et al. on fault identification in industrial processes⁵⁴. These applications may not utilize the k-NN method directly for classification or regression, but use the underlying concept of similarity of nearest neighbors.

⁴⁹ 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

⁵⁰ He and Wang, Fault detection using k-nearest neighbor rule for semiconductor manufacturing processes, *IEEE Transactions on Semiconductor Manufacturing*, 2007.

⁵¹ Facco et al., Nearest-neighbor method for the automatic maintenance of multivariate statistical soft sensors in batch processing, *Industrial Engineering & Chemistry Research*, 2010.

⁵² Borghesan et al., Forecasting of process disturbances using k-nearest neighbors, with an application in process control, *Computers and Chemical Engineering*, 2019.

⁵³ Cecilio et al., Nearest neighbors methods for detecting transient disturbances in process and electromechanical systems, *Journal of Process Control*, 2014.

⁵⁴ Zhou et al., Fault identification using fast k-nearest neighbor reconstruction, *Processes*, 2019

10.4 Process Fault Detection via KNN for Semiconductor Manufacturing Process

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 KNN 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 8 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 feature. Following He & Wang's work, k was set to 5. To compute the monitoring metric threshold, we will use KDE.

```
# 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)
```

```
D2_log_CL = get_CL_via_KDE(D2_log, 95, 'D2_log')
```

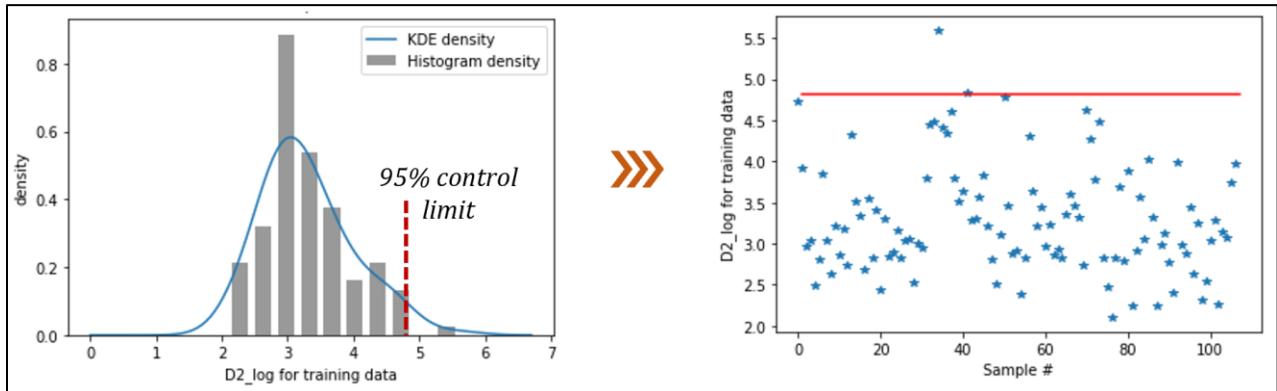


Figure 10.9 FD-KNN monitoring chart for metal-etch training samples

Figure 10.9 shows the output from fitting KDE to k-NN squared distances distribution and the resulting monitoring chart for the training samples. Note that we log-transformed the k-NN squared distances because application of KDE to D^2 resulted in non-zero densities for D^2 below 0. Figure 10.10 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)
```

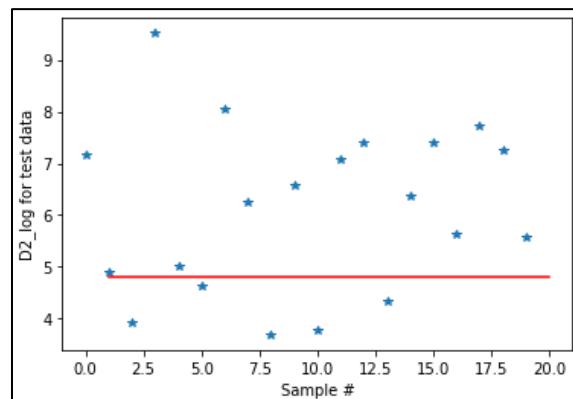
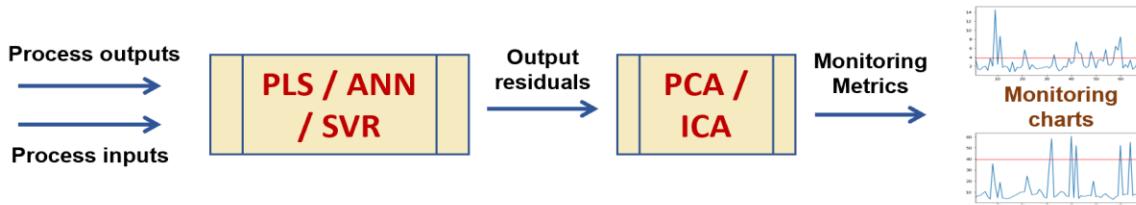


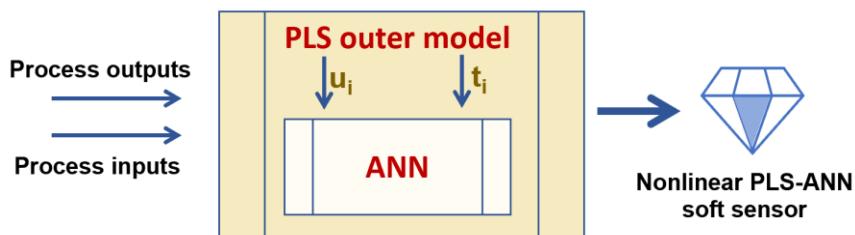
Figure 10.10: FD-KNN monitoring chart for metal-etch test samples

10.5 Combining ML Techniques

In this book, we have looked at several different ML techniques. We attempted to gain conceptual understanding of these methods and not just learn how to apply them via Sklearn modules. This enables us to tweak the standard algorithms of these techniques to suit our specific needs. An area where this can prove very useful is when you need to combine the different ML techniques together to leverage their respective strengths. A common combination scheme is to use PCA for feature extraction and any then apply other method on the extracted features. However, if you dig deeper into the ML for PSE literature, you can find several other interesting combinations. One such scheme (also called external analysis^{55 56}) is illustrated below. In this scheme, the impact of normal plant operating condition changes (as indicated by process inputs) are removed from the output variables before the application of PCA or ICA-based fault detection. The corresponding authors have shown that fault detection performance increases significantly through this scheme.



The secret behind the success of external analysis is that nonlinear or dynamic characteristics of process data can be handled by specialized modeling tools and then output residuals (difference between output measurements and predictions) can be handled efficiently by PCA. Another noteworthy scheme includes the integration of artificial neural networks (ANN) and PLS⁵⁷. The powerful dimension reduction capabilities of PLS are combined with nonlinearity modeling capabilities of ANN to obtain a high-fidelity soft-sensor model.



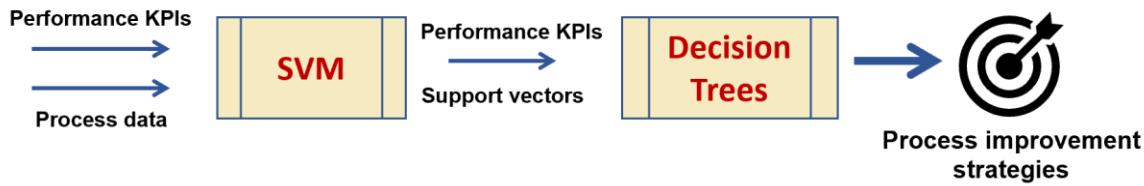
⁵⁵ Kano et al., Evolution of multivariate statistical process control: application of independent component analysis and external analysis, Computers & Chemical Engineering, 2004

⁵⁶ Ge et al., Robust online monitoring for multimode processes based on nonlinear external analysis, Industrial Engineering & Chemistry Research, 2008

⁵⁷ Qin & McAvoy, Nonlinear PLS modeling using neural networks, Computers & Chemical Engineering, 1992

Jemwa & Aldrich⁵⁸ have proposed a scheme that combines SVM and decision trees. This scheme involves judicious use of support vectors to reduce the number of samples that goes into building decision trees model. The final result comprises of optimal operating strategies of the following type

If $0.6 \leq \text{reactant mole fraction} \leq 0.9$
 Then keep reactant temperature within $270 - 290 K$



These were just a few examples to illustrate how very powerful models can be built by careful and conceptually sound combination of different ML techniques. Note that these combination strategies are more ad-hoc and situation-dependent compared to the more systematic ensemble learning techniques. The key takeaway is that as long as you are aware of the underlying mechanism and the combination makes theoretical sense, it does not hurt to try out various combinations if your problem cannot be satisfactorily handled by a standalone technique.

Summary

With this chapter we have come to the end of our journey into the world of classical ML methods. We hope that you already feel empowered and your interest in machine learning has increased. The methods you have learnt so far should enable you to tackle majority of the problems you will face as process data scientists. With the classical techniques conquered, let us now get ready to step into the amazing world of artificial neural networks.

⁵⁸ Jemwa & Aldrich, Improving process operations using support vector machines and decision trees, AIChE Journal, 2005

Part 3

Artificial Neural Networks & Deep Learning

Chapter 11

Feedforward Neural Networks

It won't be an exaggeration to say that artificial neural networks (ANN) have (re)caught the fascination of data scientists and are currently the hottest topic among ML practitioners. Several recent technical breakthroughs and computational advancements have enabled (deep) ANNs to provide remarkable results for a wide range of problems. PSE is no exception to this trend and is increasingly witnessing more applications in areas like surrogate modeling, predictive equipment maintenance, etc.

There is no doubt that ANNs have proven to be very powerful. However, with great power comes great responsibility! If you are not careful with ANN application, you will end up with disappointing results. To train an ANN model efficiently for a complex system, several hyperparameters need to be set carefully, which largely remains a trial-and-error exercise. Nonetheless, several techniques have been devised to overcome the challenges posed by ANN model training.

In this chapter, we will demystify the world of ANNs and gain insights into how ANNs work. We will learn the underlying concepts, the tips & tricks of the trade, common problems encountered, and the techniques developed to handle these problems. Specifically, the following topics are covered

- Introduction to ANNs
- Exploration of nonlinearity in ANNs
- Hyperparameter optimization for ANNs
- Tips & tricks for improved ANN model training
- Nonlinear soft sensing via ANN
- General guidelines for ANN modeling for PSE

11.1 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 11.1. 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. Infact, an ANN with just 1 hidden layer has the universal approximation property, i.e., it can approximate any continuous function with any given accuracy.

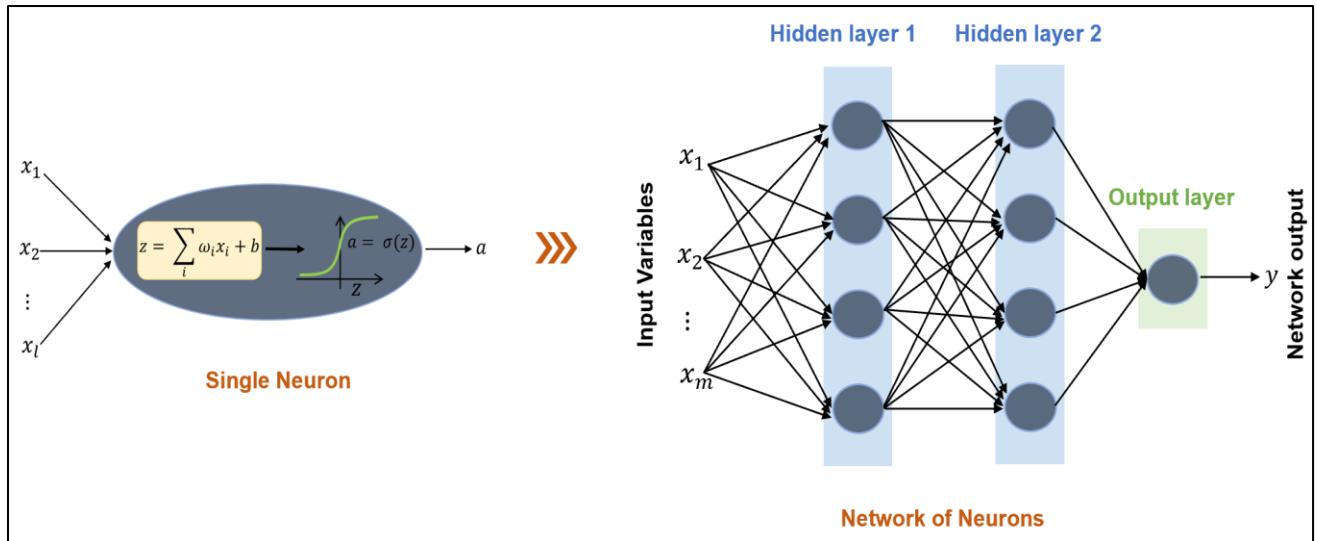


Figure 11.1: Architecture of a single neuron and feedforward neural network with 2 hidden layers for a MISO (multiple input, single output) system

The network shown in Figure 11.1 is an example of fully-connected feed-forward neural networks(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, as opposed to the RNNs which contain feedback loops. In fully-connected networks, neurons between consecutive layers are connected fully pairwise and neurons within a layer are not connected.

While there are different types of ANN architectures (and ML researchers keep coming up with new ones!), FFNNs, RNNs, and CNNs are the most common architectures. While we will study RNNs in the next chapter, in this chapter we will focus on FFNNs. CNNs are primarily employed for image processing and are not covered in this book.

Deep learning

The recent popularity of ANNs can be attributed to the successes with deep learning. Deep learning simply refers to machine learning with deep neural networks (DNNs) which are ANNs with two or more hidden layers (see Figure 11.2). While shallow networks, with large number of neurons, can theoretically model any complex function, deep networks need much fewer model parameters and hence, are theoretically faster to train. Moreover, DNNs enable bypassing manual feature engineering; the first few hidden layers implicitly generate features that are utilized by downstream layers.

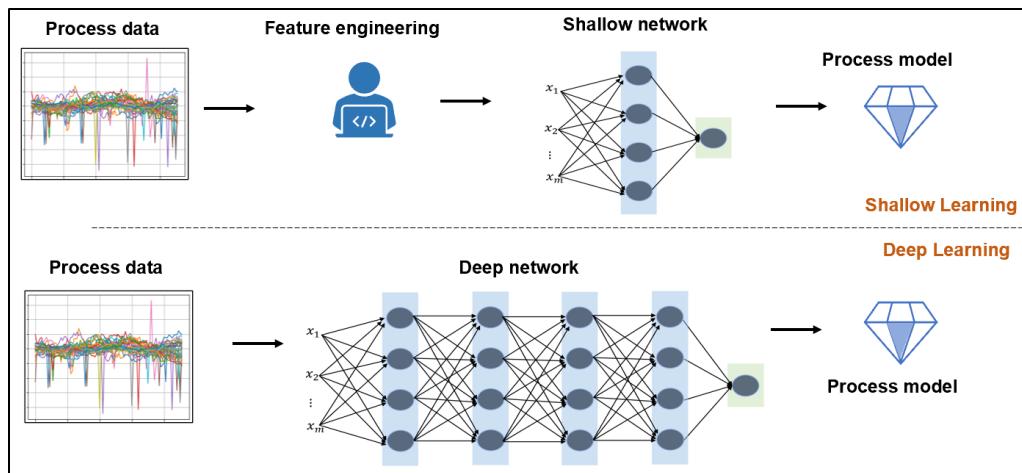


Figure 11.2: Shallow vs deep learning

Although DNNs seem superior to shallow networks, until recently, there weren't many applications of DNNs. DNNs suffered from the issue of vanishing and exploding gradients during model training. However, in recent years, several technical innovations have produced better nonlinear transformers (activation functions), initialization and regularization schemes, and, learning/optimization algorithms. These innovations have largely overcome the model training issues for DNNs and combined with the availability of cheap computing power and large amount of data have resulted in the AI revolutions that we are witnessing today.



A few words of caution: The power of DNNs should not make you undermine the importance of feature engineering. Explicit feature engineering contextualizes process data by considering the specific characteristics of the underlying process system. For example, for a time-series vibration data, feature engineering via frequency-domain transformation and binning can greatly help subsequent network training; without this, a DNN would require substantially large amount of training data to implicitly learn the features.

TensorFlow

For classical ML implementations, we utilized the packages available in Scikit-learn. For ANNs, other specialized libraries are commonly used which make it very easy to build neural net models. TensorFlow (by Google) and PyTorch (by Facebook) are the two most popular deep learning frameworks. These frameworks provide specialized algorithms for efficient training of ANNs.

Direct application of TensorFlow (TF) is not very straightforward and involves a steep learning curve. Modelers often utilized Keras, a high-level API built on top of TF, for ANN modeling. However, the latest version of TF released by Google in 2019 has integrated Keras directly into TF. This has made it possible to define, train, and evaluate ANN models in just a few lines of code. We will use TensorFlow Keras API in this book. The next section shows a quick application for modeling a combined cycle power plant to illustrate the ease of usage of these deep learning libraries.

Neural Nets for Process Systems: Brief History

The deep learning revolution has been a shot in the arm of the ML researchers in PSE community who are now exploring ‘daring’ applications of ANNs such as replacing model predictive controllers (MPCs) with ANNs by ‘learning’ MPC’s optimal policies⁺. However, neural nets are not new for PSE community. Pre-deep learning era saw several interesting ANN applications for process systems as well. Venkatasubramanian⁺⁺ and Himmelblau⁺⁺⁺ provide excellent accounts on the history of ANNs in PSE.

The 90s was a relatively high-activity period which witnessed applications of ANNs for fault detection, soft-sensing, data reconciliation*, etc. This was followed by a decline in the interest in ANNs due to computational limitations and lack of big data. Today, these limitations having largely been overcome, ANNs are poised to make significant impact in PSE.

⁺Kumar et. al., Industrial, large-scale model predictive control with structured neural networks, Computers and Chemical Engineering, 2021

⁺⁺Venkat Venkatasubramanian, The promise of artificial intelligence in chemical engineering: is it here, finally? AIChE Journal, 2018

⁺⁺⁺David Himmelblau, Accounts of experiences in the application of artificial neural networks in chemical engineering, Industrial Engineering & Chemistry Research, 2008

^{*}Karjala, Dynamic data rectification via recurrent neural networks, Ph.D. Thesis, The University of Texas at Austin, 1995

11.2 Process Modeling via FFNN for Combined Cycle Power Plant (CCPP)

We will use data from a CCPP to illustrate the ease with which deep learning models can be built using TF Keras. The dataset (see Appendix for details) was collected over a period of 6 years and contains hourly average values of ambient temperature (AT), ambient pressure (AP), relative humidity (RH), and exhaust vacuum (V). 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. 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 11.3 clearly indicates the impact of input variables on the electrical power (EP).

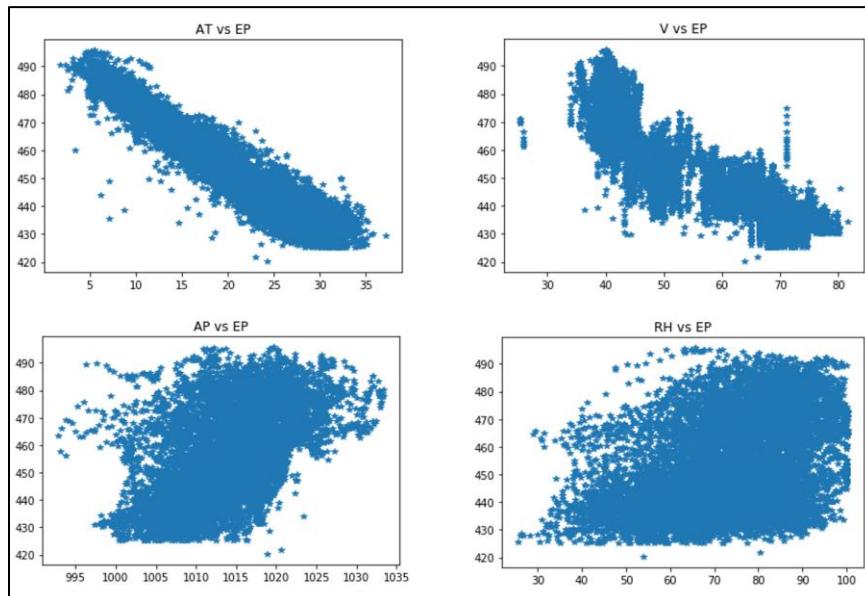


Figure 11.3: Plots of influencing variables (on x-axis) vs Electrical Power (on y-axis)

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 individually⁵⁹. 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.

```
# 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. Do not worry about the activation and kernel_initializer parameters; we will study them in more detail in later sections.

⁵⁹ 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

Next, we will compile and fit the model. Again, do not worry about the optimizer, epochs, and batch_size parameters right now.

```
# 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 11.4 compares actual vs predicted power for the test data.

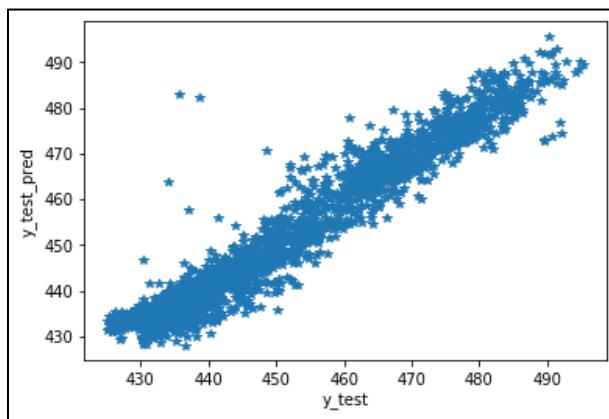


Figure 11.4: Actual vs predicted target for CCPP dataset (obtained $R^2 = 0.93$)

You can use the `model.summary` command to visualize the structure of the network and check the number of model parameters as shown in Figure 11.5. 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.

Layer (type)	Output Shape	Param #
<hr/>		
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	

Figure 11.5: Structure of FFNN model with number of model parameters

11.3 Mathematical Background

Let's consider a sample input $x \in R^4$ from the CCPP dataset and trace its path as it moves through the layers of the network to generate the output. In this forward pass (also called forward propagation), the inputs are 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

$$a_j = g(\mathbf{w}_j^T \mathbf{x} + b_j)$$

$\mathbf{w}_j \in R^4$ 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 $\mathbf{a}^{(1)} \in R^8$

$$\mathbf{a}^{(1)} = g^{(1)}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

where each row of $\mathbf{W}^{(1)} \in R^{8 \times 4}$ contains the weights of a neuron. The same activation function is used by all the neurons of a layer. $\mathbf{a}^{(1)}$ becomes the input to the 2nd hidden layer.

$$\mathbf{a}^{(2)} \in R^5 = g^{(2)}(\mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)})$$

where $\mathbf{W}^{(2)} \in R^{5 \times 8}$. 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

$$\mathbf{a}^{(3)} = \hat{y} = (\mathbf{w}^{(3)})^T \mathbf{a}^{(2)} + b^{(3)}$$

where $\mathbf{w}^{(3)} \in R^5$.

Activation functions

Activation functions (AFs) are what impart non-linear capabilities to neural nets. We used ReLU activation function for CCPP modeling. While ReLU (rectified linear unit) is the preferred AF now-a-days, pre-deep learning era used to employ sigmoid activation function. Illustration below shows the form of sigmoid AF, which transforms pre-activation, z (weighted sum of inputs plus bias), into activation $a \in (0, 1)$. The saturation of sigmoid functions at 1 (or 0) for large positive (or negative) value of pre-activation results in the issue of vanishing/zero gradients for deep networks and is the reason why sigmoid AFs are not favored anymore for hidden layers.

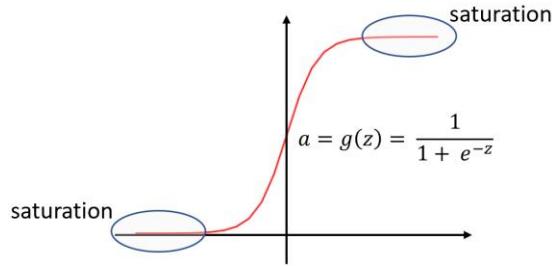
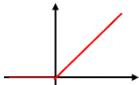
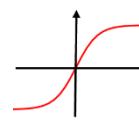


Table below lists other commonly employed activation functions. We will study in a later section how these functions impart non-linearity to ANN models.

Function	Equation	Graph	Remarks
ReLU	$g(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$		<ul style="list-style-type: none"> works very well and preferred AF computationally cheap “dead” neurons for $z < 0$ (not a big issue if ‘learning-rates’ are properly set)
Leaky ReLU	$g(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$		<ul style="list-style-type: none"> alternative if ‘dead’ ReLU neurons cause issues may not perform better than ReLU
Hyperbolic tangent (tanh)	$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$		<ul style="list-style-type: none"> Range: $(-1, 1)$ prone to vanishing gradient issue and computationally expensive

Activation functions for output layer:

The choice of activation function in the output layer depends on the task at hand. For regression problems, no activation function (equivalently, $g(z) = z$ linear function) is used. For classification tasks where the classes are mutually exclusive, softmax function is employed. Figure 11.6 shows a network for 3 exclusive classes with softmax activation in the output layer.

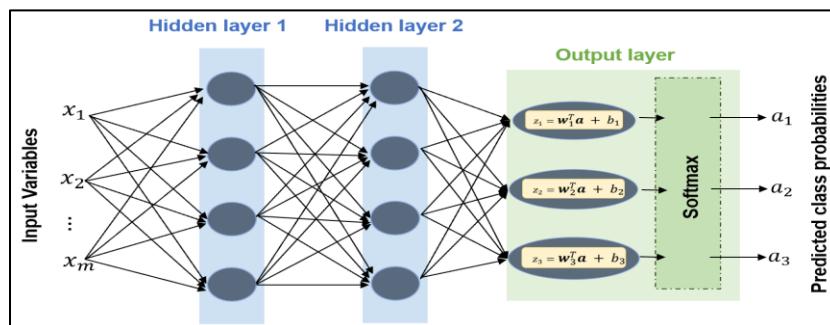


Figure 11.6: FFNN with softmax activation function. (Subscripts denoting layers are not shown for clarity)

Softmax is an exponential function that generates normalized activations so that they sum up to 1. In Figure 11.6, 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)

$$\hat{y} = \text{argmax } (a_j, j \in [1,2,3])$$

As you will see in next subsection, the predicted class probabilities (and not the predicted class label) are directly used during model training. For classification tasks where the classes are not mutually exclusive, sigmoid function is employed in the neurons of the output layer. Sigmoid function is also used for binary mutually exclusive classification problems with a single neuron in the output layer.

Loss functions & cost functions

ANNs optimize their model parameters by minimizing the mismatch between measured and predicted values of the target variable(s). Loss function quantifies this mismatch for a single sample. The average of loss function over the samples of the training dataset that is minimized during model training is called cost function. For regression problems, the commonly employed loss function is mse (mean squared error)

$$\text{MSE Loss} = (y_{sample} - \hat{y}_{sample})^2$$

The corresponding cost function becomes

$$\text{MSE Cost} = \frac{1}{\# \text{ of samples}} \times \sum_{samples} (y_{sample} - \hat{y}_{sample})^2$$

Note that the above expression is for a scalar output; for multi-dimensional output, summation is also carried over different dimensions. Other popular loss metric for regression is mae (mean absolute error) which measures the absolute difference $|y_{sample} - \hat{y}_{sample}|$. MAE cost is more robust to outliers compared to MSE as large errors are not heavily penalized through squaring. Mae metric, however, is less efficient due to the discontinuous nature. A good compromise can be achieved by rmse (root mean squared error) metric.

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.

Gradient descent optimization

Once the cost/objective function ($J(\theta)$) has been defined, the model parameters (θ) are optimized to minimize the cost function. The most common optimization approach used by deep learning libraries is gradient descent where model parameters are estimated in multiple iterations, and in each iteration, parameters are updated in the direction of steepest descent (negative of the gradient of $J(\theta)$ with respect to θ). The expression below shows the update mechanism for the i^{th} model parameter

$$\theta_{i,\text{next}} = \theta_{i,\text{current}} - \eta \frac{\partial}{\partial \theta_i} J(\theta)$$

where η (learning rate), a hyperparameter, determines the magnitude of update in each iteration. For illustration, Figure 11.7 shows the progression of cost function towards its minima through multiple iterations for simple systems (one and two model parameters).

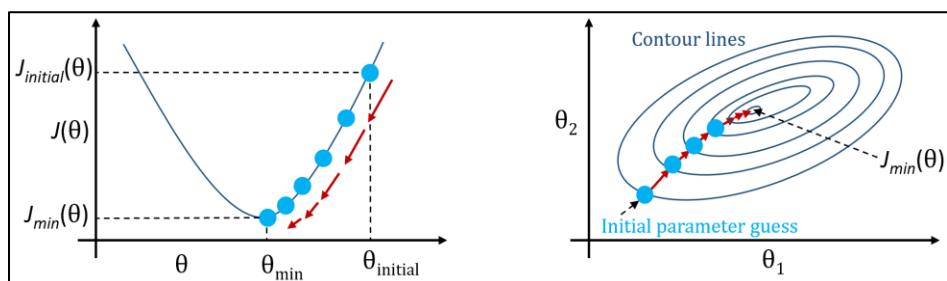


Figure 11.7: Gradient descent iterations for cost function minimization

The learning rate influences the speed of optimization convergence. Large learning rates during initial iterations help to quickly reach close to the minima point; however, large values can also lead divergence instead of convergence. On the other hand, too small learning rates leads to very slow convergence. To deal with this trade-off, learning rate is often adjusted during training via learning rate schedule (reducing η over iterations) or in-built mechanism in modern optimizers (more common approach now-a-days).

A drawback of the classical (or vanilla) gradient descent is that the optimizer is slow and often gets stuck in local minima. As a remedy, several variations of the vanilla approach have been devised which have been shown to provide better performance. These variations include Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and Adam optimization. Among these, Adam (adaptive moment estimation) optimization, which combines the ideas of momentum optimization and RMSProp, is recommended as the default optimizer to use.



Like with everything, Adam may not give good performance for some cases; in those cases, other optimizers should be tried. With Adam optimization you don't have to worry about learning rate scheduling as it's an adaptive learning rate algorithm, although the initial learning rate may need adjustments for some problems. Adam does have a few hyperparameters of its own but usually the default values work fine.

The superior performance of Adam optimization over gradient descent stems from several algorithmic innovations which includes, amongst others, using exponentially decaying average of past gradients to determine update direction. You are encouraged to check out this article⁶⁰ for an intuitive explanation of how these optimizers work.

Epochs & batch-size

In vanilla gradient descent, the entire training dataset is used for computing the gradient in each iteration. Due to this, for large datasets, each iteration can become very slow. This approach, aka batch gradient descent, is also not suitable for updating models online, i.e., updating models quickly with new data samples. Mini-batch gradient descent (MGD) and stochastic gradient descent (SGD) are two alternative approaches.

In mini-batch variation, for each parameter update iteration, only a subset of the training dataset goes into gradient computation. The number of samples in a subset is called batch-size. Batch sizes commonly range between 32 and 256. MGD results in faster convergence

⁶⁰ Sebastian Ruder, An overview of gradient descent optimization algorithms, arXiv, 2017

and noisier/oscillating gradients (different mini-batches can generate very different update directions), as shown in Figure 11.8. An advantage of noisier gradients is that it can help the optimizer to escape local minima. Stochastic gradient descent (not common now) is the extreme case of mini-batch variant where batch-size is just one! Although not technically correct, practitioners employ the term SGD also when mini-batches are used.

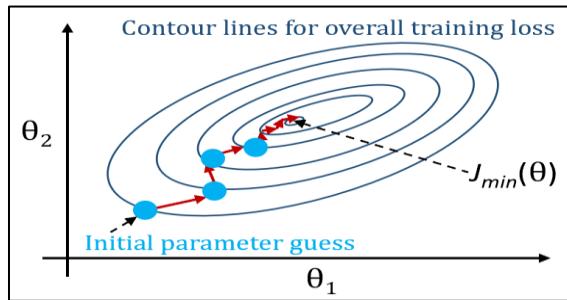


Figure 11.8: Mini-batch & stochastic gradient descents have noisier gradients compared to batch gradient descent

In MGD, several iterations (equal to $\frac{\# \text{ of training samples}}{\text{batch size}}$) are needed for the optimizer to go through the entire training dataset, also referred to as completion of one epoch. At the start of every epoch, training dataset is usually shuffled (to prevent optimizer getting stuck in cycles), divided into mini-batches, and then update iterations continue. Number of epochs is another crucial hyperparameter. Training and validation accuracies are plotted against epochs rather than iterations to keep track of model fitting. More epochs will lead to better training accuracy but can result in worse validation accuracy. We will learn more about it later. A simple thumb-rule is to use more epochs for higher batch sizes.

Backpropagation

Backpropagation computes the gradients of the cost function with respect to the model parameters efficiently through smart usage of chain rule of derivative computation. For conceptual understanding of the mechanism, let's focus on SGD (for ease of explanation) and assume that we are interested in computing the gradient with respect to ω_1 as highlighted in Figure 11.9.

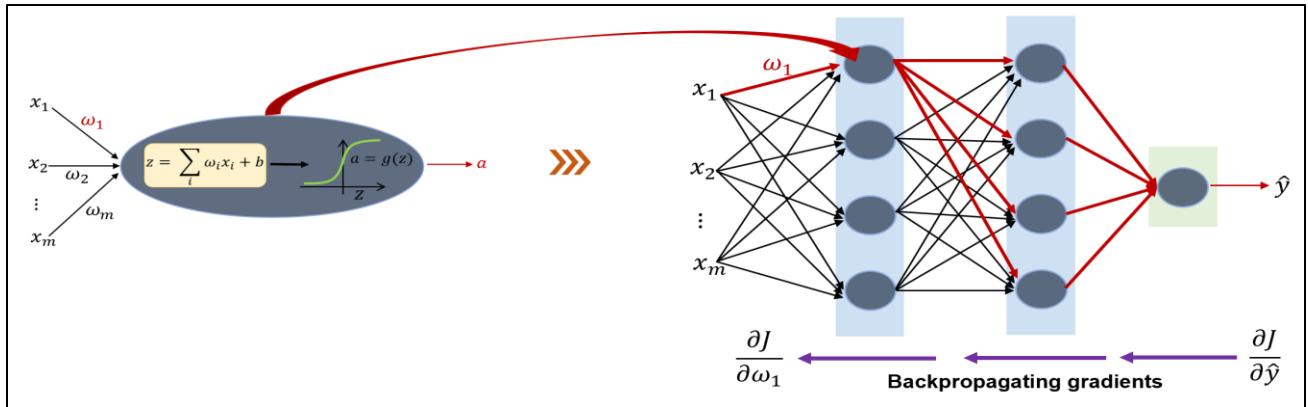


Figure 11.9: Illustration of backpropagation mechanism

Chain rule gives us the following expressions

$$\begin{aligned}\frac{\partial J}{\partial \omega_1} &= \frac{\partial J}{\partial a} \times \frac{\partial a}{\partial \omega_1} \\ &= \frac{\partial J}{\partial a} \times \left(\frac{\partial a}{\partial z} \times \frac{\partial z}{\partial \omega_1} \right)\end{aligned}$$

While $\frac{\partial z}{\partial \omega_1}$ is simply x_1 and $\frac{\partial a}{\partial z}$ is derivative of the activation function, $\frac{\partial J}{\partial a}$ is the unknown element here. However, it is easy to show that this unknown part can be computed if partial derivatives of cost function with respect to the activations of the 2nd hidden layer are known, which themselves need the derivative of cost with respect to output layer activation or \hat{y} . This final missing element is given by

$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} (y - \hat{y})^2 = \frac{1}{2}(y - \hat{y})$$

The above discussion shows that once the activations, predicted output, and error are computed by forward propagation, backward propagation is employed to compute required gradients one layer at a time starting from the last layer. For MGD, through careful bookkeeping, a compact vector-matrix form of backpropagation can be obtained just like we did for forward pass. Note that technically backpropagation only refers to the mechanism for gradient computation, but the term is often used to refer to the entire model fitting algorithm.

Vanishing/Exploding gradients

In the past, the issue of vanishing or exploding gradients posed great difficulty in model fitting for deep networks. In backpropagation, we saw that gradients are propagated from the output layer towards the inner layers. Some of these gradients were derivatives of activation functions, which, for sigmoid and tanh functions, become very small for high pre-activations.

In the case of vanishing gradient scenario, gradients of inner layer parameters become very small due to products of small gradients propagating from the outer layers. This causes virtual halt of inner parameter updates leading to slow convergence and poor model fitting. On the other hand, sometimes, the problem arises due to gradients becoming very large due to large weights. Due to backpropagation the problem gets compounded and causes training failure.

The paper by Glorot and Bengio (titled ‘Understanding the difficulty of training deep feedforward neural networks’) provides excellent investigation on these issues. The usage of sigmoid activation function and poor initialization schemes were found to be some of the culprits. Today, these issues have mostly been curtailed through usage of ReLU functions, batch normalization, correct initialization schemes, etc.

11.4 Non-linearity in Neural Nets (width vs depth)

While designing a neural network for complex nonlinear models, one of the most crucial decisions that you will make is to decide between adding more neurons in a layer (increasing network width) and adding more hidden layers (increasing network depth). Both network width and depth influence network performance, albeit in different manners. Before we delve further into this ‘width vs depth’ question, let us analyze the role of ReLU activation functions in some more details – this will provide some clues to help us make the above decision.

Consider an absolute function that takes a scalar input and outputs the absolute value. This nonlinear function can be perfectly modeled via a FFNN with 1 hidden layer containing 2 nodes with ReLU activation function as shown in Figure 11.10.

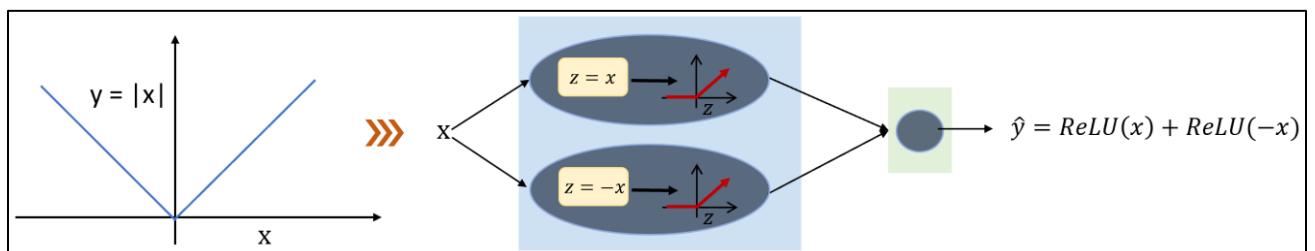


Figure 11.10: Modeling absolute function via FFNN

We were able to model the nonlinear absolute function perfectly because the ReLU-based neurons effectively bifurcated the input space into 2 regions ($x \leq 0$ and $x \geq 0$) where linear relationships hold; essentially, the absolute function got modeled via two piecewise linear functions. Continuing this logic, now assume that the scalar input-output system follows a

nonlinear (quadratic) relationship. This system can again be approximated by using several piecewise linear functions as shown in Figure 11.11. Infact, the approximation shown in the figure was obtained using 1 hidden layer with 5 ReLU neurons.

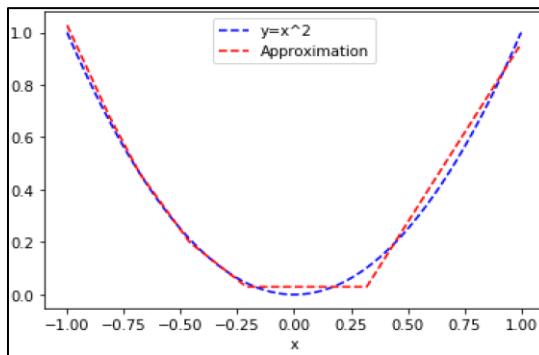


Figure 11.11: Approximation of quadratic function using piece-wise linear functions

What we learn from the above illustrations is that having several neurons in a single layer helps a neural network divide the input space into several local regions and approximate a complex function via several piecewise simpler functions. Consequently, a very nonlinear function can be modeled with a very wide network (large number of nodes in one hidden layer). However, training a very wide network can become problematic.

For a network of fixed width, another way to increase nonlinear capability is to add more depth/hidden layers to the network, enabling the network to estimate more complicated features. This is because the successive hidden layers build upon the nonlinear features generated from the inner layers to impart more nonlinearity to the overall operation. For any activation function, in a wide network with 1 hidden layer, activations of all neurons have similar degree of non-linearity. But, in a deep network, activations from different layers exhibit varied degrees of nonlinearity, with outermost layer exhibiting most nonlinearity. However, as mentioned before, deep networks are prone to vanishing/exploding gradient issues.

You can see that the answer to the ‘more width or more depth’ question is not very straightforward. Similar gain in performance can be gained by increasing network width or depth. A few words of caution here: higher network depth may fail to give adequate performance if network width is inadequately low. For example, irrespective of the network depth, absolute function cannot be accurately modeled if network width is just one. Therefore, both width and depth are important. A thumb-rule is that, for a system with low degree of nonlinearity, a high width-low depth network should be used, while for a very complex system high depth network with adequate width should be used.



For modeling process systems 2 to 3 hidden layers are usually sufficient for both regression and classification tasks. Going deeper helps if you are working with inputs which exhibit hierarchical structure such as images (complex shapes can be broken down into edges, corners, etc.).

11.5 Neural Net Hyperparameter Optimization

We have seen that before executing the `model.fit` command, several hyperparameters have to be specified. These include, amongst others, network depth, network width, number of epochs, minibatch size, learning rates, regularization penalty, activation function type. While general recommendations exist for most of the hyperparameters, specifying the number of hidden layers and the number of neurons in every layer requires some work as these depend largely on the specific problem at hand.

Common approaches for tuning the hyperparameters are based on manual trials and automated grid (or random) searches. In manual approach there are two strategies. You can choose a complex network structure (more layers and neurons than you ‘feel’ necessary) and use regularization techniques and early stopping (these are discussed in next section) to avoid overfitting. Alternatively, you can start with a simple structure (one hidden layer and reasonable number of neurons) and gradually increase the network depth and number of neurons until the overfitting begins.

The automated approach uses `GridSearchCV` from `sklearn`. A `KerasRegressor` wrapper is utilized to enable passing Keras model to `GridSearchCV`. The code below shows the implementation details where we search for the best specifications for network depth and width, regularization penalty, and learning rate. You will notice that we divide the training data into estimation and validation set to pass to `GridSearchCV` for cross-validation. Also, the FFNN model is defined via a function which we pass to `KerasRegressor` wrapper. The best model obtained uses one hidden layer with nine neurons. A R^2 value of 0.931 is obtained.

```
# separate training, validation, 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)
X_est, X_val, y_est, y_val = train_test_split(X_train, y_train, test_size = 0.3, random_state = 100)

# scale data
from sklearn.preprocessing import StandardScaler
```

```

X_scaler = StandardScaler()
X_est_scaled = X_scaler.fit_transform(X_est)
X_val_scaled = X_scaler.transform(X_val)
X_test_scaled = X_scaler.transform(X_test)

y_scaler = StandardScaler()
y_est_scaled = y_scaler.fit_transform(y_est)
y_val_scaled = y_scaler.transform(y_val)
y_test_scaled = y_scaler.transform(y_test)

# import packages
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import regularizers
from tensorflow.keras.optimizers import Adam

# model function
def FFNN_model(hidden_layers, layer_size, regularizationValue, learningRate):
    model = Sequential()
    model.add(Dense(layer_size, kernel_regularizer=regularizers.L1(regularizationValue),
                   activation='relu', kernel_initializer='he_normal', input_shape=(4,)))

    for _ in range(hidden_layers-1):
        model.add(Dense(layer_size, kernel_regularizer=regularizers.L1(regularizationValue),
                       activation='relu', kernel_initializer='he_normal'))

    model.add(Dense(1))
    model.compile(loss='mse', optimizer=Adam(learning_rate=learningRate))

    return model

# KerasRegressor
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor
model = KerasRegressor(build_fn=FFNN_model, epochs=25, batch_size=50)

# gridSearchCV
from sklearn.model_selection import GridSearchCV

param_grid={
    "hidden_layers": [1, 2],
    "layer_size": np.arange(1, 10),
    "regularizationValue": [0.001, 0.01, 0.1],
    "learningRate": [0.05, 0.01, 0.1]}

grid_searchcv = GridSearchCV(model, param_grid)

```

```

grid_searchcv.fit(X_est_scaled, y_est_scaled, validation_data=(X_val_scaled, y_val_scaled))

print("The best parameters obtained are:", grid_searchcv.best_params_)

>>> The best parameters obtained are: {'hidden_layers': 1, 'layer_size': 9, 'learningRate': 0.01,
'regularizationValue': 0.001}

# best model
model = grid_searchcv.best_estimator_.model

```

11.6 Strategies for Improved Network Training

Training ANN models is not easy and there are various issues you may encounter. Maybe your model is overfitting or underfitting, or you are getting stuck in local minima, or your model is not converging. Keeping track of the training process can help to obtain some glimpse into how well the model is fitting and provide hints about potential hyperparameter setting adjustments. One way to keep track of how the model fitting is progressing is to draw plot of training set and validation set costs w.r.t the epochs. Several strategies have been devised to deal with ANN training issues that are frequently encountered. We will study these strategies in this section.

Early stopping

As optimizer iterates for more epochs, training cost goes down; however, the validation cost may begin to increase. This is an indication of model overfitting. Figure 11.12 shows an example (generated with the Kamyr digester dataset). To prevent this, early stopping is adopted, where optimizer aborts training when validation cost begins to increase. To implement this, validation set and early stopping specifications are supplied as shown in the code below. Note the usage of the ‘history’ object which holds record of losses during training for drawing validation plot.

```

# define model
def FFNN_model():
    model = Sequential()
    model.add(Dense(20, activation='tanh', kernel_initializer='he_normal', input_shape=(19,)))
    model.add(Dense(5, activation='tanh', kernel_initializer='he_normal'))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='Adam')
    return model

```

```

# fit model with early stopping
from tensorflow.keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='val_loss', patience=15)

history = FFNN_model().fit(X_est_scaled, y_est_scaled, epochs=250, batch_size=32,
                           validation_data=(X_val_scaled, y_val_scaled), callbacks=es)

# validation plot
plt.figure()
plt.title('Validation Curves')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.plot(history.history['loss'], label='training')
plt.plot(history.history['val_loss'], label='validation')
plt.legend()

```

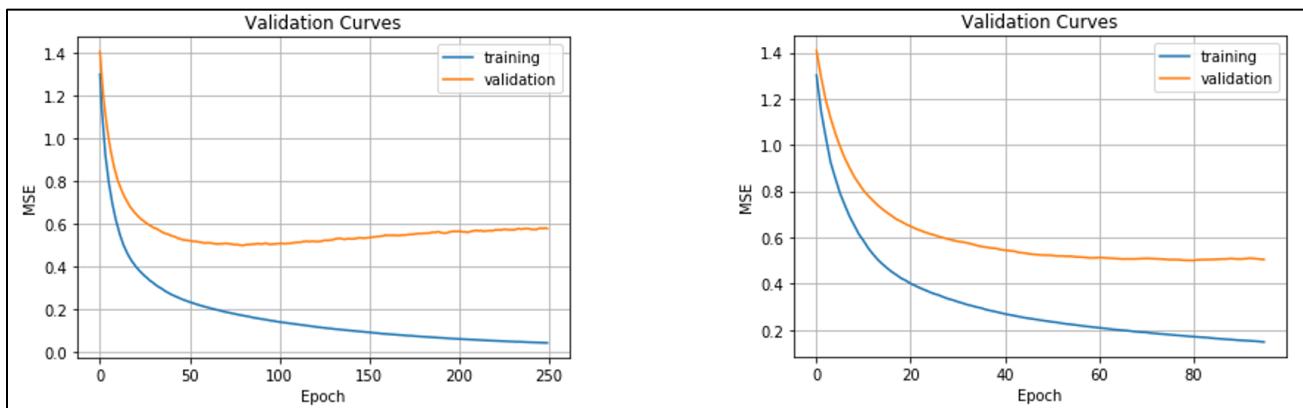


Figure 11.12: Validation plots for Kamyr digester dataset without (left) and with (right) early stopping

With early stopping, we can specify the number of epochs to a large value and not worry about overfitting as the training will automatically stop at the right time. The early stopping callback has several parameters which can be used to alter its behavior. For example, the ‘patience’ parameter that we used specifies the number of epochs to allow with no improvement on validation set after which the training is stopped. You should check out the official Tensorflow documentation⁶¹ for details on other early stopping parameters.

⁶¹ https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

Regularization

As we have learnt before, regularization is a way to constraint model's weight parameters to avoid overfitting (large gap between training and validation accuracies). As one of the best practices, you should always employ regularization for your ANN model fitting. Accordingly, the regularization penalty becomes one of the important hyperparameter. Keras makes specifying regularization (L1, L2, or both) very easy as was shown in the hyperparameter optimization example where L1 regularization was used while defining the Dense layers. Keras allows three types of regularizers, namely, kernel_regularizer, bias_regularizer, activity_regularizer, to regularize the weight parameters, bias parameters, and neuron activations; of these kernel_regularizer is the recommended default choice.

For deep networks, another form of regularization is very popular and is called 'dropout'. In this strategy, in every training iteration, some neurons are 'dropped out', i.e., their activations are not computed in forward pass and weight updates are not performed. Note that dropped neurons in current iteration can become active in the next iteration. Dropout has been shown to provide superior generalization⁶² as it forces the network to avoid relying too much on just a few neurons. In Keras, dropout is specified layer-wise. Two schemes are illustrated below. In scheme 1, dropout layer is added between the input and first hidden layer and dropout rate (fraction of neurons to drop) is set to 0.2; therefore, one out of every 5 input variables are randomly excluded in an iteration. In scheme 2, dropout layer is added between two hidden layers; here again, one out of every 5 neurons in the first hidden layer are randomly excluded in an iteration. Note that dropout is only done during model training.

Dropout on input layer	Dropout on hidden layer
<pre>model = Sequential() model.add(Dropout(0.2, input_shape=(19,)) model.add(Dense(20),...) model.add(Dense(1))</pre>	<pre>model = Sequential() model.add(Dense(20),...) model.add(Dropout(0.2)) model.add(Dense(1))</pre>

Initialization

Before the first optimization iteration is performed, model parameters need to be initialized with some initial values. Not until long ago, a trivial approach would be to initialize all weights to zero or same value. It was however realized that this trivial approach resulted in poor training. Specialized initialization schemes have now been devised. If you are using ReLU activation functions, He initialization scheme should be the default preference; here the weights parameters for neurons in a layer are initialized by drawing samples from a normal

⁶² Srivastava et. al., Dropout: A simple way to prevent neural networks from overfitting, The journal of machine learning research, 2014

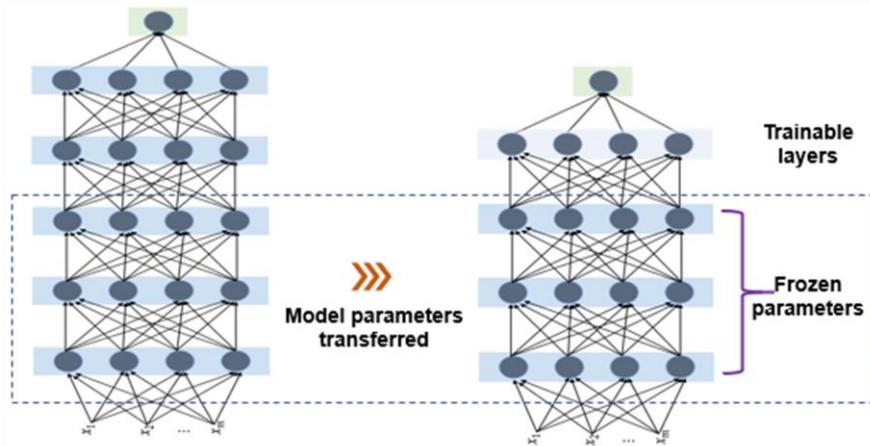
distribution centered at 0 with standard deviation = $\text{sqrt}(2/n_{\text{inputs}})$, where n_{inputs} is number of input connections (which equals number of neurons in previous layer for FFNN). For tanh activation functions, Xavier initialization scheme is found to be more suited.

Batch normalization

Batch normalization is another neat trick that have been devised to overcome the vanishing/exploding gradient issue for deep networks. The initialization schemes ensure that the weights within a layer are distributed appropriately. However, as training progresses and model parameters are update, the weights may lose their ‘nice’ distributions. This negatively impacts model training. With batch normalization strategy, at each training iteration with a mini-batch, the inputs to a layer are standardized. For deep networks, this strategy has been shown to provide significant training improvements. Implementing this in Keras is very straightforward wherein we add (model.add(BatchNormalization())) a batch normalization layer just before the layer whose inputs we wish to standardize.

Transfer Learning

Imagine that you have trained a deep network which provides good performance for some complex tasks, such as fault detection. Now you would like to replicate this to another similar site within your organization. However, you don’t have a lot of training data at this second site – maybe the second site was commissioned recently and hasn’t experienced many process faults yet! In such situations, you may want to adopt the popular strategy of ‘transfer learning’ wherein model parameters of inner layers of a pretrained model is transferred to a new model as shown in illustration below.



As alluded to before, inner layers in a DNN ‘learn’ the low-level features from process data and transfer learning simply transfers this learnt logic from one model to another for related tasks. This leads to faster convergence with limited data*.

*Li et. al., Transfer learning for process fault diagnosis: Knowledge transfer from simulation to physical processes, Computers & Chemical Engineering, 2020

11.7 Soft Sensing via FFNN for Debutanizer Column in a Petroleum Refinery

Let us employ all the knowledge we have gained regarding training FFNN models to develop a soft sensor for a very nonlinear system. For this, we will re-use the debutanizer column dataset that we saw in Chapter 7 (see the Dataset Description section in appendix for system details). The soft sensor is required to predict C4 content in bottoms product using other process data around the column. The dataset contains 2394 samples of input-output process values. The sampling time is 15 minutes. Seven process (pressures, temperatures, flows around debutanizer column) variables are used as inputs. Figure 11.13 shows that dataset has decent process variability.

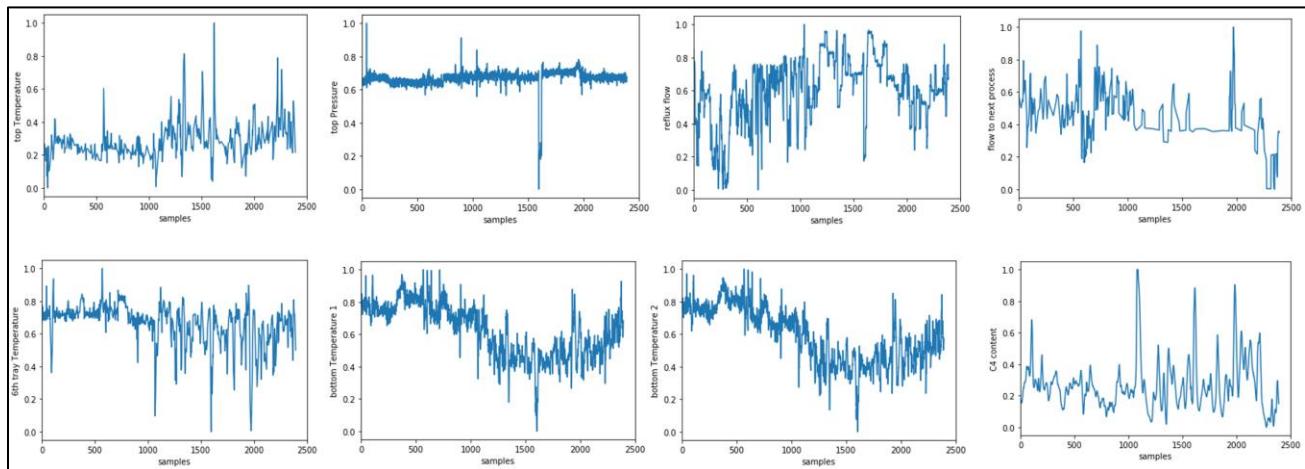


Figure 11.13: Plots of input and output (lower-right plot) variables for the debutanizer column

Before we build the FFNN model, a PLS model is built to serve as a reference for modeling accuracy assessment. Figure 11.14 suggests that the PLS model is grossly inadequate for predicting C4 content and hints the presence of strong non-linearities in the system. Specifically, the linear trend in residual plot clearly shows that the PLS model exhibits poor performance for high and low values of C4 content.

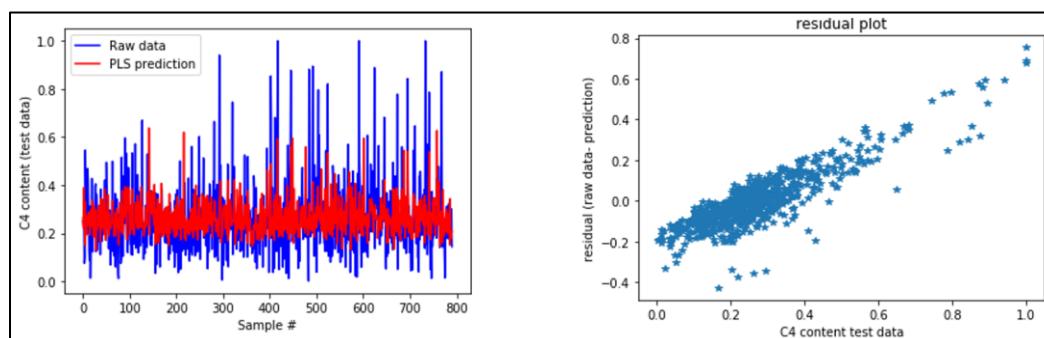


Figure 11.14: Comparison of test data with PLS predictions and residual plot

After a few trials, a network with 2 hidden layers with 60 and 30 neurons was obtained that gave 70% accuracy on test data. The code below shows the values of the rest of the hyperparameters. Figure 11.15 confirms the superior performance of FFNN model over PLS model. The residual plot does not show any significant trend indicating that the FFNN model is able to adequately capture the nonlinear relationships between the input and output variables. This case study demonstrates the power capabilities of ANN models to model complex process systems.

```
# read data
data = np.loadtxt('debutanizer_data.txt', skiprows=5)

# separate training, validation, and test data
from sklearn.model_selection import train_test_split
X, y = data[:,0:-1], data[:,-1][:,np.newaxis]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_state = 100)
X_est, X_val, y_est, y_val = train_test_split(X_train, y_train, test_size = 0.25, random_state = 100)

# import packages
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

# define model
model = Sequential()
model.add(Dense(60, kernel_regularizer=regularizers.L1(0.0000001), activation='relu',
kernel_initializer='he_normal', input_shape=(7,)))
model.add(Dense(30, kernel_regularizer=regularizers.L1(0.0000001), activation='relu',
kernel_initializer='he_normal'))
model.add(Dense(1, kernel_regularizer=regularizers.L1(0.0000001)))

# compile model
model.compile(loss='mse', optimizer=Adam(learning_rate=0.005))

# fit model
es = EarlyStopping(monitor='val_loss', patience=200)
history = model.fit(X_est, y_est, epochs=2000, batch_size=32, validation_data=(X_val, y_val),
callbacks=es)

# predict y
y_test_pred = model.predict(X_test)
```

```

# plot of raw and predicted data
plt.figure()
plt.plot(y_test, 'b', label='Raw data')
plt.plot(y_test_pred, 'r', label='FFNN prediction')
plt.ylabel('C4 content (test data)'), plt.xlabel('Sample #'), plt.legend()

# residual plot
plt.figure()
plt.plot(y_test, y_test-y_test_pred, '*')
plt.xlabel('C4 content test data'), plt.ylabel('residual (raw data- prediction)')
plt.title('residual plot')

# metrics
from sklearn.metrics import r2_score
print('R2 for test dataset:', r2_score(y_test, y_test_pred))

```

>>> R2 for test dataset: 0.707

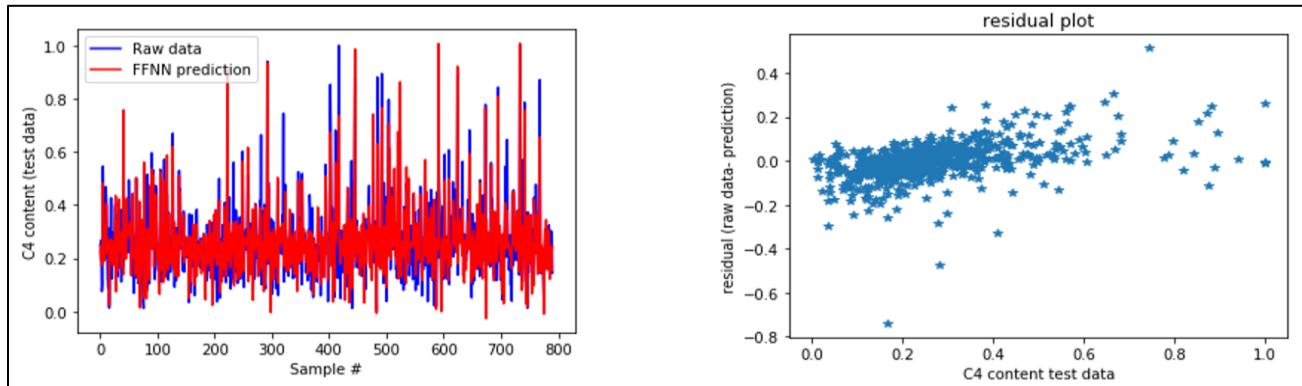


Figure 11.15: Comparison of test data with FFNN predictions and residual plot

FFNN Modeling Guidelines

ANN modeling is a very broad topic and with the plethora of tutorials on ANN modeling available out there, it is easy to get overwhelmed. However, we have seen in this chapter that ANNs are not as daunting as it may seem if we pay careful attention to a few key concepts. If you are looking for some quick guidelines on how to setup the ANN hyperparameters for process systems modeling, then the following suggestions will serve you well.

- **Number of hidden layers:** 1 for simple systems, 2 or 3 for complex systems/tasks
- **Activation function:** ReLU
- **Initialization scheme:** He initialization
- **Optimizer:** Adam
- **Regularization:** L1 or L2 kernel regularization
- **Learning rate schedule:** Not needed with Adam optimizer
- **Number of epochs:** Use a reasonably large value with early stopping
- **Mini-batch size:** 32

You will be lucky if you end up finding a good ANN model with the default settings in the very first attempt. Often the following adjustments may need to be made.

- If validation accuracy is much lower than training accuracy, then increase regularization penalty
- If optimizer is getting stuck in local minima, then adjust Adam's initial learning rate. If it does not help, then try adding more neurons or hidden layers
- If loss vs epoch curve is very noisy, then increase mini-batch size

Summary

Phew! that was a loaded chapter. We covered several techniques that are at our disposal to train ANN models efficiently. There are a lot more behind each of the several aspects that we touched upon, and we have only scratched the surface. However, you now are familiar with the core concepts of ANNs and have hands-on-experience with modeling the debutanizer column and combined cycle power plants. You are now well-aware of what it takes to develop a good ANN model for process systems. ANNs are going to be an important weapon in your arsenal which you will find yourself employing very often. In the next chapter we will learn how ANNs are deployed for dynamic or temporal data.

Chapter 12

Recurrent Neural Networks

In previous chapter, we saw that feed-forward neural networks are powerful mechanism for capturing complex relationships among process variables. In FFNNs, there is an implicit assumption of static relationships between network inputs and outputs. There is no notion of process dynamics or temporal order. However, sometimes you will encounter situations where model output depends on not just the current input but on past inputs as well. Previously, we employed input augmentation technique (in dynamic PCA/PLS) to deal with dynamic systems. Unfortunately, in FFNNs, augmentation can lead to significant increase in the number of model parameters. Recurrent neural networks are specialized networks for efficiently dealing with sequential or temporal data. In essence, RNNs are FFNNs with ‘memory’.

For process systems, RNNs have been successfully used for system identification, fault detection & classification, time series forecasting, and predictive maintenance of industrial equipment. Outside of process industry, RNNs are employed, among others, for natural language processing (speech recognition, text autofill, language translation, sentiment analysis, etc.), music composition, and stock market forecasting!

In this chapter, we will learn how RNNs are able to remember things from past to make future predictions. Since we already covered several neural network concepts in the previous chapter, we will focus on varied applications of RNNs in this chapter. In the process, we will look at different network configurations commonly employed. Specifically, the following topics are covered

- Introduction to RNNs
- Different topological configurations of RNN networks
- System identification via RNN
- Using RNNs for fault classification
- Employing RNNs for predictive maintenance

12.1 RNN: An Introduction

Recurrent neural networks (RNNs) are ANNs for dealing with sequential data, where the order of occurrence of data holds significance. For example, in a production plant, consistently increasing temperature measurements may indicate one kind of process fault while consistently decreasing measurements may indicate another fault type. 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 12.1 shows the architecture of a RNN consisting of a single cell and how it processes a data sequence with ten samples.

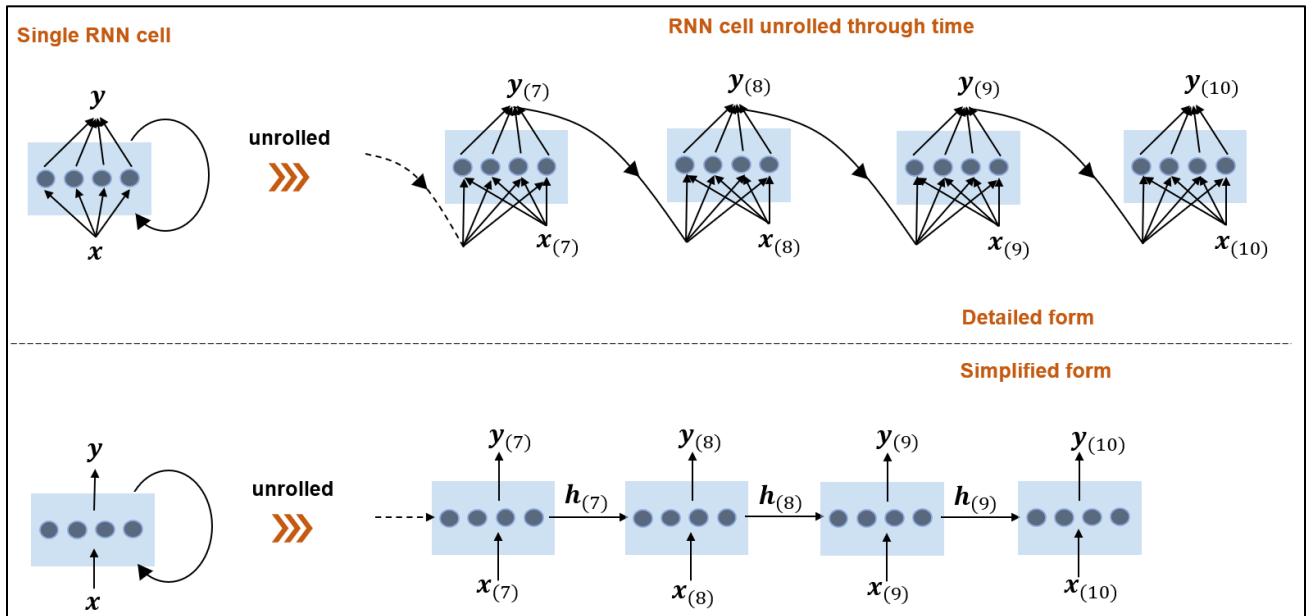


Figure 12.1: Representation of an RNN cell in rolled and unrolled format. The feedback signal in rolled format denotes the recurrent nature of the cell. Here, the hidden state (h) is assumed to be same as intermediate output (y). $h_{(0)}$ is usually taken as zero vector.

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, $x_{(10)}, x_{(9)}, \dots, x_{(1)}$. We will see later how this ‘recurrent’ mechanism leads to efficient capturing of temporal patterns in data.

RNN outputs

If the neural layer in the RNN cell in Figure 12.1 contains n neurons (n equals 4 in the shown figure), then each $\mathbf{y}_{(i)}$ or $\mathbf{h}_{(i)}$ is a n -dimensional vector. For simple RNN cells, $\mathbf{y}_{(i)}$ equals $\mathbf{h}_{(i)}$. Let \mathbf{x} be a m -dimensional vector. At any i^{th} step, we can write the following relationship

$$\mathbf{y}_{(i)} = g(\mathbf{W}_x \mathbf{x}_{(i)} + \mathbf{W}_y \mathbf{y}_{(i-1)} + \mathbf{b})$$

where $\mathbf{W}_x \in \mathbb{R}^{n \times m}$ with each row containing the weight parameters of a neuron as applied to \mathbf{x} vector, $\mathbf{W}_y \in \mathbb{R}^{n \times n}$ with each row containing the weight parameters of a neuron as applied to \mathbf{y} vector, $\mathbf{b} \in \mathbb{R}^n$ 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. We will see later how we can direct Keras to discard all outputs except the last one. Moreover, the last step output may need to be further processed and so a FC layer is often added. Figure 12.2 shows one such topology.

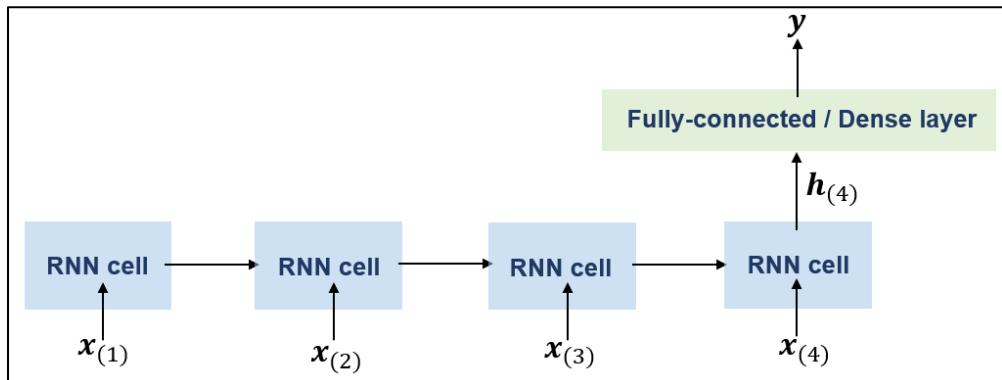


Figure 12.2: Sequence to vector RNN topology with a FC layer

Figure 12.3 shows two other popular architectures, namely, vector to sequence RNN and delayed sequence to sequence RNN. In the former scheme, a single input returns a sequence. A real-life example could be dynamic process response due to a step change in process inputs. The delayed sequence to sequence network (also called encoder-decoder network) is also a many-to-many network, however, here, the output sequence is delayed. This scheme is utilized if each step of output sequence depends on the entire input sequence. Encoder-decoder scheme is often utilized for language translation because initial words of the translation can be influenced by final words of the sentence being translated.

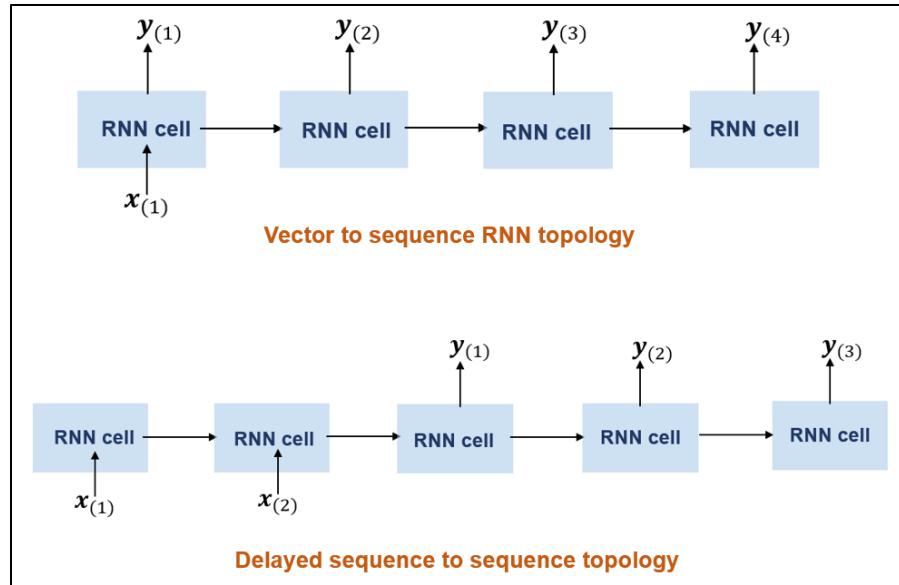


Figure 12.3: Vector to sequence and delayed sequence to sequence networks

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 12.4. Moreover, in a LSTM cell, the internal state is stored in two separate vectors, $\mathbf{h}_{(t)}$ or hidden state and $\mathbf{c}_{(t)}$ or cell state. Both these states are passed from one LSTM cell to the next during sequence processing. $\mathbf{h}_{(t)}$ can be thought of as the short-term state/memory and $\mathbf{c}_{(t)}$ as the long-term state/memory (we will see later why), 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. While we will understand the mathematical details later, for now, it suffices to know the following about these gates:

- **Forget gate** determines what parts of long-term state, $\mathbf{c}_{(t)}$, are retained and erased
- **Input gate** determines what parts of new information (obtained from processing of $\mathbf{x}_{(t)}$ and $\mathbf{h}_{(t-1)}$) are stored in long-term state
- **Output gate** determines what parts of long-term state are passed on as short-term state

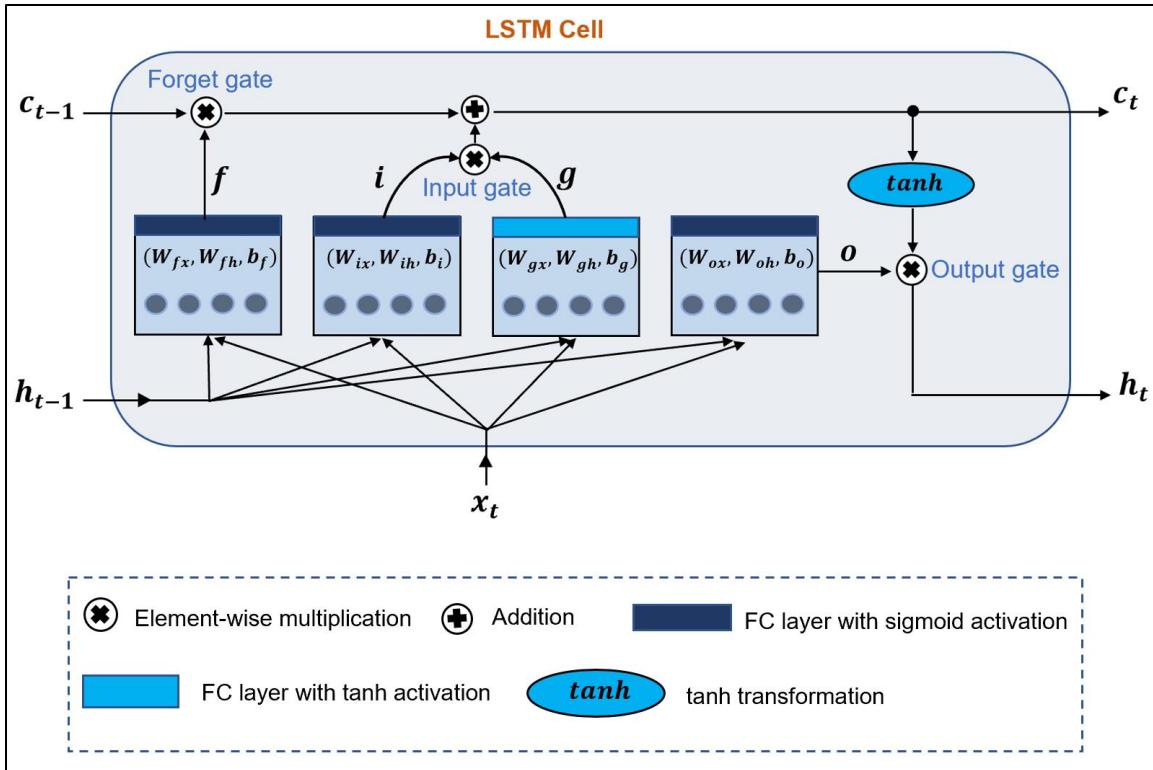
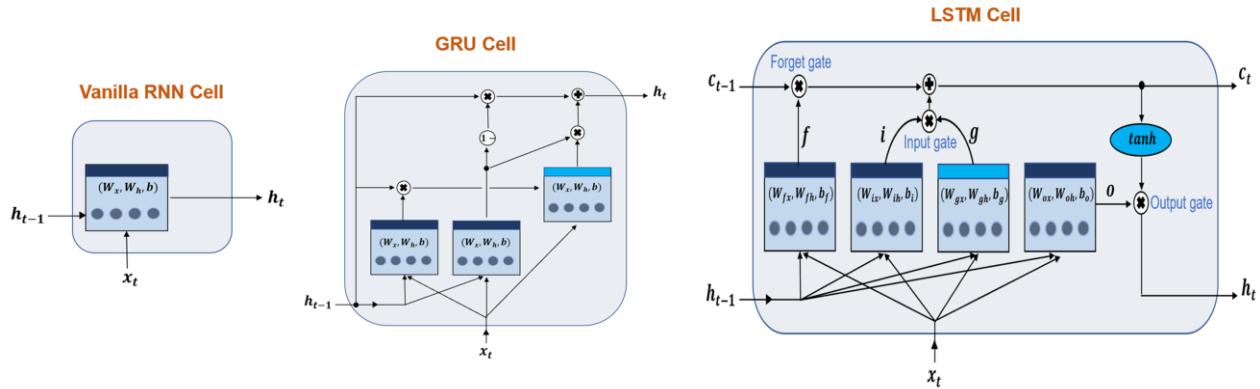


Figure 12.4: Architecture of an LSTM cell. Three FC layers use sigmoid activation functions and one FC layer uses tanh activation function. Each of these neural layers have its own parameters \mathbf{W}_x , \mathbf{W}_h , and \mathbf{b}

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 networks are the default RNNs now-a-days. In the next section we will see a quick application of RNN for system identification using LSTM cells.

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.



12.2 System Identification via LSTM RNN for SISO Heater System

The task of building mathematical models of dynamical processes using input and output data is referred to as system identification (SI). RNNs are aptly suited for SI as they are designed to capture dynamic relationships. To illustrate this, we will use data from a single input single output (SISO) heater system where heater power output is manipulated to maintain desired system temperature. The dataset contains 4 hours of training data and 14 minutes of test data. Data is sampled at every second. Our modeling objective is to predict the next temperature value using the current and past data. Let us first quickly look at training data.

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

# read data
data = pd.read_csv('TCLab_train_data.txt')
heaterPower = data[['Q1']].values
temperature = data[['T1']].values
```

```

# plot data
plt.plot(temperature, 'k'), plt.ylabel('Temperature'), plt.xlabel('Time (sec)')

plt.figure()
plt.plot(heaterPower), plt.ylabel('Heater Power'), plt.xlabel('Time (sec)')

```

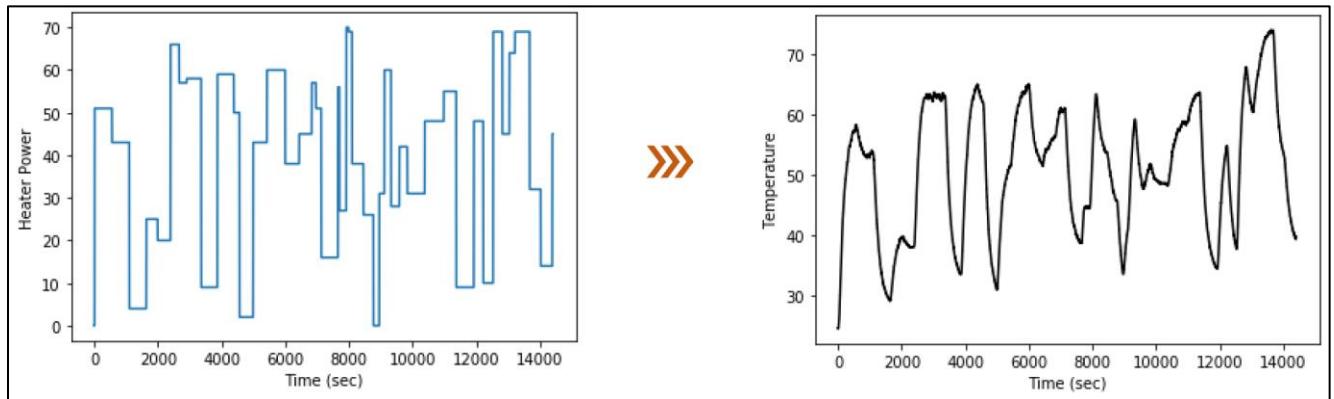


Figure 12.5: Plots of input and output data of a dynamic heater system

Figure 12.5 clearly shows the dynamic nature of data; quick, large, and consistent variations in heater power leads to temperature being in transient state most of the times. It is obvious that temperature value at any instant would depend on not just the current heater power but also the past values of heater power. To reduce the amount of past data required, we will use both power and temperature values from the past to predict current temperature value. In SI terminology, we are attempting to build an ARX (autoregressive with exogenous variables) model.

```

# decide model input-outputs and scale data
from sklearn.preprocessing import StandardScaler

X = data[['T1','Q1']].values
y = data[['T1']].values

X_scaler = StandardScaler()
X_scaled = X_scaler.fit_transform(X)

y_scaler = StandardScaler()
y_scaled = y_scaler.fit_transform(y)

```

Now that we have decided our model inputs and outputs variables, there is one more data rearrangement that needs to be done. Currently, the shape of **X** array is {# of samples, # of features or input variables}. For RNN, this input data matrix needs to be converted into the shape {# of sequence samples, # of time steps, # of features} such that each entry (of shape {# of time steps, # of features}) along the 0th dimension is a complete sequence of past data

which is used to make prediction of temperature values. Here, the number of time steps is taken to be 70^{63} and number of features equals 2. If you do not have a good idea of how many time steps to use for your system, then it will become a model hyper-parameter which will need to be optimized. For each input sequence, a scalar temperature value is predicted and therefore, the **Y** matrix is re-arranged accordingly as shown in the code below

```
# rearrange X data into (# sequence samples, # time steps, # features) form
nTimeSteps = 70
X_train_sequence = []
y_train_sequence = []

for sample in range(nTimeSteps, X_scaled.shape[0]):
    X_train_sequence.append(X_scaled[sample-nTimeSteps:sample,:])
    y_train_sequence.append(y_scaled[sample])

X_train_sequence, y_train_sequence = np.array(X_train_sequence), np.array(y_train_sequence) # convert lists of arrays into arrays
```

Basically, each block of 70 continuous rows in (scaled) **X** array becomes a sequence. The topology of RNN that we will build is same as that shown in Figure 12.2, except for the number of time-steps. Like we did for FFNN modeling, we will import relevant Keras libraries. In the code below, a LSTM RNN layer is followed by a dense layer.

```
# import Keras libraries
from tensorflow.keras import Sequential
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dense
from tensorflow.keras import regularizers

# define model
model = Sequential()
model.add(LSTM(units=25, kernel_regularizer=regularizers.L1(0.001), input_shape=(nTimeSteps,2)))
# LSTM cell with 25 neurons in each of the 4 neural layers
model.add(Dense(units=1))
# 1 neuron in output layer
```

The above 3-line code completely defines the structure of the required RNN. The single neuron in the output layer does the job of transforming the 25th dimensional hidden state vector from the last step of RNN layer into a scalar value. Note that by default LSTM layer returns only the last step output. The model summary below shows the number of model

⁶³ <https://apmonitor.com/do/index.php/Main/LSTMNetwork>

parameters in each layer. We will understand in the next section how we ended up with 2800 parameters in the LSTM layer.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 25)	2800
dense (Dense)	(None, 1)	26
=====		
Total params: 2,826 Trainable params: 2,826 Non-trainable params: 0		

Figure 12.6: Structure of RNN model with number of model parameters

Next we compile and fit the model. Note that instead of providing an explicit validation dataset, we used the `validation_split` parameter to specify 30% validation split.

```
# compile and fit model with early stopping
from tensorflow.keras.callbacks import EarlyStopping
model.compile(loss='mse', optimizer='Adam')
es = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
history = model.fit(X_train_sequence, y_train_sequence, epochs=100, batch_size=250,
validation_split=0.3, callbacks=[es])

# check fit on training data
y_train_sequence_pred = model.predict(X_train_sequence)
y_measured = y_scaler.inverse_transform(y_train_sequence)
y_pred = y_scaler.inverse_transform(y_train_sequence_pred)
```

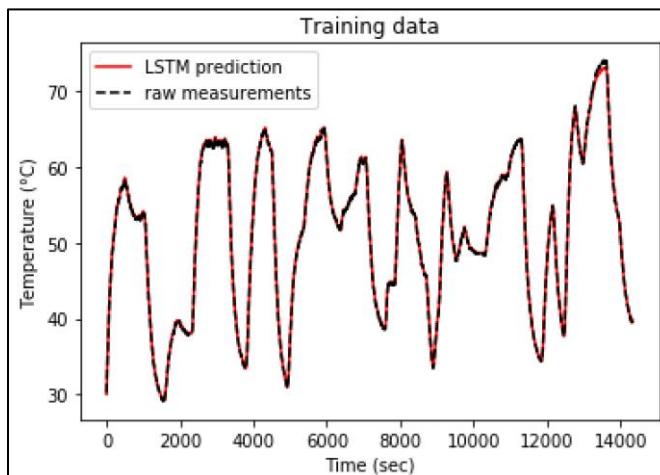


Figure 12.7: Measured vs predicted temperature values for SISO heater system

Figure 12.7 shows that our RNN can fit the training data well. Let us now check its performance on test dataset.

```
# read test data
data_test = pd.read_csv('TCLab_test_data.txt')
X_test = data_test[['T1','Q1']].values
y_test = data_test[['T1']].values

# scale data
X_test_scaled = X_scaler.transform(X_test)
y_test_scaled = y_scaler.transform(y_test)

# re-arrange data into sequence form
X_test_sequence = []
y_test_sequence = []

for sample in range(nTimeSteps, X_test_scaled.shape[0]):
    X_test_sequence.append(X_test_scaled[sample-nTimeSteps:sample,:])
    y_test_sequence.append(y_test_scaled[sample])

X_test_sequence, y_test_sequence = np.array(X_test_sequence), np.array(y_test_sequence)

# predict y_test
y_test_sequence_pred = model.predict(X_test_sequence)
y_test_pred = y_scaler.inverse_transform(y_test_sequence_pred)
```

Figure 12.8 confirms the good generalization capability of the RNN model as the model predictions match the raw temperature values in test dataset very well. Note that model predictions are not available before the first 70 samples.

It is now a good time for a quick comparison between FFNN and RNN modeling. If we had built a FFNN model with past power and temperature values as additional input variables, then we would have ended up with 140-dimensional input variable. Assuming we use a hidden layer with only 25 neurons followed by a single output neuron, we would have ended up with more than 3500 model parameters – 25% more than that used by our RNN model. This rough thought-experiment shows how RNNs end up with better parameter efficiency for modeling dynamic systems. And by now, you would know that lower number of model parameters generally implies lower chances of over-fitting and better model training. Therefore, for dynamic systems, RNNs should be the preferred model choice.

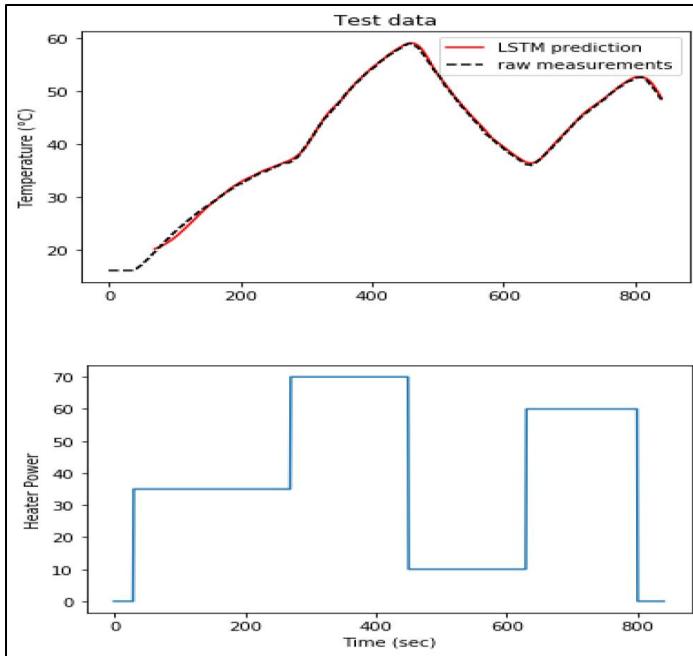
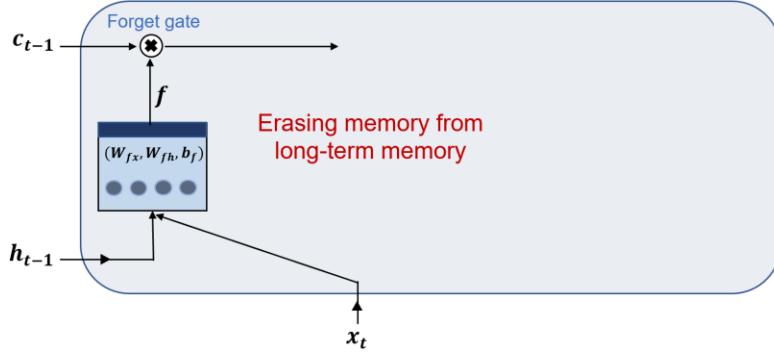


Figure 12.8: Measured vs predicted temperatures for SISO heater system test dataset

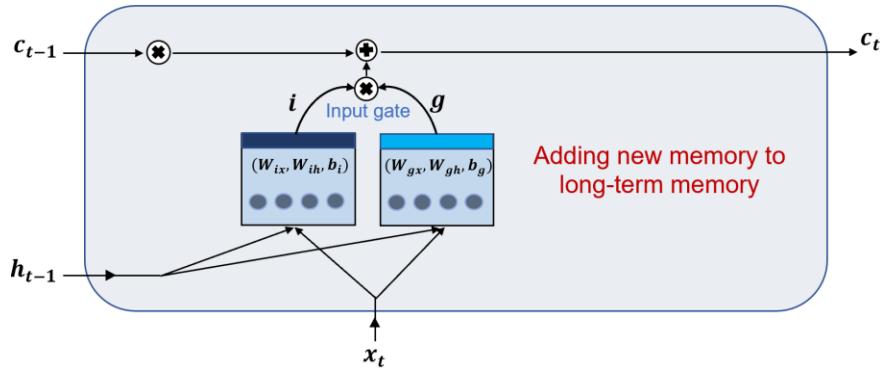
12.3 Mathematical Background

It is always nice to have a good understanding of the impact of adding additional neurons on total number of model parameters; pre-emptive steps can be taken to prevent over-fitting as model complexity increases. In vanilla RNN cell, we saw that $W_x \in \mathbf{R}^{n \times m}$, $W_h \in \mathbf{R}^{n \times n}$ and, $b \in \mathbf{R}^n$. This leads to total number of parameters equal to $mn + n^2 + n$. In an LSTM cell, each of the 4 neural layers have their own set of these parameters. Therefore an LSTM cell has $4(mn + n^2 + n)$ model parameters. In the previous heater example, we have $m = 2$ and $n = 25$; therefore we ended up with $4(50 + 625 + 25) = 2800$ model parameters in the LSTM layer.

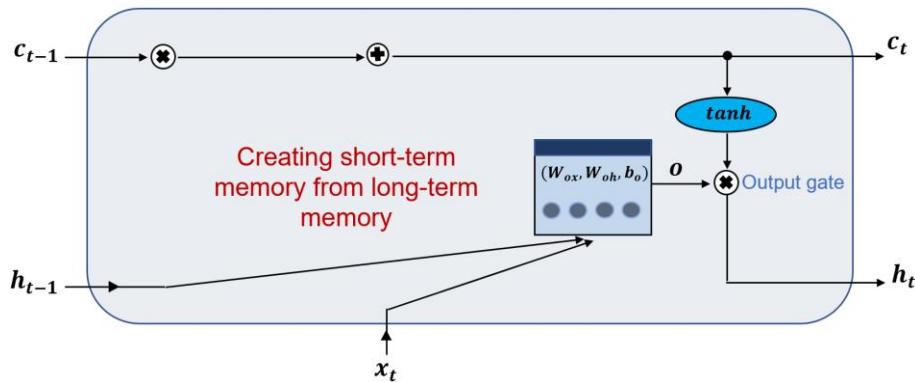
Let us now take a sneak peak into the inner workings of a LSTM cell. An LSTM cell may seem intimidating, however, they are actually simple to understand. Let us look at each of the neural layer and the gates, and understand the specific roles they play. The core component of the LSTM cell is the cell state ($c_t \in \mathbf{R}^n$) that almost cuts through the cell untouched, except for a couple of element-wise operations which modify the individual elements/components of c_t . These modifications determine which components are useful or no longer useful for further processing of the sequence. To understand these modifications, consider the first of the 4 neural layers.



Here, the input (x_t) and the hidden state from previous step (h_{t-1}) are processed to generate vector f ($\in R^n$) whose values lie between 0 and 1. If the i^{th} component of f is 0, it implies that the i^{th} component of c_{t-1} gets erased/forgotten after the element-wise multiplication (and hence, the name ‘forget gate’). The second and third neural layers generate n-dimensional vectors i and g , respectively. The component values of i ($\in [0, 1]$) determine how much the components of g ($\in [-1, 1]$) are passed through the input gate and get added to the cell state.



These are all the modifications that occur to the cell state before it is passed on to the next LSTM cell. However, there is one more major task that occurs inside LSTM cell. The hidden state, $h_t \in R^n$, is generated from the cell state, which acts as the cell output at any sequence step. The fourth neural layer participates in this task as shown below.



The vector output, \mathbf{o} ($\in \mathbb{R}^n$), from the fourth neural layer determines which parts (and how much) of the cell state is output as the hidden state at this time step. The cell state is passed through a tanh function before the output gate to push its values to be between -1 and 1.

We hope that the above description gave you a good conceptual overview of how memories are created and passed along the RNN chain during sequence processing to capture long-term patterns in any sequence. We will now continue to look at a few more interesting application of LSTMs.

12.4 Stacked/Deep RNNs

In FFNNs, we created deeper networks by adding more hidden layers between the input and output layers. Similarly, in RNNs, deep networks can be built by stacking RNN cells on top of each other, as shown in Figure 12.9. While both the cell state and hidden state are passed along the chain (horizontally), only the hidden state are passed from one layer to the next (vertically). The hidden states of the cells in the last layer become the network outputs.

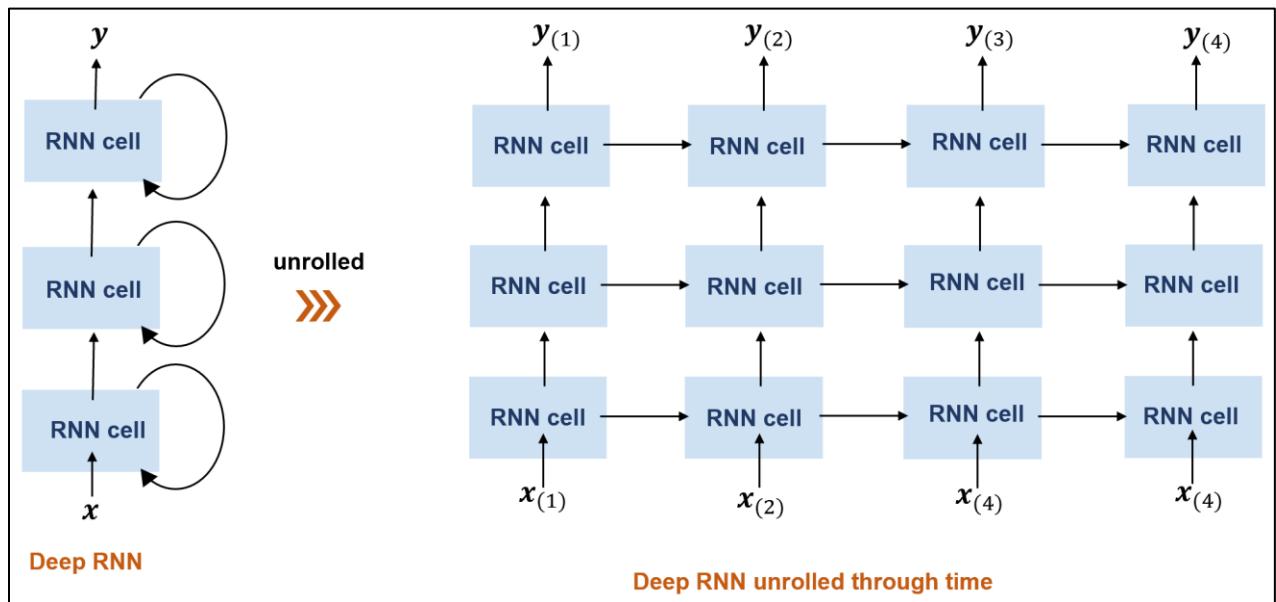


Figure 12.9: Representation of a deep RNN in rolled and unrolled format. The hidden state (h) is passed from one layer to the next.

Like we did for shallow RNN, we can ignore all outputs except that from the last step to obtain a sequence-to-vector network in deep RNNs as well. We will see one such application for fault classification for a large-scale system in the next section.

12.5 Fault Classification via LSTM for Tennessee Eastman Process

The way process variables evolve in the presence of a process fault can give crucial clues about the nature/kind of process fault. Therefore, a sequence of process values (rather than just a single snapshot of current process values) is often used as inputs for fault classification. RNNs are best suited for classification using sequence data. To illustrate this, we will use data from Tennessee Eastman Process (TEP), a large-scale chemical manufacturing process.

In chapter 6, we had used simulated TEP data for fault detection and classification using static data to introduce ICA and FDA methods. We will again use simulated data but from another source⁶⁴ where a much bigger dataset is available. The dataset still contains 21 different fault classes (along with no-fault operation); however, for each fault class, 500 separate simulations runs were conducted for both training and test dataset. Each training simulation run contains 500 time samples from 25 hours of operation and each test run contains 960 time samples from 48 hours of operation. Figure 12.10 shows temporal profiles for the first 10 signals in the dataset for 4 complete simulation runs with no fault, with fault of class 1, and with fault of class 8.

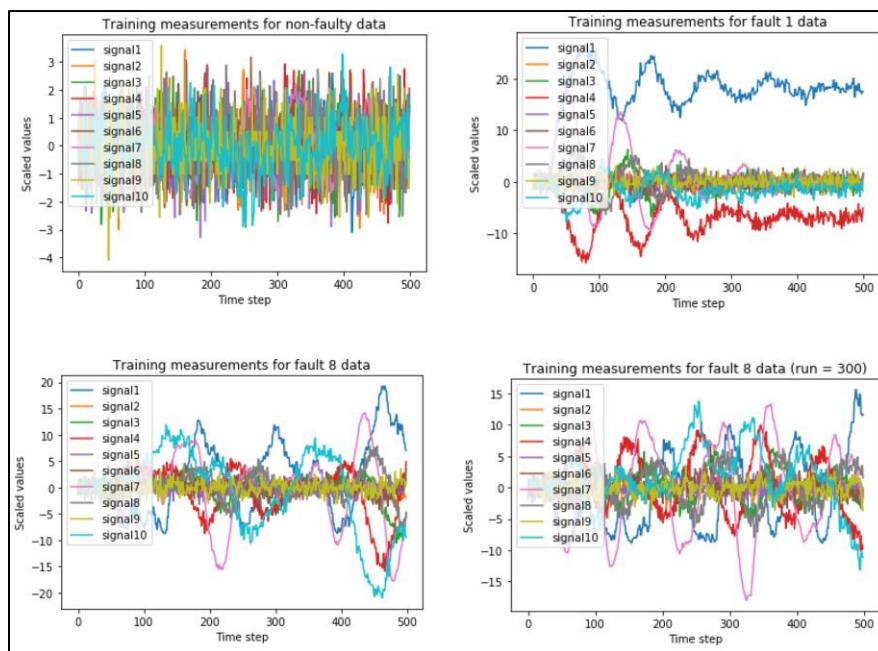


Figure 12.10: Dynamic profile of first 10 signals for complete simulation runs with no fault, with fault of class 1, and with fault of class 8.

⁶⁴ Reith, C.A., B.D. Amsel, R. Tran., and B. Maia. Additional Tennessee Eastman process simulation data for anomaly detection evaluation. Harvard Dataverse, Version 1, 2017

Figure 12.10 suggests that just the end values of process variables may not accurately indicate the specific process fault; instead, the temporal profiles hold the key to accurate fault classification. Our objective is to predict the fault class (including no fault class) using the complete simulation run data. Let's begin by loading the training data into python. The original data is in .RData format which we can read using pyreadr package.

```
# read data
import pyreadr

fault_free_training_data = pyreadr.read_r('TEP_FaultFree_Training.RData')['fault_free_training']
# pandas dataframe
fault_free_testing_data = pyreadr.read_r('TEP_FaultFree_Testing.RData')['fault_free_testing']
faulty_training_data = pyreadr.read_r('TEP_Faulty_Training.RData')['faulty_training']
faulty_testing_data = pyreadr.read_r('TEP_Faulty_Testing.RData')['faulty_testing']
```

The snippet below gives a quick overview of `fault_training_data` dataframe. All four dataframes follow the same column arrangement.

- Column 1 (`faultNumber`) denotes the fault class, ranging from 0 (fault-free) to 20
- Column 2 (`simulationRun`) denotes the simulation run, ranging from 1 to 500
- Column 3 (`sample`) denotes the sampling instances for a simulation run, ranging from 1 to 500 for training data and 1 to 960 for test data
- Columns 45 to 55 contain the sensor measurements

Index	faultNumber	simulationRun	sample	xmeas_1	xmeas_2	xmeas_3	xmeas_4	xmeas_5
0	1	1	1	0.25038	3674	4529	9.232	26.889
1	1	1	2	0.25109	3659.4	4556.6	9.4264	26.721
2	1	1	3	0.25038	3660.3	4477.8	9.4426	26.875
3	1	1	4	0.24977	3661.3	4512.1	9.4776	26.758
4	1	1	5	0.29405	3679	4497	9.3381	26.889
5	1	1	6	0.29303	3691.7	4502.2	9.378	27.111

Before we proceed, we will remove some faults classes from our dataset because these faults are not recognizable⁶⁵.

```
# remove fault classes 3,9,15 from faulty datasets
faulty_training_data = faulty_training_data[faulty_training_data['faultNumber'] != 3]
faulty_training_data = faulty_training_data[faulty_training_data['faultNumber'] != 9]
```

⁶⁵ Chemical Process Fault Detection using deep learning, www.mathworks.com

```

faulty_training_data = faulty_training_data[faulty_training_data['faultNumber'] != 15]
faulty_testing_data = faulty_testing_data[faulty_testing_data['faultNumber'] != 3]
faulty_testing_data = faulty_testing_data[faulty_testing_data['faultNumber'] != 9]
faulty_testing_data = faulty_testing_data[faulty_testing_data['faultNumber'] != 15]

```

Next, we set aside some simulation runs as validation dataset. Moreover, only a small fraction of faulty simulation data is used for training and validation. In typical real-life scenarios, number of faulty instances are few and therefore, it would be unrealistic to assume equal number of fault-free and faulty samples. The code below accomplishes this and generates the final training, validation, and test datasets.

```

# separate validation dataset out of training dataset and create imbalanced faulty dataset
fault_free_validation_data = fault_free_training_data[fault_free_training_data['simulationRun'] > 400]
fault_free_training_data = fault_free_training_data[fault_free_training_data['simulationRun'] <= 400]
faulty_validation_data = faulty_training_data[faulty_training_data['simulationRun'] > 490]
faulty_training_data = faulty_training_data[faulty_training_data['simulationRun'] <= 50]

# convert to numpy
fault_free_training_data = fault_free_training_data.values
fault_free_validation_data = fault_free_validation_data.values
fault_free_testing_data = fault_free_testing_data.values
faulty_training_data = faulty_training_data.values
faulty_validation_data = faulty_validation_data.values
faulty_testing_data = faulty_testing_data.values

# complete training, validation, and test datasets
training_data = np.vstack((fault_free_training_data,faulty_training_data))
validation_data = np.vstack((fault_free_validation_data,faulty_validation_data))
testing_data = np.vstack((fault_free_testing_data,faulty_testing_data))

```

Sensor measurements, which start from 4th column onwards, are the input variables and the fault number in 1st column is our output variable. We separate out the input variables and scale them.

```

# separate X and y data
X_train = training_data[:,3:]
X_val = validation_data[:,3:]
X_test = testing_data[:,3:]

y_train = training_data[:,0]
y_val = validation_data[:,0]
y_test = testing_data[:,0]

```

```
# scale data
from sklearn.preprocessing import StandardScaler

X_scaler = StandardScaler()
X_train_scaled = X_scaler.fit_transform(X_train)
X_val_scaled = X_scaler.transform(X_val)
X_test_scaled = X_scaler.transform(X_test)
```

Now, like we did for system identification example, we need to re-arrange the data into sequence form. In TEP dataset, each simulation run forms a complete sequence. For example, each simulation run in the training dataset with 500 samples is a sequence with 500 time-steps.

```
# rearrange X data into (# sequence samples, # time steps, # features) form
nTimeStepsTrain = 500 # length of a simulation run in training data
nTimeStepsTest = 960 # length of a simulation run in testing data
X_train_sequence = []
y_train_sequence = []
X_val_sequence = []
y_val_sequence = []
X_test_sequence = []
y_test_sequence = []

for sample in range(0, X_train_scaled.shape[0], nTimeStepsTrain):
    X_train_sequence.append(X_train_scaled[sample:sample+nTimeStepsTrain,:])
    y_train_sequence.append(y_train[sample])

for sample in range(0, X_val_scaled.shape[0], nTimeStepsTrain):
    X_val_sequence.append(X_val_scaled[sample:sample+nTimeStepsTrain,:])
    y_val_sequence.append(y_val[sample])

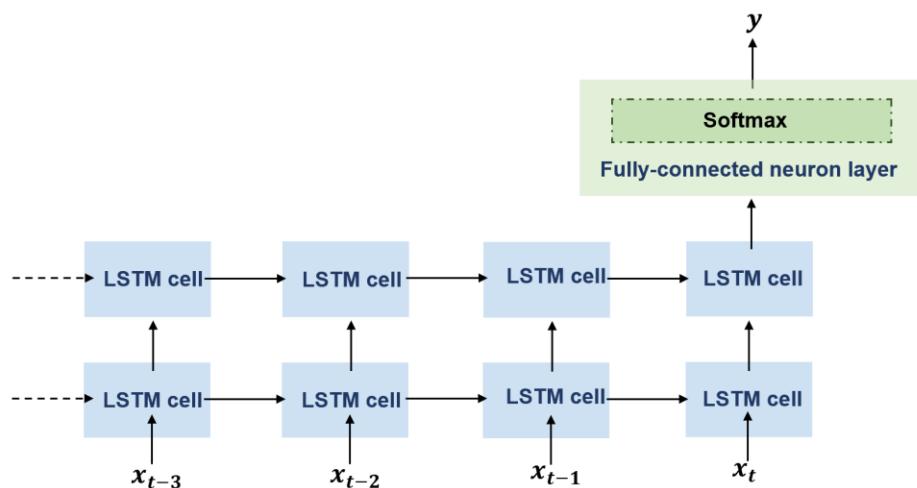
for sample in range(0, X_test_scaled.shape[0], nTimeStepsTest):
    X_test_sequence.append(X_test_scaled[sample:sample+nTimeStepsTest,:])
    y_test_sequence.append(y_test[sample])

X_train_sequence, y_train_sequence = np.array(X_train_sequence), np.array(y_train_sequence)
X_val_sequence, y_val_sequence = np.array(X_val_sequence), np.array(y_val_sequence)
X_test_sequence, y_test_sequence = np.array(X_test_sequence), np.array(y_test_sequence)
```

Before we fit the model, a final pre-processing is done to convert our categorical output labels into the one-hot encoded form. This converts the 1-D output vectors into 21-D matrices.

```
# convert fault class labels to one-hot encoded form
from tensorflow.keras.utils import to_categorical
Y_train_sequence = to_categorical(y_train_sequence, num_classes=21)
Y_val_sequence = to_categorical(y_val_sequence, num_classes=21)
Y_test_sequence = to_categorical(y_test_sequence, num_classes=21)
```

Alright, we are now ready to define our RNN model. Model's topology is shown below. Since the output fault classes are mutually exclusive, we will use softmax output layer. Also, we specify `return_sequences=True` parameter into LSTM to ensure that sequence output is passed from the 1st LSTM layer to the 2nd.



```
# import packages
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras import regularizers

# define model
model = Sequential()
model.add(LSTM(units=128, kernel_regularizer=regularizers.L1(0.0001), return_sequences=True,
input_shape=(nTimeStepsTrain,52)))
# 1st LSTM layer with sequence output
model.add(LSTM(units=64, kernel_regularizer=regularizers.L1(0.0001)))
# 2nd LSTM layer without sequence output
model.add(Dense(21, activation='softmax'))
# Softmax output layer with 21 neurons
```

We can now compile and fit our model. A few noteworthy things in the code below. Categorical_crossentropy loss, which is suitable for multiclass classification, is used. An additional ‘accuracy’ metric is specified which will be tracked (along with ‘loss’) during model fitting.

```
# compile and fit model
from tensorflow.keras.callbacks import EarlyStopping

model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])
es = EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True)
history = model.fit(X_train_sequence, Y_train_sequence, epochs=100, batch_size=250,
validation_data=(X_val_sequence,Y_val_sequence), callbacks=[es])
```

Figure 12.11 shows that almost 100% accuracies are obtained on both training and validation dataset, suggesting the adequacy of our model.

```
# plot validation curve
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='val')
plt.title('Validation Curves: Loss')
plt.xlabel('Epoch'), plt.ylabel('Loss')

plt.plot(history.history['accuracy'], label='train')
plt.plot(history.history['val_accuracy'], label='val')
plt.title('Validation Curves: Accuracy')
plt.xlabel('Epoch'), plt.ylabel('Accuracy')
```

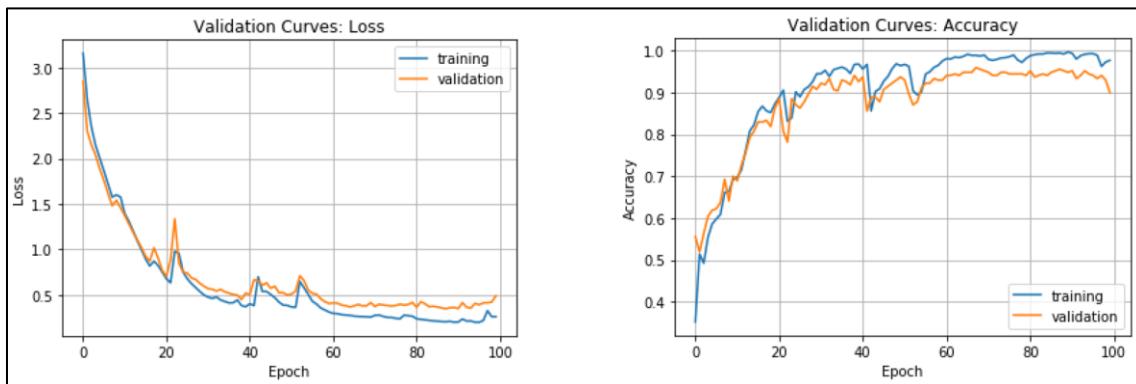


Figure 12.11: Evolution of losses and accuracies during model fitting for TEP fault classification

Let’s see how well we can classify the process faults in test dataset. We will visualize the results using confusion matrix.

```

# model performance on test data
resultsTest = model.evaluate(X_test_sequence, Y_test_sequence)
print("test accuracy:", resultsTest[1])

>>> test accuracy: 0.875

# generate confusion matrix
from sklearn.metrics import confusion_matrix

Y_test_sequence_pred = model.predict(X_test_sequence)
y_test_sequence_pred = np.argmax(Y_test_sequence_pred, axis = 1)
conf_matrix = confusion_matrix(y_test_sequence, y_test_sequence_pred, labels=list(range(21)))

# plot confusion matrix
import seaborn as sn

sn.set(font_scale=1.5) # for label size
sn.heatmap(conf_matrix, fmt='0f', annot=True, cmap='Blues')
plt.ylabel('True Fault Class', fontsize=35)
plt.xlabel('Predicted Fault Class', fontsize=35)

```

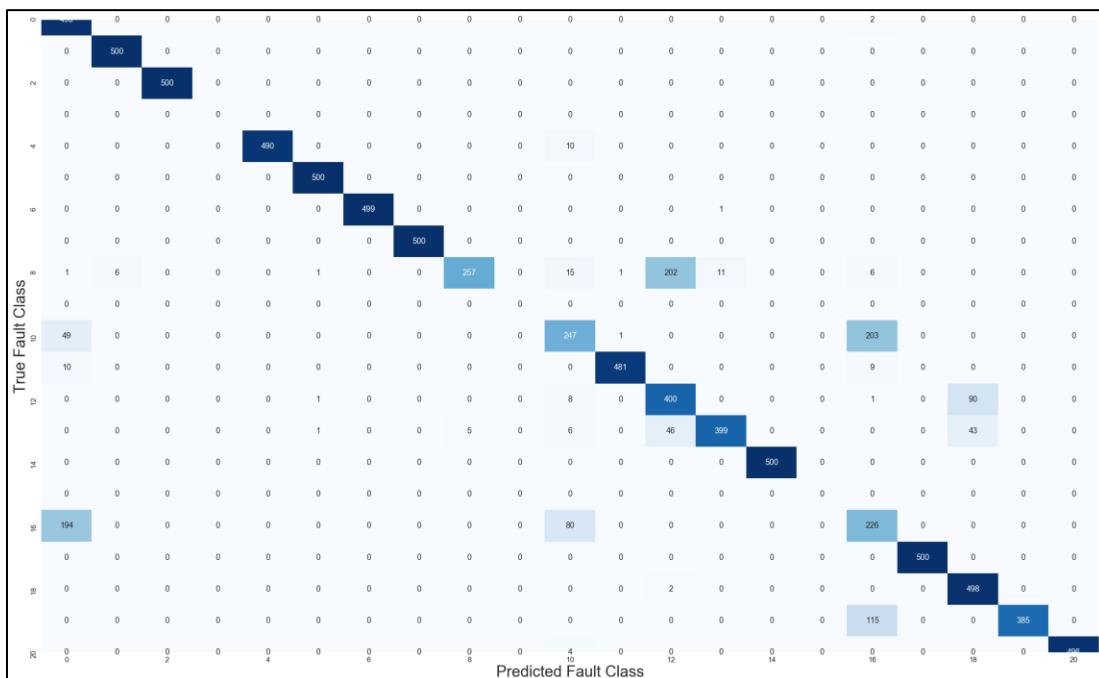


Figure 12.12: Confusion matrix for TEP fault classification with test dataset

Figure 12.12 shows that the RNN model can correctly identify most of the fault classes. It faces some difficulty with classes 8, 10, and 16. For example, several simulations with fault 8

are identified as having fault 12. Overall, the model gives ~88% accuracy. Note that the time-steps in training and test sequence do not have to be the same as shown in this example where each test sequence has 960 time-steps.

Hopefully, this example gave you a good understanding of how to deploy a RNN model for fault classification using dynamic data. In the next section, we will study another very important application of RNNs, predictive maintenance.

12.6 Predictive Maintenance using LSTM Networks

The process-fault detection models that we have built till now have relied on using latest (and recent past) process data to detect process faults that have already occurred. The process reliability personnel get relatively short amount of time to prepare for the maintenance. Wouldn't it be great if our models could predict well in advance (a week or may be a month) the impending failure of any equipment? Such models have been successfully deployed in process industry and are referred to as predictive maintenance (PdM) models.

PdM models use historical past failure data to extract dynamic patterns as indicators of failures in the future. Infact, models can be built to predict the precise failure time in future. The time until failure of an equipment is also referred to as remaining useful life (RUL). RNNs are among the most useful technique available at our disposal to tackle RUL estimation or failure prediction problems.

For illustration, we will use aircraft gas turbine engine dataset which consists of dynamic data 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. Training datasets contain complete data until engine failures, while the test dataset contains data until some point prior to failure. Actual RUL has 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.

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 advance prediction-based preventive maintenance (PdM).

Preventive Maintenance	Condition-based Maintenance	Predictive Maintenance
<ul style="list-style-type: none">Equipment maintenance is done at pre-set scheduleNon-optimal strategy as maintenance may happen too early or too lateE.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 failure are detectedManual expert analysis is carried out to determine exact maintenance planE.g., a compressor may be monitored using PCA to detect the need to repair	<ul style="list-style-type: none">Like CBM, utilizes continuous monitoring of equipmentUtilizes temporal degradation pattern in sensor readings to detect impending failure at very early stageE.g., a compressor may be monitored using RNN to alarm about impending failure

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 12.13, 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 12.13 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.

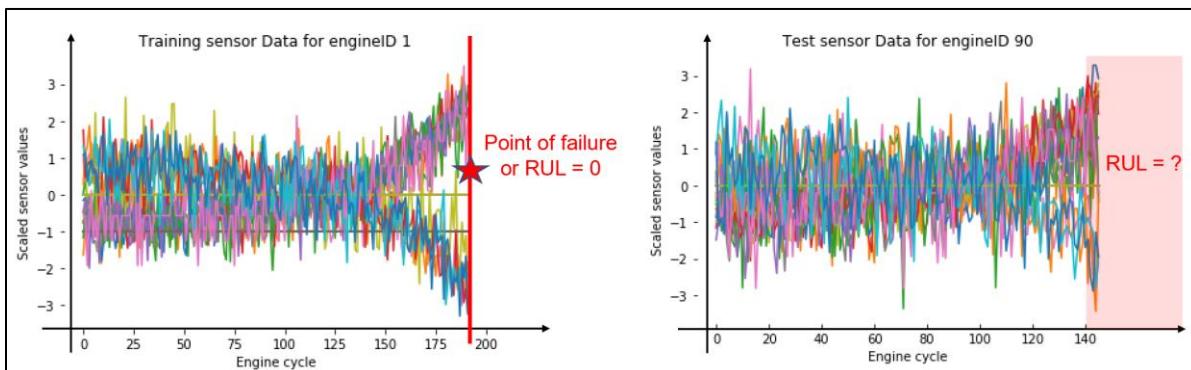


Figure 12.13: Sensor reading from training dataset (left) and test dataset (right). The actual RUL for the shown engine ID 90 is 28 as provided in the RUL_FD001.txt.

Failure prediction using LSTM

In this illustration we will build a LSTM network to predict whether an engine will fail within the next 30 cycles or not. This implies that we have a binary classification problem: will fail (output = 1) or not fail (output = 0). As done by others⁶⁶, 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

⁶⁶ <https://github.com/umbertogriffo/Predictive-Maintenance-using-LSTM>

Next, we will do some dataframe manipulations to compute RUL and failure label at any given cycle for an engine. The code below basically adds the highlighted columns shown in the snippet below to the training and test dataframes.

The diagram illustrates the transformation of a training DataFrame (`train_df`) into a final DataFrame. On the left, the `train_df` DataFrame is shown with columns: Index, EngineID, cycle, OPsetting1, OPsetting2, OPsetting3, s1, and s2. Rows 159 through 164 are highlighted in grey. An arrow points from this DataFrame to a second DataFrame on the right. This second DataFrame has columns: s21, engineRUL, and binaryLabel. Rows 32 through 28 are highlighted in pink. The `s21` column lists values 23.2817, 23.2962, 23.1538, 23.1419, and 23.1761. The `engineRUL` column lists values 0, 0, 1, 1, and 1 respectively. The `binaryLabel` column lists values 0, 0, 1, 1, and 1 respectively.

train_df - DataFrame							
Index	EngineID	cycle	OPsetting1	OPsetting2	OPsetting3	s1	s2
159	1	160	-0.0006	-0.0004	100	518.67	643.45
160	1	161	0.0008	0.0001	100	518.67	643
161	1	162	-0.0005	0.0004	100	518.67	643.15
162	1	163	0.0003	-0.0004	100	518.67	642.85
163	1	164	0.0005	-0.0002	100	518.67	643.17

s21	engineRUL	binaryLabel
23.2817	0	0
23.2962	0	0
23.1538	1	1
23.1419	1	1
23.1761	1	1

```
# 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

w1 = 30
train_df['binaryLabel'] = np.where(train_df['engineRUL'] <= w1, 1, 0)

# 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 & binary label 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)

test_df['binaryLabel'] = np.where(test_df['engineRUL'] <= w1, 1, 0)
```

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','binaryLabel'])
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','binaryLabel']].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','binaryLabel']].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; output label for the sequence is decided based on whether the engine's RUL at the end of sequence is more than 30 or not. 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])
        engine_y_train_sequence.append(engine_data[sample,-1])

    return engine_X_train_sequence, engine_y_train_sequence

```

```

# generate sequence samples
for engineID in train_df_scaled['EngineID'].unique():
    engine_df = train_df_scaled[train_df_scaled['EngineID'] == engineID]
    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', 'binaryLabel']]
    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 is similar to the previous fault classification example except that here we use a single neuron output layer with sigmoid activation function. Moreover, for regularization, we utilize the dropout technique.

```

# import ANN packages
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout

# 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, activation='sigmoid'))

# compile model
model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=['accuracy'])

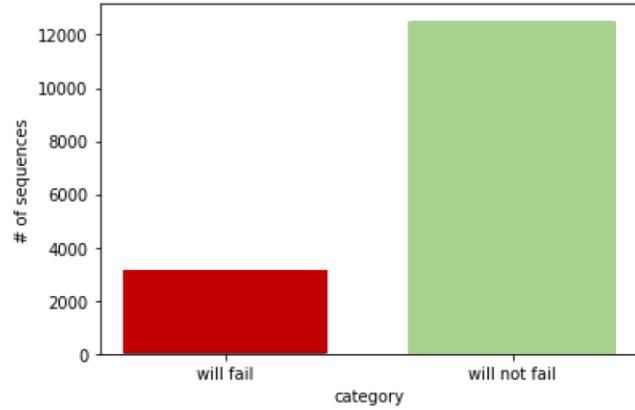
```

Before we fit our RNN, we need to set aside some sequences for validation. If we look at the distribution of class labels in the training sequences, we will notice class imbalance. As a best practice, validation dataset should be chosen with the same class distribution. This is done easily in Sklearn by using the stratify parameter as shown below

```

from sklearn.model_selection import train_test_split
X_est_sequence, X_val_sequence, y_est_sequence, y_val_sequence =
train_test_split(X_train_sequence, y_train_sequence, stratify=y_train_sequence, test_size = 0.3,
random_state = 100)

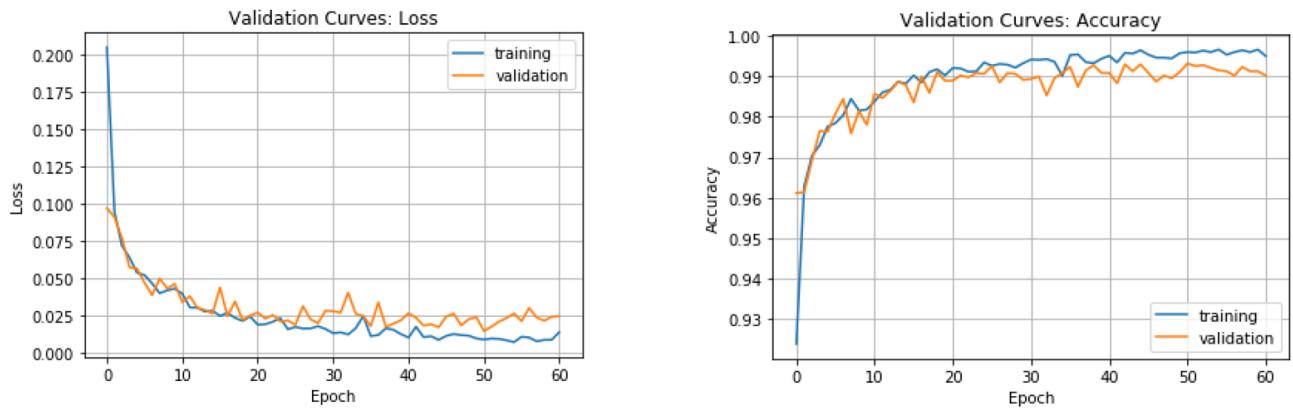
```



We can now fit the model. Validation curves show that we can obtain almost 100% accuracies on training dataset.

```
from tensorflow.keras.callbacks import EarlyStopping

es = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
history = model.fit(X_est_sequence, y_est_sequence, epochs=100, batch_size=250,
validation_data=(X_val_sequence, y_val_sequence), callbacks=[es])
```



For test data, we will create one sequence per engine using the last 50 cycles. Confusion matrix shows that we can correctly predict failure in 84% of the cases. Hyperparameter tuning may lead to even better performance.

```
# input/output test sequences (only the last sequence is used to predict failure)
X_test_sequence = []
y_test_sequence = []

for engineID in test_df_scaled['EnginID'].unique():
    engine_df = test_df_scaled[test_df_scaled['EnginID'] == engineID]
```

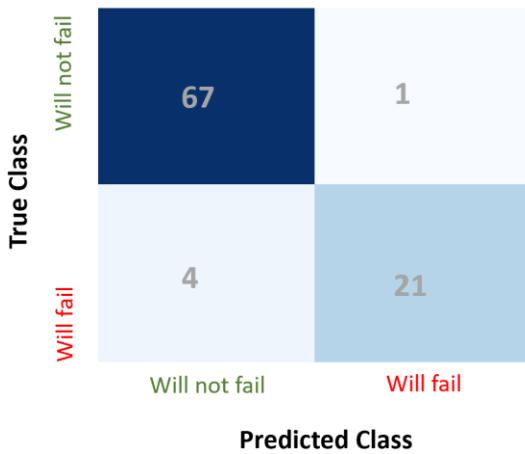
```

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', 'binaryLabel']].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 test labels
y_test_sequence_pred = model.predict(X_test_sequence) > 0.5 # probabilities to binaryLabels
conf_matrix_test = confusion_matrix(y_test_sequence, y_test_sequence_pred)

```



In this current example, we attempted to predict if an engine will fail within the next 30 cycles. In the next subsection, we will try to predict the numeric RUL of each engine directly. The code will largely remain the same except for a few tweaks.

Remaining useful life (RUL) prediction using LSTM

To convert the binary classification model into a regression model for RUL estimation, we will employ the following minor modifications in the previous code. Reader should refer to the complete code provided online.

- Binary labels are not computed
- RUL values are used as target outputs
- Linear activation function is used instead of sigmoid function in output layer

Figure 12.14 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.

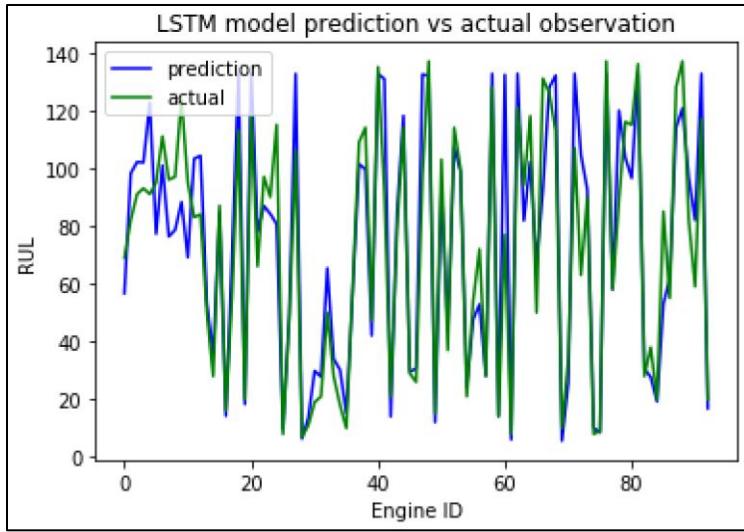


Figure 12.14: Predicted vs observed engine RULs for test aircraft engine dataset

It won't be surprising if you are already thinking about all the different applications where you can use the predictive capabilities of RNNs (or more specifically LSTMs)! Such enthusiasm is not unwarranted, however, remember that a key requirement for a successful predictive solution is the availability of historical database on failure events. Moreover, the available sensor readings should show some degradation pattern as indications of impending failure. As long as these two data requirements are met, you can achieve your predictive objectives.

Summary

In this chapter, we studied the application of ANNs for dynamic/temporal systems. We learnt about the remarkable capabilities of LSTM in extracting patterns from long sequences. We implemented RNNs for a few typical applications in PSEs, viz, system identification, fault classification, and predictive failure prediction. After working out these examples, you would have gained a very good understanding on how to setup the RNN-based solution for your specific problems. In the next chapter, we will study reinforcement learning, which is another powerful ANN-based technique.

Chapter 13

Reinforcement Learning

The ML algorithms we have learnt till now rely upon supply of all modeling-relevant input/output data prior to model training. In contrast, reinforcement training (RL) takes ML a step further and is designed to collect the required training data by itself through interactions with the physical system that it is trying to learn about. Through trial-and-error, an RL model learns what actions to take to accomplish any given task. During training, actions that result in favorable results/rewards get reinforced and after multiple interactions, the model eventually learns an optimal action plan/policy! Sounds impressive, right?

RL mimics how we, humans, learn things (such as riding a cycle) through trial & error and environment interactions. This concept opens up a plethora of potential RL applications. You have probably already heard or seen some of the remarkable feats achieved by RL models such as computers playing games better than the best human players or humanoid robots learning how to run, etc. In this chapter, we will focus on process industry-related applications of RL, specifically for process control.

RL is a very broad and constantly evolving field. There are a lot of RL-specific terminology and concepts. We will declutter the world of RL in this chapter, and you will learn how to setup and solve an RL problem. Specifically, this chapter covers the following topics

- Introduction to RL and RL agents as process controllers
 - Introduction to Q-learning, deep RL, and Actor-Critic architecture
 - Introduction to DDPG algorithm for handling process industry-relevant problems
 - Tank level control using an RL agent
-

13.1 Reinforcement Learning: An Introduction

RL is the branch of ML wherein an agent repeatedly interacts with its environment to learn the best way to accomplish a task or the optimal action policy. Figure 13.1 shows the basic setup of reinforcement learning. As shown, the RL agent receives information about the current state (s_t) of the environment based on which an action (a_t) is decided (as per agent's action policy). As a result of the action, the environment moves to a new state (s_{t+1}) and generates a scalar reward (r_{t+1}) indicating how good was the taken action. Before the agent takes another action, the learning algorithm uses the information ($s_t, a_t, r_{t+1}, s_{t+1}$) to improve its policy and then the cycle continues. Eventually, an optimal policy is obtained that maps environment states to optimal actions such that the total rewards earned till task completion is maximized. Once trained, the learning process can be stopped, and the policy function can be deployed.

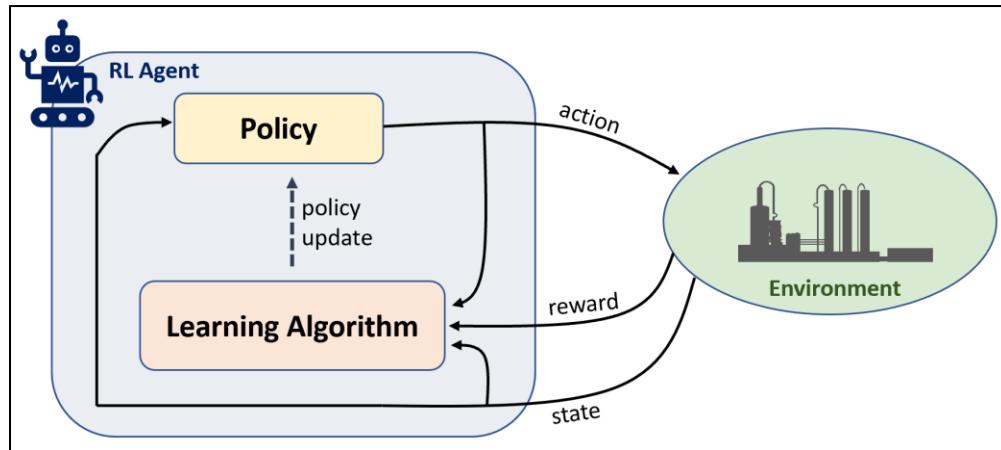


Figure 13.1: Reinforcement learning setup depicting an agent's interactions with its environment

A simple real-life analogy could be the task of finding the optimal driving route from your office to home in a new town. Here, you are the agent and environment would comprise of your car, city's roads and highways network, traffic, weather, your geospatial location, and basically everything excluding you. The total reward to maximize could be the negative of the time taken to reach home (less driving time is better). During driving, depending on the environment state, you would take decisions on whether to take any highway exit or make any turn or not. Assuming no internet (and google maps!), being new in town, you would not know if taking those exits or turns will help you reach home faster and therefore you will explore different possible routes. After several trials, you would gain a good understanding of the town and eventually would be able to take the most optimal action for any given environment state at any point during driving. RL follows the same methodology to find the optimal mapping using some systematic (and very smart) learning algorithms.

RL for process control

The RL framework of observing an environment's current state and deciding upon corresponding optimal actions fits quite naturally to the setup of process controllers in process industry. You might already know that these controllers are tasked with keeping process operations optimal by keeping critical process variables (product purity, fluid level, temperatures, etc.) at optimal values through adjustments in manipulated variables (stream flows, valve openings, etc.). Figure 13.2 shows the typical setup of a process controller and the division of this setup into agent/environment framework. Rewards are designed as some suitable combination of reference signal, environment state, and controller action.

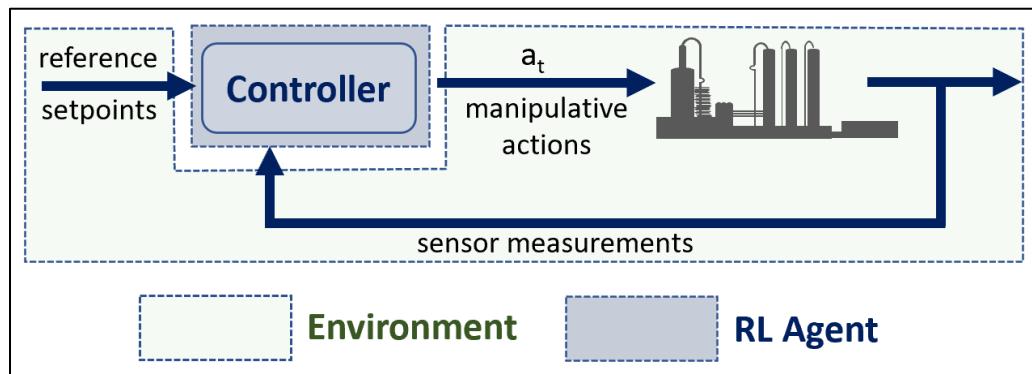


Figure 13.2: Process control system in RL framework

In process industry, although PID and MPC controllers are well-established, their shortcomings are well-known. While PID controllers perform unsatisfactorily for complex nonlinear systems, MPCs solve optimization problems using process models which makes online action computation infeasible for large-scale nonlinear systems. Moreover, both these controllers suffer performance degradation issues (due to changing process conditions, process drift) over time, necessitating regular maintenance. Controller maintenance entails re-identification of process models which can be time and resource intensive and may require interference to normal plant operations for training data collection.

Given the aforementioned issues with the current state-of-the-art for the process controllers and the recent successes of RL, interest in leveraging RL technology for process control has been reignited. Several recent studies⁶⁷ have demonstrated how RL-based controllers can provide superior performance. Not requiring online optimization (because optimal action policy is pre-computed), easy adaptation of action policy under changing environment by

⁶⁷ Cassol et. Al., Reinforcement learning applied to process control: A Van der Vusse reactor case study, Computer Aided Chemical Engineering, 2018

Rajesh Siraskar, Reinforcement learning for control of valves, Machine Learning with Applications, 2021

Ma et. Al., Continuous control of a polymerization system with deep reinforcement learning, Journal of Process Control, 2019

continued learning with new process data are some characteristics that make RM-based controllers very promising.



You may be slightly concerned about training an RL agent in the real plant environment because the generated actions in early stages of training can be ‘very bad’ and even unsafe. Moreover, for complex system, thousands of interactions may be required to reach even a reasonably good policy; plant managers will certainly never agree to this! These are valid concerns and therefore, the common practice is to use a sufficiently accurate model of the plant and train RL agent offline in a simulated environment. Once offline learning is complete, the learning process can be turned off and RL agent deployed in real plant. There are, however, a few good reasons to keep learning on (continually or sporadically) post-deployment. First, your plant model will probably not be 100% accurate. Therefore, the RL agent may use some online interactions to fine-tune its policy. Second, as alluded to before, the plant behavior may change over time and the agent will need to tweak its policy to re-adjust to changes in its environment.

This discussion also brings us to the point of model-free vs model-based RL. Model-free RL doesn’t use any environment/plant model during training and learns its policy based solely on its interactions with real environment. Model-based RL, on the other hand, uses a model either for simulating the environment or assisting the learning algorithm.

13.2 RL Terminology & Mathematical Concepts

We have showered enough praises on RL. Let’s now get down to understanding how RL actually works. For this, we will first learn some RL terminology and concepts. A quick disclaimer here that some of these new concepts may seem ‘abstract’ and not immediately useful. But, as you read through this chapter, we promise that all dots will connect and their utility will become ‘obvious’.

Environment and Markov decision process

While informally an environment is anything that an RL agent interacts with, in a formal setting, it is represented as a Markov decision process (MDP). A MDP is named such because it exhibits Markov property, i.e., the (probability of) transition from state s_t to s_{t+1} depend only

on the most recent state (s_t) and action (a_t). Such memoryless characteristic is important in RL because it allows the agent to only consider the current state when deciding the next optimal action and not worry about what actions were previously taken to reach s_t .

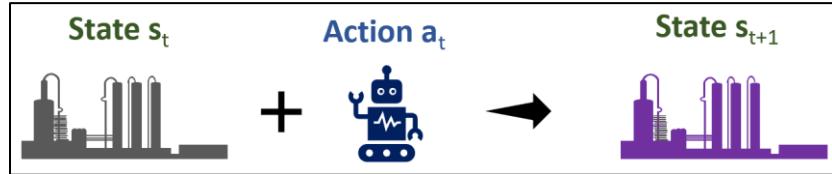
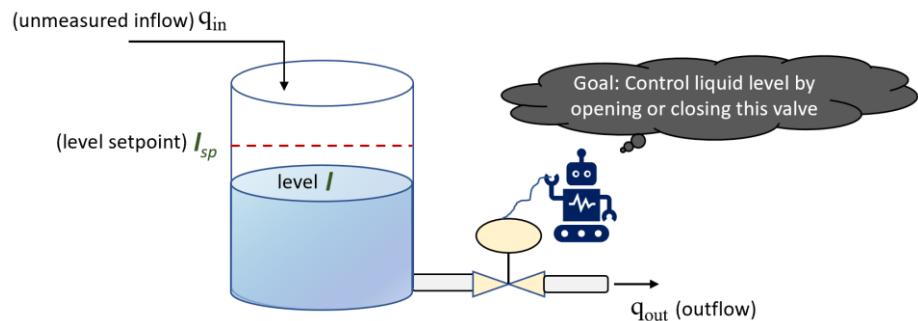


Figure 13.3: Transition in an MDP depend only on current state and action

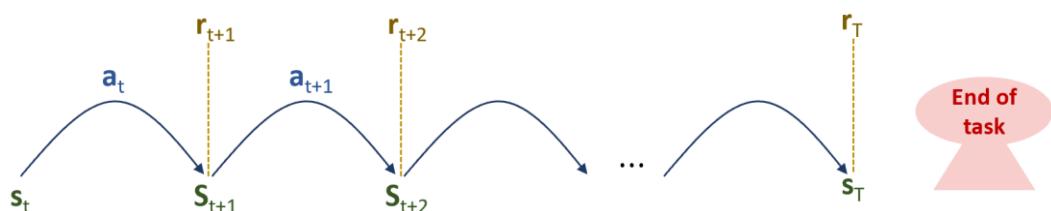
In an MDP, selection of features that completely characterize the state of the environment and enable the agent to act on its basis becomes crucial. For example, consider the following problem of controlling the liquid level in a tank



Here, l , l_{sp} , and the rate of change of l could be used to define the environment state. l_{sp} can also be substituted with $l - l_{sp}$ as an alternative formulation. Depending on your problem formulation, data from the past may also be included in the state vector to facilitate the agent's decision making.

Reward and return

Upon execution of an action (a_t), the RL agent receives a reward signal (r_{t+1}) which quantifies and informs the agent the goodness of being in the new state (s_{t+1}) and taking action a_t . Consider the following sequence of an agent's interactions with its environment from time t onwards during the execution of a task.



The agent's job is to take actions that maximize the cumulative reward (also called return) over the course of the task. The return can be represented as $R_t = \sum_{k=1}^{T-t} r_{t+k}$. However, this direct summation can cause trouble for tasks which don't have a definite end state. The level control task is one such example where $T=\infty$ (because the task is to control the level forever!). To keep R_t bounded, the following formulation is used instead

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots + \gamma^{T-t-1} r_T \\ = \sum_{k=1}^{T-t} \gamma^{k-1} r_{t+k} \quad \text{eq. 1}$$

The return in Eq. (1) is called discounted return and $\gamma \in (0,1)$. The farther the reward in the future, less importance is accorded to it. A γ close to 0 makes an agent consider only the immediate reward (r_{t+1}) for choosing a_t while γ close to 1 makes long-term rewards important.

The concept of discounted reward maximization is an elegant (and mathematically tractable) way of making RL agent look at both immediate and future rewards. For level control problem, a natural reward function is squared tracking error

$$r_t = -(l_t - l_{sp})^2$$

Appropriate crafting of reward function is very important as it strongly influences the agent's decisions and RL training convergence.

Policy

A policy is a rule or mapping used by an RL agent to determine a_t given s_t . A policy can be deterministic or stochastic. A deterministic policy provides a single action

$$a_t = \mu(s_t)$$

while a stochastic policy (π) provides a probability over the set of actions

$$\pi(a|s_t) = \text{probability that action } a \text{ is taken in state } s_t$$

The goal in RL is to learn a policy which provides maximum return. The optimal policy is often denoted as μ^* or π^* .

Value function

RL algorithms often make use of value functions. State value function, $V_\mu(s)$, gives the return an agent expects to receive from being in the state s and acting according to some policy μ .

$$V_\mu(s) = E[R_t | s_t = s, \text{policy } \mu]$$

State-action value function, $Q_\mu(s, a)$, is another useful function that gives the expected return from taking an arbitrary action (not necessarily following policy μ) from state s and then following policy μ .

$$Q_\mu(s, a) = E[R_t | s_t = s, a_t = a, \text{policy } \mu]$$

When the optimal policy, μ^* , is followed, the value functions are denoted as $V^*(s)$ and $Q^*(s, a)$. In above definitions, we take expectations ($E[\cdot]$) of the returns because real-life environments are uncertain and stochastic, i.e., taking an action a_t at state s_t can result in slightly different s_{t+1} in different iterations. This transition uncertainty can be denoted as the probability $P(s_{t+1} = s' | s_t = s, a_t = a)$. Above definitions incorporate this transition probability during return computation.

Bellman equation

The value functions can also be represented in a recursive form. Consider the optimal state function,

$$\begin{aligned} V^*(s) &= E[R_t | s_t = s, \mu^*] \\ &= E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots | s_t = s, \mu^*] \\ &= E[r_{t+1} + \gamma Q^*(s_{t+1}, \mu^*) | s_t = s, \mu^*] \\ &= E[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, \mu^*] \end{aligned}$$

The optimal state-action value can also be similarly re-written as

$$\begin{aligned} Q^*(s, a) &= E[r_{t+1} + \gamma Q^*(s_{t+1}, \mu^*(s_{t+1})) | s_t = s, a_t = a] \\ \text{or} \\ Q^*(s, a) &= E \left[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a \right] \end{aligned}$$

Note again that $s_{t+1} \sim P(s_{t+1} = s' | s_t = s, a_t = a)$ and is sampled following the environment's transition probability. The above recursive equations, also called Bellman equations, are central to several RL algorithms. Bellman equation simply states that the value of the current state (or state-action) is the sum of immediate reward and the value of whatever state the environment lands next.

Alright, we now have enough fundamentals in place to start taking a deeper look into RL algorithms for optimal policy learning. Let's begin with one of the most popular technique, Q-learning.

13.3 Fundamentals of Q-learning

Q-learning refers to the RL algorithms designed for estimating the $Q^*(s, a)$ values. Once the optimal value function is available, an optimal policy can be framed as simply picking the action corresponding to the highest value for the current state as shown below

$$a_t = \arg \max_{a'} Q^*(s_t, a)$$

To see how Q^* may be obtained, consider an MDP with discrete state and action space. We can represent the Q function as a Q-table⁶⁸

	s_1	s_2	s_3	...
a_1	$Q^*(s_1, a_1)$	$Q^*(s_2, a_1)$	$Q^*(s_3, a_1)$	
a_2	$Q^*(s_1, a_2)$	$Q^*(s_2, a_2)$	$Q^*(s_3, a_2)$	
a_3	$Q^*(s_1, a_3)$	$Q^*(s_2, a_3)$	$Q^*(s_3, a_3)$	
⋮				

To find the optimal values for each state/action pair, all the values in Q-table are first initialized to 0, some initial state is arbitrarily chosen, agent-environment interactions start, and then the bellman equation is used in the following iteration form

$$\widehat{Q}^*_{new}(s_t, a_t) \leftarrow (1 - \alpha)\widehat{Q}^*_{old}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a'} \widehat{Q}^*_{old}(s_{t+1}, a') \right)$$

or

$$\widehat{Q}^*_{new}(s_t, a_t) \leftarrow \widehat{Q}^*_{old}(s_t, a_t) + \alpha \underbrace{\left(r_{t+1} + \gamma \max_a \widehat{Q}^*_{old}(s_{t+1}, a') - \widehat{Q}^*_{old}(s_t, a_t) \right)}_{\text{Temporal difference (TD) error}} \quad \text{eq. 2}$$

where $0 \leq \alpha \leq 1$ is a learning-rate hyperparameter. Eq. 2 is a remarkably simple formula, and it has been shown that given sufficient iterations, the estimated Q values will converge to the

⁶⁸In RL literature, states are usually shown along the rows in a Q-table. We have placed states along the columns purely for convenience in the simple illustration later.

optimal values! In Eq. 2, selection of a_t is an important consideration and ε -greedy scheme is usually adopted. In this scheme, at every iteration, the agent acts randomly with probability ε or greedily (chooses action corresponding to the highest value \widehat{Q}^* for a given s_t) with probability $1 - \varepsilon$. This scheme allows good combination of exploitation (taking optimal actions based on current estimates, allowing the estimates to get better over time) and exploration (visiting unknown but potentially rewarding parts of the environment). During initial iterations when \widehat{Q}^* estimates are expectedly poor, ε is given a high value (close to 1) and reduced gradually as training completes. The pseudo-code below summarizes the tabular Q-learning algorithm.

Algorithm 1: Tabular Q-learning

1. Set $\widehat{Q}^*(s, a) = 0$ for all state-action pairs
2. for episode = 1 to N_{episode} do:
3. initialize state s
4. for each step (t) of episode until s is terminal do:
5. choose action (a_t) for s_t using ε -greedy policy
6. take action a_t and observe r_{t+1} and next state s_{t+1}
7. compute δ (TD error) = $r_{t+1} + \gamma \max_{a'} \widehat{Q}^*_{\text{old}}(s_{t+1}, a') - \widehat{Q}^*_{\text{old}}(s_t, a_t)$
8. $\widehat{Q}^*_{\text{new}}(s_t, a_t) \leftarrow \widehat{Q}^*_{\text{old}}(s_t, a_t) + \alpha \delta$
9. end for
10. end for

In the pseudo-code above, there are two terms whose meaning may not be immediately clear to you, namely, ‘episode’ and ‘terminal state’. An episode simply refers to one complete simulation or experiment. For example, one instance of a drive from your office to home would be one episode. A terminal state marks the end of an episode. In the driving example, arrival at home or end of fuel could be the terminal states. For tasks that may not necessarily end in a terminal state (such as controlling the fluid level), a maximum number of steps are pre-specified for an episode during training.

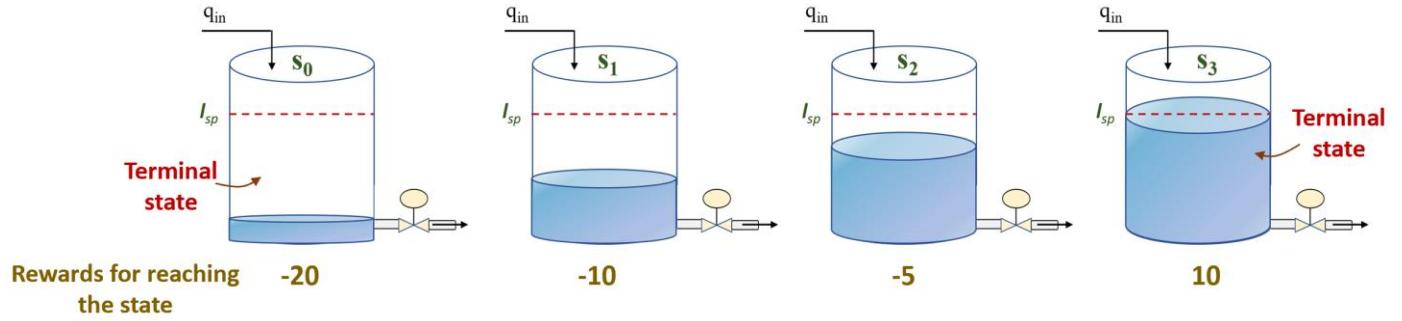


Off-policy vs on-policy algorithms

Q-learning belongs to the category of off-policy RL algorithms. In such algorithms the target policy being learnt (optimal policy) is different from the policy being executed or used to select actions (such as ε -greedy policy) during training. The executed policy is also called behavior policy. For on-policy RL algorithms, behavior policy equals target policy.

Simple illustration

If you are not yet entirely convinced about how taking some random actions can help an agent arrive at an optimal Q-table and an optimal policy, then let's consider a hypothetical control problem. We will take our level control problem and consider only the following 4 symbolic states



RL agent here has 2 action choices - close or open the valve by a fixed amount. Closing a valve takes the environment to the higher fluid level state; the agent of course does not know this to begin with. We will set $\alpha = 1$ and $\gamma = 0.9$. A random action is taken if both actions have same value, otherwise greedy action is taken ($\varepsilon = 0$). Let's now run some mock experiments.

Episode	t	s_t	\widehat{Q}^*_{old}	a_t	s_t	r_{t+1}	\widehat{Q}^*_{new}
1	1	s_1	Open	Close	s_2	-5	Open
			Close	Open			Close
1	2	s_2	Open	Close	s_3	10	Open
			Close	Open			Close

Detailed description: The table tracks two episodes. In Episode 1, Step 1, the state is s_1 . The Q-table (\widehat{Q}^*_{old}) shows all actions (Open, Close) leading to s_2 with a value of 0. The agent chooses Close (randomly chosen). The next state is s_2 and the reward is -5. The TD update for the Close action in the Q-table is $0 + 1 * [-5 + 0.9 * 0 - 0] = -5$. The new Q-table (\widehat{Q}^*_{new}) shows Close leading to s_2 with a value of -5 and Open leading to s_3 with a value of 0. In Episode 1, Step 2, the state is s_2 . The Q-table shows Close leading to s_3 with a value of -5 and Open leading to s_3 with a value of 0. The agent chooses Close (randomly chosen). The next state is s_3 and the reward is 10. The TD update for the Close action in the Q-table is $0 + 1 * [10 + 0.9 * 0 - 0] = 10$. The new Q-table shows Close leading to s_3 with a value of 10 and Open leading to s_3 with a value of 0.

Since the agent has reached a terminal state (s_3), next episode will start. Initial state will be reset but the Q-table will remain intact.

Episode	t	s_t	\widehat{Q}^*_{old}	a_t	s_t	r_{t+1}	\widehat{Q}^*_{new}																				
2	1	s_1	<table border="1"> <tr><td>Open</td><td>s_0</td><td>s_1</td><td>s_2</td><td>s_3</td></tr> <tr><td>Close</td><td>0</td><td>-5</td><td>10</td><td>0</td></tr> </table>	Open	s_0	s_1	s_2	s_3	Close	0	-5	10	0	open (greedy action)	s_0	-20	<table border="1"> <tr><td>Open</td><td>s_0</td><td>s_1</td><td>s_2</td><td>s_3</td></tr> <tr><td>Close</td><td>0</td><td>-20</td><td>0</td><td>0</td></tr> </table>	Open	s_0	s_1	s_2	s_3	Close	0	-20	0	0
Open	s_0	s_1	s_2	s_3																							
Close	0	-5	10	0																							
Open	s_0	s_1	s_2	s_3																							
Close	0	-20	0	0																							
3	1	s_1	<table border="1"> <tr><td>Open</td><td>0</td><td>-20</td><td>0</td><td>0</td></tr> <tr><td>Close</td><td>0</td><td>-5</td><td>10</td><td>0</td></tr> </table>	Open	0	-20	0	0	Close	0	-5	10	0	Close (greedy action)	s_2	-5	<table border="1"> <tr><td>Open</td><td>0</td><td>-20</td><td>0</td><td>0</td></tr> <tr><td>Close</td><td>0</td><td>4</td><td>10</td><td>0</td></tr> </table>	Open	0	-20	0	0	Close	0	4	10	0
Open	0	-20	0	0																							
Close	0	-5	10	0																							
Open	0	-20	0	0																							
Close	0	4	10	0																							

Before we continue with any further iterations, let's pause and observe what we have achieved in just 4 iterations. The current \widehat{Q}^* is already good enough to give the desired optimal policy! In state s_1 , RL agent will close the valve because of higher expected return ($4 > 20$). In state s_2 , agent will again close the valve. With further iterations, the \widehat{Q}^* values will get closer to more accurate estimates, but the optimal policy will remain the same.

The above illustration should have made the Q-learning algorithm clear to you. It is simple to implement yet is quite powerful. A direct industrial application of Q-learning can be seen in the work of Syafiie et al. for automatic control of pH neutralization processes⁶⁹.

⁶⁹ Syafiie et al., Model-free control of neutralization processes using reinforcement learning, Engineering Applications of Artificial Intelligence, 2007

RL Taxonomy

Q-learning, although a breakthrough RL algorithm, is just one of the many RL algorithms currently out there. Because many of them borrow ideas from each-other, it is difficult to list and bucket all of them comprehensively into some reasonable categories. Nonetheless, the figure below presents a non-comprehensive picture of the current RL algorithm landscape.

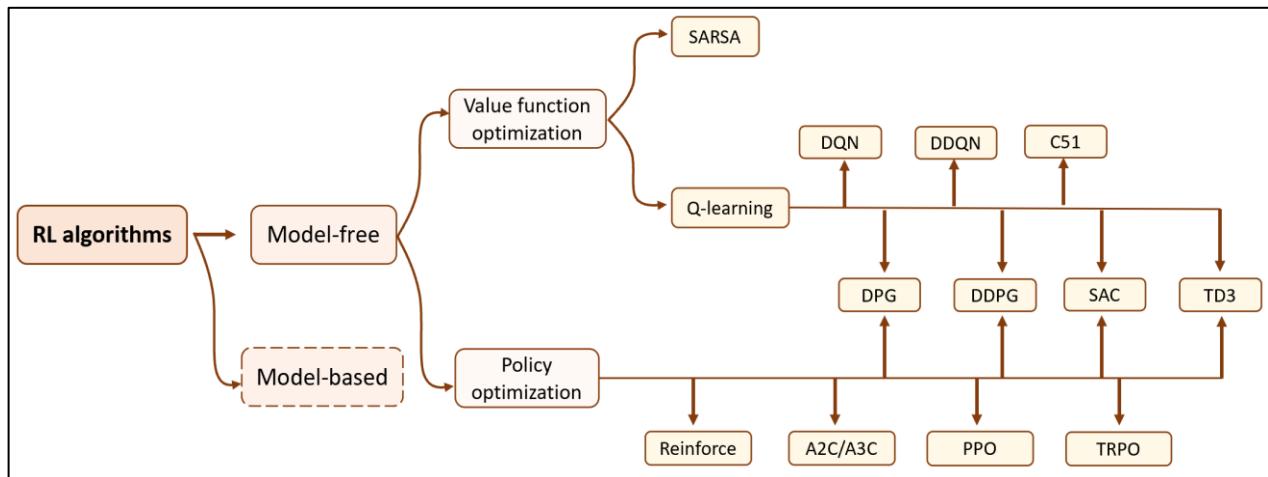


Figure 13.4: Non-comprehensive taxonomy of modern model-free RL algorithms

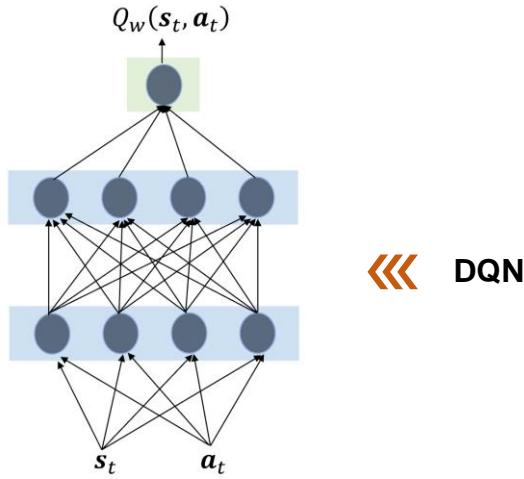
Figure 13.4 focusses on model-free algorithms as these are more popular due to their ease of implementation. Another broad categorization shown is in terms of whether an agent tries to learn an optimal policy directly or indirectly through estimation of Q^* first. Extremely powerful algorithms have recently emerged by combining certain features of policy optimization and Q-learning algorithms. Among these DDPG has been used in several promising studies for RL-based process control and therefore is covered later in the chapter.

13.4 Deep RL & Actor-Critic Framework

While tabular Q-learning is an elegant method, it is unfortunately not suitable for MDPs with high-dimensional continuous state and/or action spaces. Consider the fluid level control problem with fluid level as its state and valve adjustments as action. Here, fluid level and valve opening can attain any value from 0% to 100%. For tabular Q-learning, we may discretize these variables crudely into 100 intervals each. This already leads to a 100 X 100 sized Q-table. With finer discretization and more states/actions, the Q-table can become very large and unmanageable. Unfortunately, most process control problems would face these issues.

Deep Q-learning

One solution to continuous space problems is to use a function that approximates the Q-table and returns the Q-value for any state-action pair. When a deep neural network is used as the function (as shown below), the methodology is called deep Q-learning and the neural network is called deep Q-network (DQN).



Like tabular Q-learning, the parameters, \mathbf{w} , of Q-network are updated iteratively such that the output $Q_w(s_t, a_t)$ gets closer to its target value. But what's the target value when the optimal $Q^*(s_t, a_t)$ is unknown? The Bellman equation is used to provide what is called a TD target or bootstrap⁷⁰ target (\tilde{y}_t). Let $(\mathbf{s}_t, \mathbf{a}_t, r_{t+1}, \mathbf{s}_{t+1})$ be a transition tuple obtained via environment interaction at time t , then

$$\tilde{y}_t = r_{t+1} + \gamma \max_{\mathbf{a}'} Q_w(\mathbf{s}_{t+1}, \mathbf{a}')$$

The above TD target is used to derive a network parameter update mechanism as shown below

$$L_t(\mathbf{w}) = \frac{1}{2} (\tilde{y}_t - Q_w(\mathbf{s}_t, \mathbf{a}_t))^2$$

$$\frac{\partial L_t}{\partial \mathbf{w}} \approx -(\tilde{y}_t - Q_w(\mathbf{s}_t, \mathbf{a}_t)) \frac{\partial Q_w(\mathbf{s}_t, \mathbf{a}_t)}{\partial \mathbf{w}}$$

$$\mathbf{w}_t \leftarrow \mathbf{w}_t - \alpha \frac{\partial L_t}{\partial \mathbf{w}}$$

Loss function
Although \tilde{y}_t depends on \mathbf{w}_t , this dependence is ignored for this partial derivative
Parameter update

The shown loss function is also called mean squared Bellman error (MSBE) which quantifies the error in satisfying the Bellman equation. This update scheme should be familiar to you

⁷⁰ In RL literature, the practice of updating value functions using other value estimates is called bootstrapping

because we had seen this in Chapter 11 for ANNs. The pseudo-code below summarizes the basic form⁷¹ of DQN algorithm

Algorithm 2: Deep Q-learning

1. Initialize network parameters, w , with random values
2. for episode = 1 to N_{episode} do:
3. initialize state s
4. for each step (t) of episode until s is terminal do:
5. choose action (a_t) for s_t as per some behavior policy $\beta(\cdot | s_t)$
6. take action a_t and observe r_{t+1} and next state s_{t+1}
7. $\tilde{y}_t = r_{t+1} + \gamma \max_{a'} Q_w(s_{t+1}, a')$
8. $w \leftarrow w + \alpha (\tilde{y}_t - Q_w(s_t, a_t)) \frac{\partial Q_w(s_t, a_t)}{\partial w}$
9. end for
10. end for

While DQNs have given remarkable results⁷², their convergence is not guaranteed⁷³. Moreover, DQNs are not well suited for process control problems which have continuous action spaces as the computation of $\max_{a'} Q_w(s_{t+1}, a')$ in each step for continuous variable a' is quite inconvenient, if not impractical. Note that the behavior policy may include solving an optimization for choosing a_t as well.



Although we will see a detailed RL-based process control case study using DDPG algorithm in the next section, it is still useful to conceptually understand DQNs and policy gradient methods. These are precursors to more modern techniques and understanding them will throw crucial insights into DDPG's algorithm.

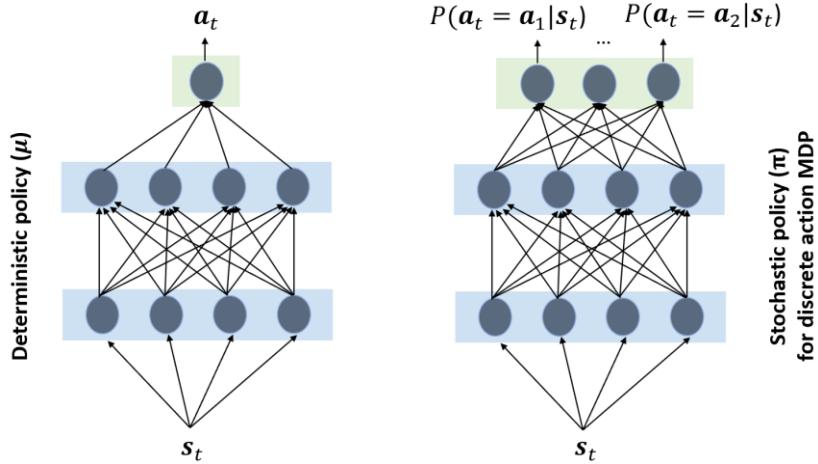
Policy gradient methods

Different from Q-learning, there are another class of algorithms that attempt to learn a function that can directly serve as an optimal policy (either deterministic or stochastic) function as shown below with neural networks-based functions.

⁷¹ Without considering replay memory and target networks. These are introduced in the next section on DDPG.

⁷² Mnih et. al, Playing Atari with deep reinforcement learning, *arXiv preprint*, 2013

⁷³ The combination of Q function approximation, bootstrapping, and off-policy behavior is known as the ‘deadly triad’ and is known to cause divergence



The algorithm behind learning of these policy functions is simple: during training, all the rewards obtained during an episode are collected and then the model parameters (θ) are updated so as to increase returns in subsequent episodes. When the update mechanism makes use of the gradient of model's performance w.r.t θ , the algorithm is called policy-gradient algorithm. The issue with these algorithms is that model training is very slow due to slow convergence. This is partly due to the fact that update occurs at end of episodes (unlike in Q-learning where updates occur at each step) due to the requirement of cumulative reward or return.

Actor-Critic framework

The strength and weakness analysis of DQN and policy-gradient algorithms show that they are kind of complementary to each-other in the sense that weakness of one is strength of the other. Policy-gradient algorithms don't have to solve any optimization problem at every iteration, while DQNs don't wait for the end of an episode for an update. RL researchers saw this complementarity and proposed an actor-critic (AC) framework (Figure 13.5) that merges the two algorithms.

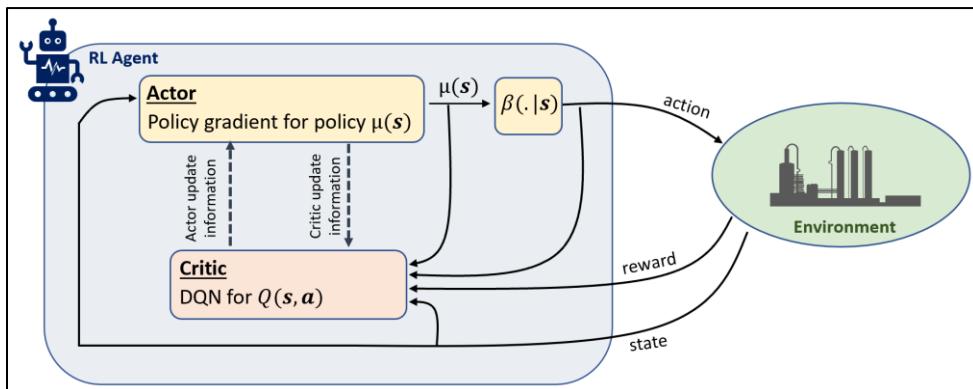


Figure 13.5: RL agent training within AC framework. Note that actor and critic can take any functional form and not necessarily be neural networks.

During any training step, the critic uses the actor to help evaluate $\max_a Q_w(s_{t+1}, a')$ approximately as $Q_w(s_{t+1}, \mu_\theta(s_{t+1}))$ and therefore avoids the difficult optimization; the actor uses the critic to help evaluate whether its predicted action $\mu_\theta(s_t)$ is good or bad via $Q_w(s_t, \mu_\theta(s_t))$ and therefore updates itself immediately instead of waiting till episode's end to know if its actions are leading to higher return or not. This is how actor and critic help each other. Let's look at the update mechanism in more detail.

AC update mechanism

Let the actor and critic be neural networks with model parameters, ω and θ , respectively. The critic update is same as that in deep Q-learning except the highlighted part below

$$\begin{aligned}\tilde{y}_t &= r_{t+1} + \gamma Q_w(s_{t+1}, \mu_\theta(s_{t+1})) \\ w &\leftarrow w + \alpha_c (\tilde{y}_t - Q_w(s_t, a_t)) \frac{\partial Q_w(s_t, a_t)}{\partial w}\end{aligned}$$

For actor network, we will use the rationale that when an action is predicted $a = \mu_\theta(s_t)$, the objective is to maximize the return

$$\begin{aligned}J(\theta) &= E[R_t] && \xleftarrow{\quad\quad\quad} \text{Actor's objective} \\ J(\theta) &\approx Q_w(s_t, a)|_{a=\mu_\theta(s_t)} && \xleftarrow{\quad\quad\quad} \text{Critic provides the performance objective} \\ \Rightarrow \frac{\partial J}{\partial \theta} &= \frac{\partial Q_w(s_t, a)}{\partial a}|_{a=\mu_\theta(s_t)} \frac{\partial \mu_\theta(s_t)}{\partial \theta} && \xleftarrow{\quad\quad\quad} \text{Chain rule from calculus} \\ \theta &\leftarrow \theta + \alpha_\theta \frac{\partial J}{\partial \theta}\end{aligned}$$

Note that since the actor's objective is to maximize the return, it performs gradient ascent, unlike the critic which does a gradient descent to minimize its TD error. Pseudo code in Algorithm 3 summarizes the AC algorithm.

Algorithm 3: Deterministic off-policy AC learning

1. Initialize network parameters, \mathbf{w} and $\boldsymbol{\theta}$, with random values
2. for episode = 1 to N_{episode} do:
3. initialize state s
4. for each step (t) of episode until s is terminal do:
5. choose action (\mathbf{a}_t) for s_t as per some behavior policy $\beta(\cdot | s_t)$
6. take action \mathbf{a}_t and observe r_{t+1} and next state s_{t+1}
7. $\tilde{y}_t = r_{t+1} + \gamma Q_{\mathbf{w}}(s_{t+1}, \mu_{\boldsymbol{\theta}}(s_{t+1}))$
8. $\mathbf{w} \leftarrow \mathbf{w} + \alpha_c (\tilde{y}_t - Q_{\mathbf{w}}(s_t, \mathbf{a}_t)) \frac{\partial Q_{\mathbf{w}}(s_t, \mathbf{a}_t)}{\partial \mathbf{w}}$
9. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} \frac{\partial Q_{\mathbf{w}}(s_t, \mathbf{a})}{\partial \boldsymbol{\theta}}|_{\mathbf{a}=\mu_{\boldsymbol{\theta}}(s_t)} \frac{\partial \mu_{\boldsymbol{\theta}}(s_t)}{\partial \boldsymbol{\theta}}$
10. end for

With the fundamentals of AC algorithm (and the underlying mathematics that powers it) in place, we are ready to learn about the DDPG algorithm which was among the first RL algorithm suitable for complex industrial process control problems.

13.5 Deep Deterministic Policy Gradient (DDPG)

In 2016, Lillicrap et al.⁷⁴ published an actor-critic, model-free algorithm, DDPG, that could successfully handle MDPs with high dimensional continuous state and action spaces. Although it was based on the previously shown AC architecture (with both critic and actor as neural networks), it brought together several innovative ideas from other works (Silver et al.⁷⁵, Mnih et al.⁷⁶, etc.) which greatly improved the robustness and stability of RL agent training. These innovations include:

- ❖ Replay buffer or replay memory
- ❖ Target networks
- ❖ Ornstein-Uhlenbeck process-based behavior policy
- ❖ Batch normalization

⁷⁴ Lillicrap et al., Continuous control with deep reinforcement learning, *arXiv*, 2016

⁷⁵ Silver et al., Deterministic policy gradient algorithms, *ICML*, 2014

⁷⁶ Mnih et al., Human level control through deep reinforcement learning, *Nature*, 2015

Figure 13.6 shows how and where DDPG incorporates these innovations into the AC framework of Figure 13.5. Don't worry if this figure looks confusing. We will dissect these new concepts one-by-one.

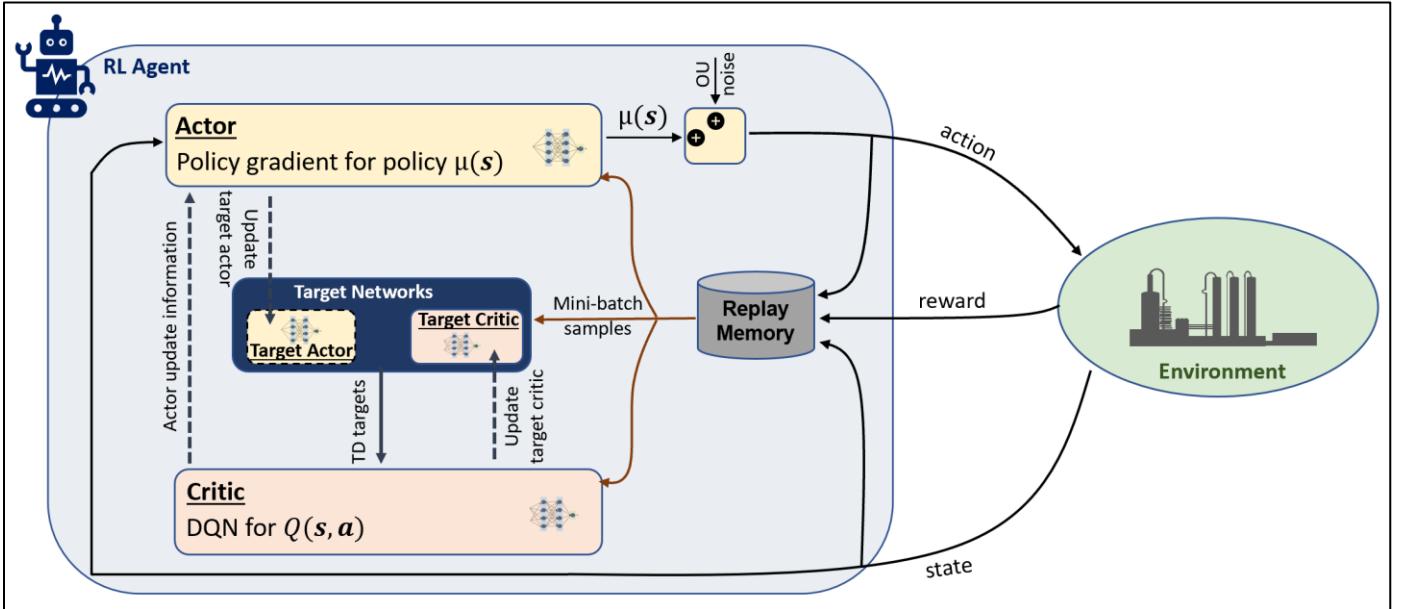


Figure 13.6: RL agent training within DDPG framework. Note that 4 neural networks are used.

Replay memory

In Algorithm 3 we saw that the network parameter update is based on a single transition tuple $(\mathbf{s}_t, \mathbf{a}_t, r_{t+1}, \mathbf{s}_{t+1})$. The problem with this (apart from less computational efficiency compared to mini-batch updates) is that the sampled tuples at successive updates are correlated and therefore not completely independent. This has been found to cause convergence issues. Replay buffer or replay memory (RM) helps to overcome this issue.

A replay memory is simply a large cache of transition tuples. At any training step, a transition tuple $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}')$ is generated according to the agent's behavior policy and stored in RM. If the RM is already full (number of tuples become equal to some prespecified size K), the oldest sampled tuple is removed. The actor and critic networks are updated using a random mini-batch of M ($M < K$) tuples from the RM. Expressions below show the mini-batch update mechanism. Let $B = \{(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}'_i)\}_{i=1}^K$ be the mini-batch of transition tuples from RM. Then,

$$\mathbf{w} \leftarrow \mathbf{w} + \frac{\alpha_c}{M} \sum_{i=1}^M (\tilde{y}_i - Q_{\mathbf{w}}(\mathbf{s}_i, \mathbf{a}_i)) \frac{\partial Q_{\mathbf{w}}(\mathbf{s}_i, \mathbf{a}_i)}{\partial \mathbf{w}}$$

$$\text{where } \tilde{y}_i = r_i + \gamma Q_{\mathbf{w}}(\mathbf{s}'_i, \mu_{\theta}(\mathbf{s}'_i))$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \frac{\alpha_{\boldsymbol{\theta}}}{M} \sum_{i=1}^M \frac{\partial Q_w(s_i, a)}{\partial a} \Big|_{a=\mu_{\boldsymbol{\theta}}(s_i)} \frac{\partial \mu_{\boldsymbol{\theta}}(s_i)}{\partial \boldsymbol{\theta}}$$
eq. 3

We had previously seen how mini-batch update is superior to stochastic update for ANN training in Chapter 11. Similar observations have been made for RL as well.

Target networks

Critic networks are trained by minimizing the error between the predicted Q value and TD target. However, as we noted previously, the TD target itself depends on the critic network that we are trying to fit. This makes the AC algorithm prone to divergence. This is resolved by creating replicas of the actor and critic networks, called target networks, whose parameters evolve slowly but track the main networks being learnt. Let $Q_{w,target}$ and $\mu_{\boldsymbol{\theta},target}$ denote the target networks. The TD target for the i^{th} transition tuple from RM at any training step is given by

$$\tilde{y}_i = r_i + \gamma Q_{w,target} \left(s'_i, \mu_{\boldsymbol{\theta},target}(s'_i) \right)$$

After the actor and critic networks have been updated (using Eq. 3), target networks are updated as follows

$$\begin{aligned} w_{target} &\leftarrow \tau w + (1 - \tau) w_{target} \\ \boldsymbol{\theta}_{target} &\leftarrow \tau \boldsymbol{\theta} + (1 - \tau) \boldsymbol{\theta}_{target} \end{aligned}$$

Where τ is a hyperparameter between 0 and 1 (usually kept close to 0). DDPG algorithm, therefore, has 4 neural networks that are fitted simultaneously.

OU process as exploration noise

In tabular Q-learning we had used ε -greedy behavior policy. Unfortunately, ε -greedy assumes discrete action space. To enable sufficient exploration in DDPG during RL agent training, Lillicrap et al. added noise to the actor's actions

$$\beta(s_t) = \mu_{\boldsymbol{\theta}}(s_t) + N$$

The noise N was chosen to be given by an Ornstein-Uhlenbeck (OU) process. An OU process is a stochastic process (details provided later in the code) which provides temporally correlated signals favoring deep exploration.

The overall DDPG algorithm is summarized in the pseudo-code below.

Algorithm 4: DDPG algorithm

1. Initialize network parameters, \mathbf{w} and $\boldsymbol{\theta}$, with random values
2. Set $\mathbf{w}_{target} \leftarrow \mathbf{w}$, $\boldsymbol{\theta}_{target} \leftarrow \boldsymbol{\theta}$
3. for episode = 1 to $N_{episode}$ do:
4. initialize state \mathbf{s}
5. for each step (t) of episode until \mathbf{s} is terminal do:
6. get action (\mathbf{a}_t) for \mathbf{s}_t as per current policy μ_θ and OU noise
7. take action \mathbf{a}_t and observe r_{t+1} and next state \mathbf{s}_{t+1}
8. Store tuple $(\mathbf{s}_t, \mathbf{a}_t, r_{t+1}, \mathbf{s}_{t+1})$ in RM
9. Sample a minibatch of M tuples from RM
10. Compute TD targets using target networks :
$$\tilde{y}_i = r_i + \gamma Q_{\mathbf{w},target}(\mathbf{s}'_i, \mu_{\theta,target}(\mathbf{s}'_i)) \text{ for each } i^{\text{th}} \text{ tuple in minibatch}$$
11. Update critic:
$$\mathbf{w} \leftarrow \mathbf{w} + \frac{\alpha_c}{M} \sum_{i=1}^M (\tilde{y}_i - Q_{\mathbf{w}}(\mathbf{s}_i, \mathbf{a}_i)) \frac{\partial Q_{\mathbf{w}}(\mathbf{s}_i, \mathbf{a}_i)}{\partial \mathbf{w}}$$
12. Update actor:
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \frac{\alpha_\theta}{M} \sum_{i=1}^M \frac{\partial Q_{\mathbf{w}}(\mathbf{s}_i, \mathbf{a})}{\partial \mathbf{a}} |_{\mathbf{a}=\mu_\theta(\mathbf{s}_i)} \frac{\partial \mu_\theta(\mathbf{s}_i)}{\partial \boldsymbol{\theta}}$$
13. Update target networks:
$$\mathbf{w}_{target} \leftarrow \tau \mathbf{w} + (1 - \tau) \mathbf{w}_{target}$$

$$\boldsymbol{\theta}_{target} \leftarrow \tau \boldsymbol{\theta} + (1 - \tau) \boldsymbol{\theta}_{target}$$
14. end for
15. end for

You are now ready to build your RL-based process controllers using DDPG algorithm. Let's revisit the fluid level control problem and look at it in more detail.

13.6 DDPG RL Agent as Level Controller

Figure 13.7 below shows several details of our level control problem⁷⁷. As alluded to earlier, the RL agent controller needs to learn how to control the tank liquid level (specifically, keep it between 47.5% to 52.5%) under the influence of inflow disturbances. This simple example will hopefully help you understand how to setup RL agent training and its subsequent deployment.

⁷⁷ The system, problem, and solution mechanism are adopted from the thesis work: 'E. R. Mageli, Reinforcement learning in process control, Norwegian University of Science and Technology, 2019'.

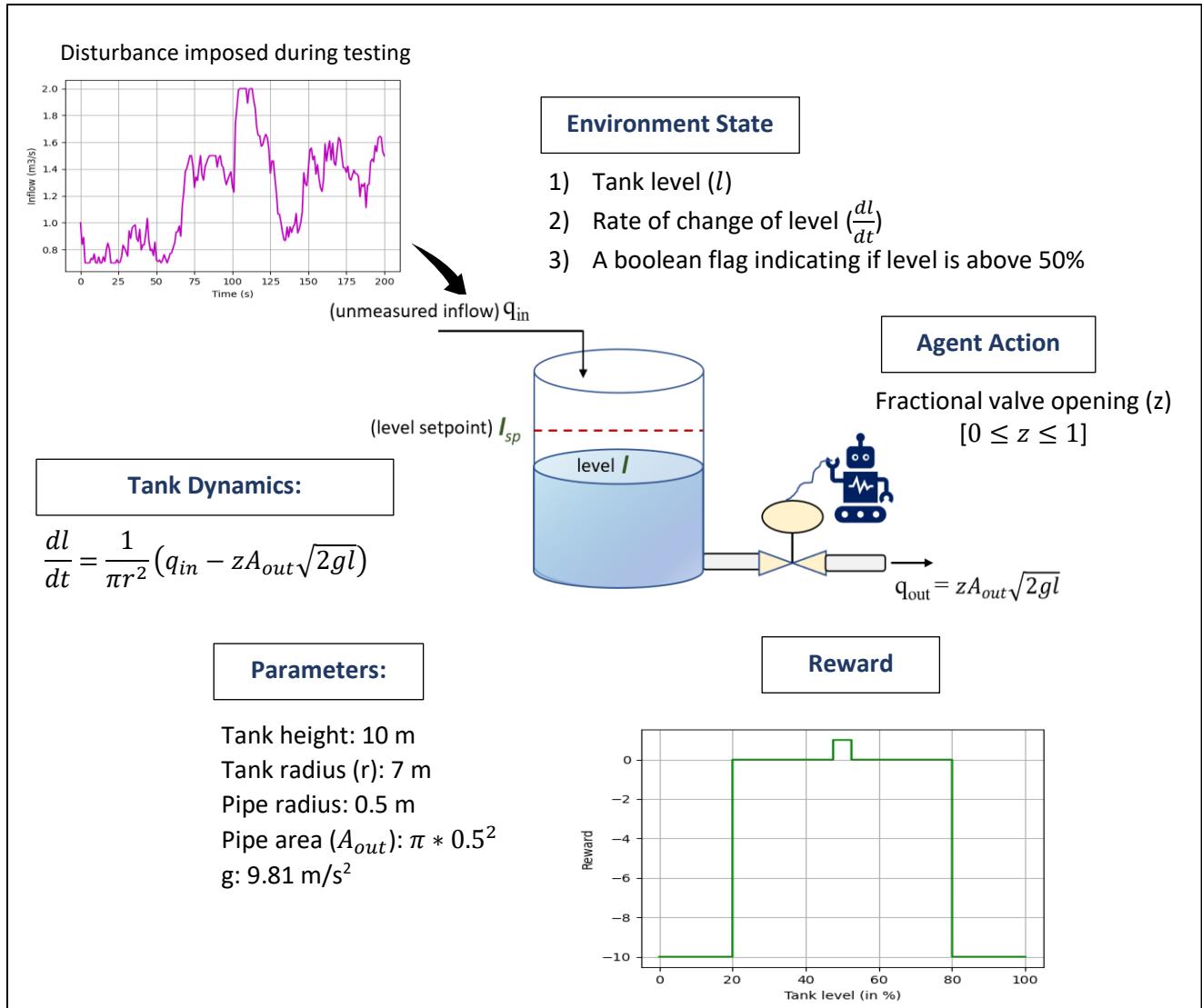


Figure 13.7: RL-based control of tank level: System settings

Let us first define the tank environment⁷⁸ that the RL agent will interact with. If you are not familiar with using Python Classes, don't worry. Most of the code is self-explanatory and appropriate annotations have been provided where necessary.

```
# Tank class defined here will be used later in our RL agent training script
import numpy as np
import random

class tank_environment:
    """ Create an OpenAI style environment
    """

```

⁷⁸ A popular Python library, Gym, contains several ready-built environments such as cartpole and pendulum.

```

def __init__(self, pre_def_dist=False, pre_distFlow=[1]):
    # tank related parameters
    self.height = 10
    self.radius = 7
    self.level = 0.5 * self.height # initial level is at 50%
    self.hard_max_level = 0.8 * self.height
    self.hard_min_level = 0.2 * self.height
    self.soft_max_level = 0.525 * self.height
    self.soft_min_level = 0.475 * self.height
    self.pipe_radius = 0.5
    self.pipe_Aout = np.pi*self.pipe_radius*self.pipe_radius

    # disturbance related
    self.pre_def_dist = pre_def_dist
    self.pre_distFlow = pre_distFlow # disturbance flows used during testing
    self.distFlow = [1] # stores disturbance flows during a training episode

```

def get_disturbanceFlow(self, t=0):

....

Arguments:

t: corresponds to the step number of an episode [used during testing]

Returns:

a (scalar) disturbance flow

....

if self.pre_def_dist:

 return self.pre_distFlow[t]

else:

 new_flow = random.normalvariate(self.distFlow[-1], 0.1)

 # impose bounds on disturbance flow

 if new_flow > 2:

 new_flow = 2

 elif new_flow < 0.7:

 new_flow = 0.7

 self.distFlow.append(new_flow)

 return new_flow

__init__ function is used to initialize the tank_environment

Breach of these levels imply task termination

Desired range for the liquid level



Disturbance flow modeled as a Gaussian random-walk time series where current flow is normally distributed around the flow at the previous step

```
def step(self, action):
```

```
    """
```

Description:

accepts an action and returns a tuple (state, reward, done).

Args:

action (valve opening): an action provided by the agent

Returns:

state (numpy array of size (3,)): agent's observation of the current environment

reward (float) : amount of reward returned after taking the action

done (bool): indicates whether an episode has been terminated

```
"""
```

```
# parameters
```

```
g = 9.81
```

```
for i in range(5):
```



The system is simulated for 5 timesteps for an action during training. This was found to help during training.

```
# compute rate of change of tank level
```

```
q_dist = self.get_disturbanceFlow()
```

```
q_out = action*self.pipe_Aout*np.sqrt(2*g*self.level)
```

```
dhdt = (q_dist - q_out)/(np.pi*self.radius*self.radius)
```

```
# compute new tank level
```

```
self.level = self.level + dhdt
```



Simulating tank dynamics

```
# check termination status
```

```
done = False
```

```
if self.level < self.hard_min_level:
```

```
    done = True
```

```
    break
```

```
elif self.level > self.hard_max_level:
```

```
    done = True
```

```
    break
```

```
# check level above 50%
```

```
if self.level >= 0.5*self.height:
```

```
    above = 1
```

```
else:
```

```
    above = 0
```

```

# compute reward
if done:
    reward = -10
elif self.level > self.soft_min_level and self.level < self.soft_max_level:
    reward = 1
else:
    reward = 0

# generate observation/state vector
next_state = np.array([self.level/self.height, dhdt, above]) # using normalized level

return next_state, reward, done

def reset(self):
    """
    Description: Reset tank environment to initial conditions
    Returns: Initial state of the environment
    """

    self.level = 0.5 * self.height
    self.distFlow = [1]

    return np.array([self.level/self.height, 0, 1])

```

With the behavior of the tank environment defined, we can now start writing the code for agent training. We will begin by importing some packages and defining some utility functions. If you have defined the `tank_environment` class in a separate script file (say, `Tank_Environment.py`), you will have to import the class in your current script (as done here).

```

# import packages
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from Tank_Environment import tank_environment

# define utility functions
def ActorNetwork(n_states=3, n_action=1):
    """
    Description: Get actor network with the given parameters
    Returns: the keras model
    """


```

```

model = Sequential()
model.add(Dense(8, activation='relu', input_shape=(n_states,)))
model.add(Dense(8, activation='relu'))
model.add(Dense(n_action, activation='sigmoid'))

return model

def CriticNetwork(n_states=3, n_action=1):
    """
    Description: Get critic network with the given parameters
    Returns: the keras model
    """

    model = Sequential()
    model.add(Dense(8, activation='relu', input_shape=(n_states+n_action,)))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1))

    return model

```

The above two functions will be used later to create our actor, critic, and the targets networks. The next function performs a soft update of the target networks.

```

def update_target(target_network, ref_network, tau=0.01):
    """
    update target network weights using the weights of the reference/main networks
    """

    target_network_weights = target_network.get_weights()
    ref_network_weights = ref_network.get_weights()

    for index in range(len(target_network_weights)):
        target_network_weights[index] = tau*ref_network_weights[index] +
                                       (1-tau)*target_network_weights[index]
    target_network.set_weights(target_network_weights)

    return target_network

```

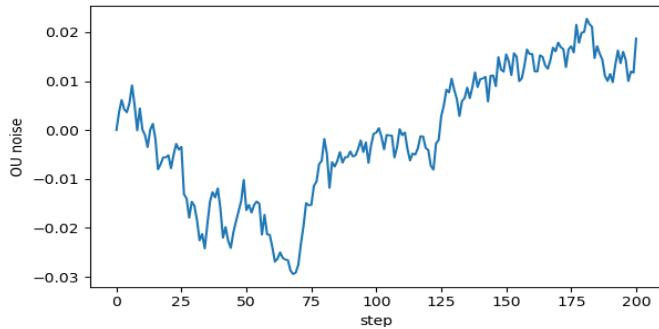
The next function generates OU noise that is added as a disturbance to the actor's action. The subsequent figure shows an example of the kind of noise pattern generated by this mechanism. As is apparent, an OU process shows greater inertia (compared to a wildly fluctuating gaussian process) as the OU noise signals tend to evolve in the same direction for long durations; this promotes deep exploration during RL agent's training.

```

def OU_noise(noise_prev, theta=0.15, mu=0, sigma=0.025, n_action=1):
    """
    Description: Simulates OU process to generate time-correlated noise
    Adapted from github.com/openai/baselines/blob/master/baselines/ddpg/noise.py
    """

    noise = noise_prev + theta*(mu-noise_prev) + sigma*np.random.randn(n_action)
    return noises

```



Noise profile over an episode generated using the OU_noise function

Next, we create a function to sample a batch of stored transition tuples from the RM. You will notice that an additional variable ‘done’ is part of a transition tuple here. The significance of this variable will become clear in the next utility function.

```

def sample_ReplayMemory():
    """
    Returns:
        states: A 2D numpy array with 3 columns
        actions: A 2D numpy array with 1 column
        rewards: A 1D numpy array
        next_states: A 2D numpy array with 3 columns
        dones: A 1D numpy array
    """

```

Adapted from github.com/ageron/handson-ml2/blob/master/18_reinforcement_learning.ipynb

```

# select a batch of transitions from the replay memory
indices = np.random.randint(len(replay_memory), size=batchSize)
batch = [replay_memory[index] for index in indices]

# separate the states, actions, rewards, next_states, dones from the selected transitions
states, actions, rewards, next_states, dones = [np.array([transition[field_index] for transition in
    batch]) for field_index in range(5)]

return states, actions, rewards, next_states, dones

```

Previously, we computed the TD targets as $\tilde{y}_i = r_i + \gamma Q_{w,target}(s'_i, \mu_{\theta,target}(s'_i))$ for each i^{th} transition tuple in a minibatch from RM (see Algorithm 4). We will now make a small modification in this formula as follows

$$\tilde{y}_i = r_i + (1 - done_i) * \gamma Q_{w,target}(s'_i, \mu_{\theta,target}(s'_i)) \text{ for each } i^{th} \text{ tuple in minibatch}$$

The revised formula states the obvious fact that the TD target is simply the reward (r_i) if the episode or task terminates ($done_i$ equals 1 or True) from taking action a_i at state s_i . The function below implements this formula.

```
def get_TD_targets(rewards, target_Q_values, dones, gamma=0.95):
```

```
    """
```

Args:

rewards: A 1D numpy array

target_Q_values: A 2D numpy array with 1 column

dones: A 1D numpy array

Returns:

td_targets: A 2D numpy array with 1 column

```
    """
```

```
    td_targets = np.zeros_like(target_Q_values)
```

```
    for i in range(target_Q_values.shape[0]):
```

```
        if dones[i]:
```

```
            td_targets[i] = rewards[i]
```

```
        else:
```

```
            td_targets[i] = rewards[i] + gamma*target_Q_values[i]
```

```
    return td_targets
```

Alright, we are now at the last utility function that we will define. This function updates the main actor and critic networks using data from the sampled minibatch. There are several new concepts involved here that warrant some explanations. You will notice the use of `tf.function` decorator⁷⁹ which converts a regular Python function to a Tensorflow function resulting in faster computations. The input signature specifies the shape and type of each Tensor argument to the function. Within the function, `GradientTape API`⁸⁰ is utilized which implements automatic differentiation for gradient computations. This API allows Tensorflow to keep a ‘record’ of what operations are executed within the context of the `tf.GradientTape`. When

⁷⁹ <https://www.tensorflow.org/guide/function>

⁸⁰ <https://www.tensorflow.org/guide/autodiff>

tape.gradient function is called, Tensorflow computes the gradients of the recorded operations w.r.t. the specified variables.

```
@tf.function(input_signature=[tf.TensorSpec(shape=(None, 3), dtype=tf.float32),
                             tf.TensorSpec(shape=(None, 1), dtype=tf.float32),
                             tf.TensorSpec(shape=(None, 1), dtype=tf.float32)])
```

```
def ddpg_updateStep(s, a, td_targets):
    ....
```

Description: Update the main networks using gradient descent

Args:

- s: A 2D numpy array with 3 columns
- a: A 2D numpy array with 1 column
- td_targets: A 2D numpy array with 1 column

....

update critic via step 11 in Algorithm 4

networks, loss_fn (mean squared error) and optimizers are defined later in the script
with tf.GradientTape() as tape:

```
Q_predicted = critic(tf.concat([s, a], 1))
```

critic_loss = tf.reduce_mean(loss_fn(td_targets, Q_predicted)) } Critic loss: $L(\mathbf{w})$

```
critic_gradients = tape.gradient(critic_loss, critic.trainable_variables)
```

$$\frac{\partial L}{\partial \mathbf{w}}$$

```
critic_optimizer.apply_gradients(zip(critic_gradients, critic.trainable_variables))
```

$$\mathbf{w}_t \leftarrow \mathbf{w}_t - \alpha \frac{\partial L}{\partial \mathbf{w}}$$

update actor

with tf.GradientTape() as tape:

actor_loss = -tf.reduce_mean(critic(tf.concat([s, actor(s)], 1)))

actor_gradients = tape.gradient(actor_loss, actor.trainable_variables)

```
actor_optimizer.apply_gradients(zip(actor_gradients, actor.trainable_variables))
```

$$\text{Actor loss} = -J(\theta) \approx Q_{\mathbf{w}}(s_t, a)|_{a=\mu_{\theta}(s_t)}$$

$$\frac{\partial \text{loss}}{\partial \theta} = \frac{-\partial Q_{\mathbf{w}}(s_t, a)}{\partial \theta}$$

Note that chain rule is not used here

Okay, let's train the agent now.

```
# define replay memory
from collections import deque
replay_memory = deque(maxlen=100000)
```

```

# define networks and set their optimizers
critic = CriticNetwork()
target_critic = CriticNetwork()
target_critic.set_weights(critic.get_weights())

actor = ActorNetwork()
target_actor = ActorNetwork()
target_actor.set_weights(actor.get_weights())

critic_optimizer = keras.optimizers.Adam(lr=0.01)
actor_optimizer = keras.optimizers.Adam(lr=0.01)

loss_fn = keras.losses.MeanSquaredError()

# define environment
from Tank_Environment import tank_environment
env = tank_environment() # __init__ function of the class gets called implicitly here

# some training settings
rewards = [] # stores the total reward collected over each episode
n_episodes = 300
n_steps = 200
batchSize = 64

# the training loop
for episode in range(n_episodes):
    state = env.reset()
    noise = 0
    episode_reward = 0

    for step in range(n_steps):
        # determine action
        action = actor.predict(state.reshape(1,-1)).flatten()
        noise = OU_noise(noise)
        action = np.clip(action + noise, 0, 1) # ensuring action remains within 0 and 1

        # implement action, get back new state and reward, and save in RM
        next_state, reward, done = env.step(action[0])
        episode_reward = episode_reward + reward
        replay_memory.append((state, action, reward, next_state, done))

    # update the networks

```

```

if len(replay_memory) >= batchSize:
    # sample transitions
    RM_states, RM_actions, RM_rewards, RM_next_states, RM_dones = sample_ReplayMemory()

    # compute TD targets (step 10 in Algorithm 4)
    RM_target_next_actions = target_actor.predict(RM_next_states)
    RM_target_Q_values = target_critic.predict(np.concatenate((RM_next_states,
                                                               RM_target_next_actions), axis=1))
    RM_td_targets = get_TD_targets(RM_rewards, RM_target_Q_values, RM_dones)

    # update main networks
    ddpg_updateStep(tf.convert_to_tensor(RM_states, dtype=tf.float32),
                    tf.convert_to_tensor(RM_actions, dtype=tf.float32),
                    tf.convert_to_tensor(RM_td_targets, dtype=tf.float32))

    # update target networks
    target_critic = update_target(target_critic, critic)
    target_actor = update_target(target_actor, actor)

    # preparation for next step
    state = next_state
    if done:
        break

    rewards.append(episode_reward)

# save trained models
actor.save('actor_saved')

```

The above script completes the implementation of RL agent's training via DDPG algorithm. As commonly encountered with RL trainings, hyperparameter tuning proved to be difficult even for our simple tank system. Often, training would converge to a trivial RL controller that would completely open and close the valves to control the liquid level. Figure 13.8 shows the performance of RL agent during testing with a converged reasonable solution. The code below shows how to setup RL testing/deployment.

```

# Test RL agent with pre-specified disturbance profile
distFlow_fileName = 'disturbance_200.csv'
distFlow = np.loadtxt(distFlow_fileName, delimiter=',')
env = tank_environment(pre_def_dist=True, pre_distFlow=distFlow)
state = env.reset()

```

```

level_hist = []
valve_opening_hist = []

actor = keras.models.load_model('actor_saved')
for step in range(n_steps):
    # take action
    action = actor.predict(state.reshape(1,-1)).flatten()
    next_state, reward, done = env.step_test(action[0], step) # step_test function is same as step
    # function except that the system is simulated for only 1 timestep for each action

    # store
    valve_opening_hist.append(action[0])
    level_hist.append(next_state[0]*env.height)

    # check termination
    if done:
        break
    else:
        state = next_state

```

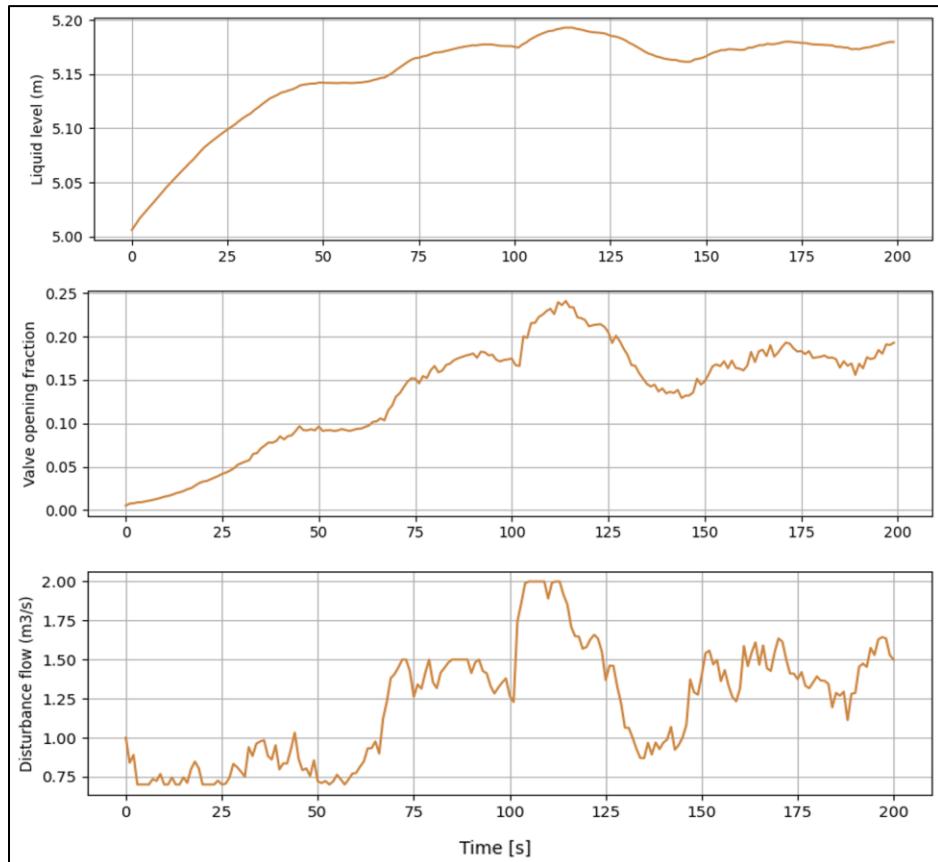


Figure 13.8: Evaluation of RL controller for the tank liquid level control

As the plots show, RL agent is able to keep the liquid level within the most profitable region by nicely modulating the valve opening. For example, around 110 seconds, when the disturbance flow increases sharply, RL controller opens the valve further to keep the level stable. Remember that RL agent does not have knowledge of the disturbance's magnitude. It has been trained to take an optimal decision only using information about the current liquid level and the rate of change of level.

As previously mentioned, RL training may throw up several challenges at you. However, you do have several recourses. Reformulating your problem can be a worthwhile thing to do. For example, in the tank level control problem, you may use Δf (change in valve opening) as the action and add the current f as another state variable⁸¹. You may also engineer the reward function differently and penalize the controller for making large action moves. More recent algorithms such as TD3, PPO may be tried that try to tackle the training stability issues. Hopefully, you have now gained adequate understanding of RL fundamentals to help you judge your situation appropriately and make wise decisions.

This concludes our quick tour of the world of reinforcement learning. As remarked before, RL is a vast field and many more algorithms have been proposed since DDPG which are gaining popularity. The exposition in this chapter will help you get started with RL and think of all the different ways you can utilize this powerful tool. Unlike other ML technologies, RL has not yet seen widespread adoption in the process industry (due to long training times, difficulty in imposing process constraints, requirement of adequate simulator for initial offline training, etc.). However, considering the amount of research ongoing in RL and increasing ease of availability of cheap computational resources, it is not far-fetched to state that RL will play an important role in industrial process control in the near future.

Summary

In this chapter, we looked at theoretical and practical aspects of RL. We studied two popular algorithms namely, Q-learning and DDPG, in detail. We developed a RL-based controller for controlling the fluid level in a tank. We hope that RL is no longer a mystery for you. With this chapter, we have also reached the end of our ML journey. During this journey, you picked up several powerful tools. As our parting message, we would just advise you the ‘No Free Lunch’ theorem that states that no single method exists that can obtain best results all the time. Hopefully, the knowledge you gained from the book will help you make the right (and educated) modeling choices for your problems. All the best!

⁸¹ Sometimes, to ensure that the state vector has all the needed information, you may need to add some past observations and actions in your state vector. This is analogous to data augmentation we have seen previously.

Part 4

Deploying ML Solutions Over Web

Chapter 14

Process Monitoring Web Application

You have done all the hard work to obtain a ML model that meets performance criteria and now it's time to deliver the solution / model to its end-users who will be using the model's results on a regular basis. But how do you do it? If the end-users are non-technical (from data-science perspective) like plant operators, you cannot ask them to have their own Python installation to run the ML model. Under such circumstances, deployment over web is a good and frequently employed solution for delivering ML results.

For web deployment, Python provides several frameworks (like Django, Flask, CherryPy) for developing simple to complex web application fast. Very often, all you may want is a quick prototype that allows you to collect user feedback. Keeping this in mind, we will show you how to build a light-weight web application from scratch and demonstrate how easy it is to do so. Specifically, we will build a process monitoring tool that provides 24 X 7 fault detection & diagnosis (FDD) results to plant operators over web. During the course of building this solution, we will learn the following topics

- Deploying Python applications using CherryPy's web server
 - Embedding ML models into a web application
 - Saving ML models for later reuse
 - Building front-end user interfaces using HTML, CSS, Javascript
-

14.1 Process Monitoring Web App: Introduction

Figure 14.1 shows the user interface that we will build in this chapter to display the results in real-time from a process monitoring ML model. The user interface is to be accessed via a web browser and provides two crucial information to the plant operators. First, the fault detection component communicates to the operators the state of the process and alerts them if any process abnormality is detected. Second, the fault diagnosis component identifies the faulty variables responsible for process abnormality. The monitoring model will be built using PCA and the methodology introduced in Chapter 5 will be employed for FDD. The process system is the same as that in Chapter 5, i.e., the polymer processing plant with 33 variables.

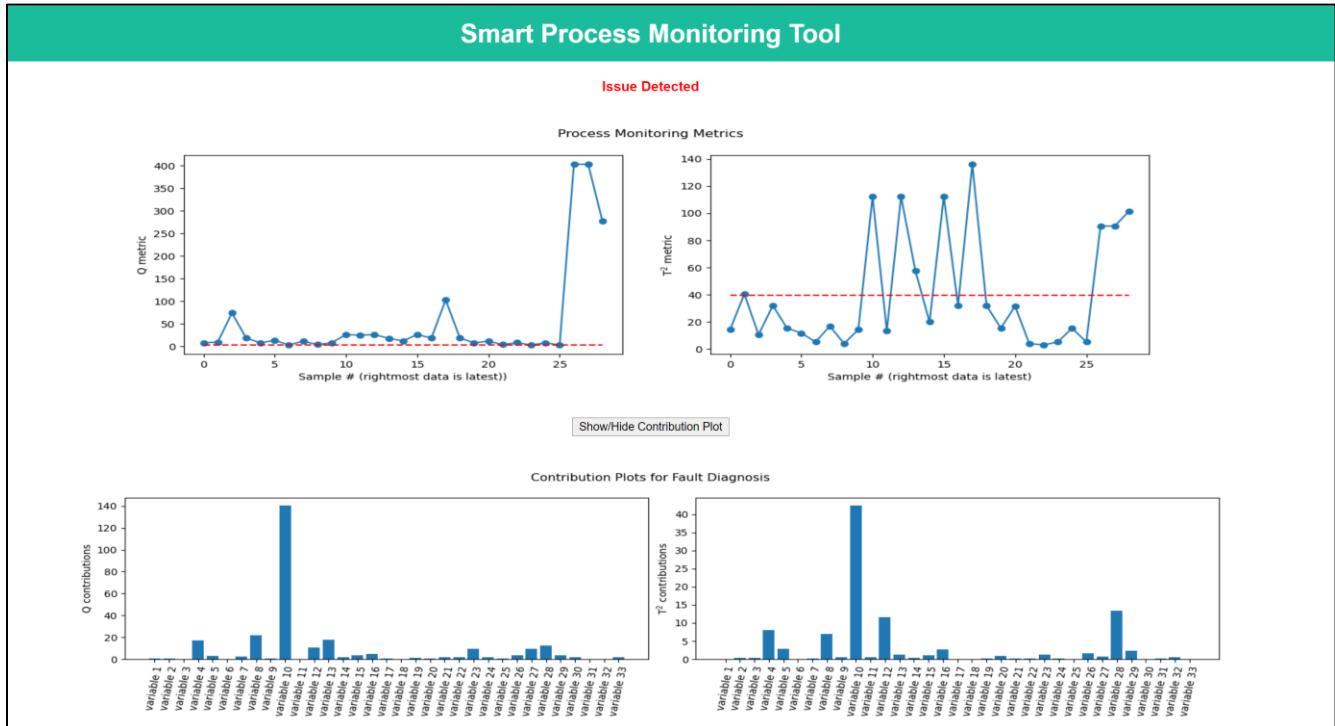
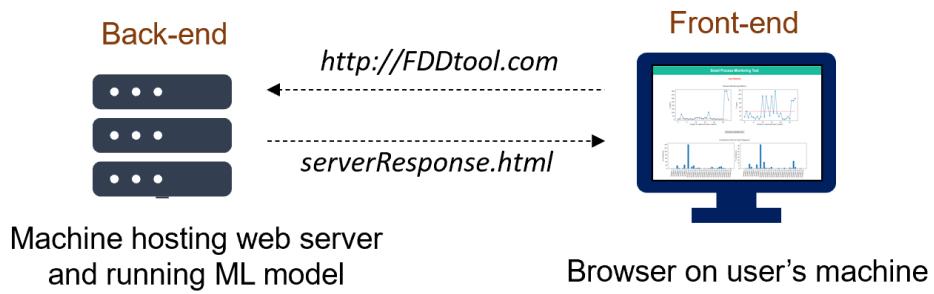


Figure 14.1: Process monitoring web app user interface

A web application primarily has 2 main parts: front-end and back-end. Front-end (or client-side) is the part that end-users see and interact with directly through web browsers. The back-end works behind the scenes to deliver information to the browsers. When a user enters your website's URL in browser, a request is sent to the back-end which parses this request, processes it, and sends back a response which is displayed on the front-end. Illustration below shows the data transmission scheme that we will employ



As you can see, the data analytics occurs on the back-end residing on a dedicated computer and the results are seen on any machine via browsers. While the front-end in our application will do only the basic job of displaying FDD results, front-ends can do much more such as user-data entry validation, animations, etc., and these complex functionalities are handled by any modern internet browser.

14.2 A Simple ‘Hello World’ Web App

Before we build the full-fledged monitoring tool, let’s build a simple ‘Hello World’ application to gain some familiarity with the process of developing a web application using Python. We will use `CherryPy`⁸² package, a popular and simple web application framework, which is used for rapid deployment of web apps and is very easy to learn. CherryPy provides a production-ready web server which can easily handle medium-scale web applications and therefore, is suitable for our purpose.

For the simple application, let’s type the following code in a Python script⁸³ `helloWorld.py` and execute it in the Anaconda command terminal (to execute from windows command terminal, you will need to edit Windows Environment variables to add Anaconda to your System Path)

```
# import package
import cherrypy

# Web application will be written as a Python class.
# Methods of the class will be used to respond to client requests
class HelloWorld(object):
```

⁸² You can install CherryPy via `pip install cherrypy` command. The official CherryPy doc has some excellent instructive tutorials (<https://docs.cherrypy.org/en/latest/tutorials.html>). Do check them out if you want to learn more.

⁸³ We will not cover the concepts behind Python classes or CherryPy configuration in detail. While understanding them is useful, it’s not a necessity to be able to understand the script. You can just follow the structure shown here and tweak them if needed as you become more familiar with developing web applications.

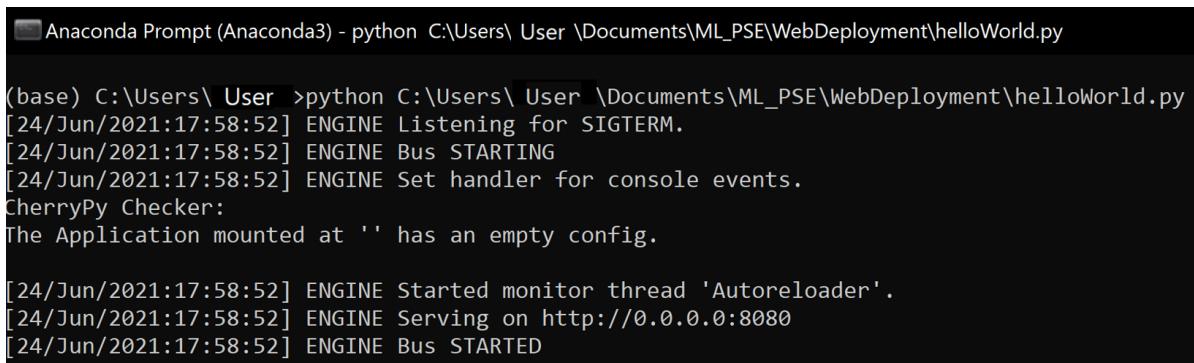
```

@cherrypy.expose
def index(self):
    return "Hello world!"

# execution settings
cherrypy.config.update({'server.socket_host': '0.0.0.0'})
if __name__ == '__main__':
    cherrypy.quickstart(HelloWorld()) # when this script is executed, host HelloWorld app

```

You will see the following ‘strange looking’ output on the terminal



```

Anaconda Prompt (Anaconda3) - python C:\Users\ User \Documents\ML_PSE\WebDeployment\helloWorld.py

(base) C:\Users\ User >python C:\Users\ User \Documents\ML_PSE\WebDeployment\helloWorld.py
[24/Jun/2021:17:58:52] ENGINE Listening for SIGTERM.
[24/Jun/2021:17:58:52] ENGINE Bus STARTING
[24/Jun/2021:17:58:52] ENGINE Set handler for console events.
CherryPy Checker:
The Application mounted at '' has an empty config.

[24/Jun/2021:17:58:52] ENGINE Started monitor thread 'Autoreloader'.
[24/Jun/2021:17:58:52] ENGINE Serving on http://0.0.0.0:8080
[24/Jun/2021:17:58:52] ENGINE Bus STARTED

```

This output simply indicates that a web server has started and is ready to accept incoming requests on the address `http://localhost:8080` (or `http://127.0.0.1:8080`). Go ahead and type this address on your browser’s URL address bar. You should see the following



What’s happening here is that the web server directs the incoming request to the `helloWorld` app which triggers the execution of the default `index`⁸⁴ function which returns the text ‘Hello World’ to the browser which is then displayed to you. In this illustration, the back-end and the front-end both reside on your PC; however, for tool deployment, you would setup the back-end on some other dedicated machine. Our web server can also listen to requests coming from some other PC that is in the same workplace network – the URL will need to be modified to make it explicitly specify that the web server is hosted on your PC. Assuming your PC name is `machineName`, you can ask any colleague of yours to use the address `http://machineName:8080` on their browser – they should see the same result as above.

⁸⁴ ‘index’ function is executed by default if no explicit function (or path segment) is provided in the URL

This is it! This is all that is needed to create and deploy a web application. You would not have expected it to be so easy, would you? By now, you might be beginning to get some ideas on how we can extend this simple web app for our monitoring tool. All we need is that our web server serves FDD results upon receiving browser requests. We will now add the required components in our simple web app to enable this functionality.

14.3 Embedding ML Models into Web Apps

To provide monitoring results to plant operators, we will replace the `HelloWorld` class in our `HelloWorld.py` script with a `FDDapp` class as shown below and save the file as `FDD.py`.

```
# FDD tool web application
class FDDapp(object):
    @cherrypy.expose
    def getResults(self):
        processState = runPCAmodel() # returns 'All Good' or 'Issue Detected'
        return processState
```

If you use the web address `http://machineName:8080/getResults` (after we have defined the logic for `runPCAmodel` function), the `getResults` function will get executed which will execute `runPCAmodel` function to generate results and return the current process state information. For now, our returned output is very simple, but we will modify it later to return more instructive results. Before we look into the code for `runPCAmodel` function, an important note that we would not want to (re) train and generate a PCA model every time our web app is accessed. What we can do instead is to save the generated PCA model (and other model variables) after model training once and then just re-use the saved model in the `runPCAmodel` function.

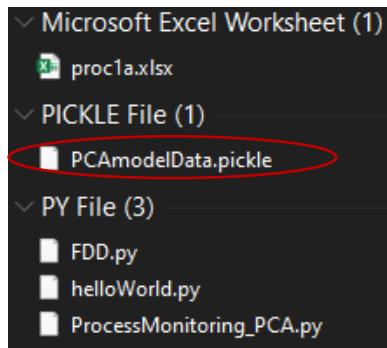
To save and re-use ML models and model variables, we can use the `pickle` package. Let's add the following code to our `ProcessMonitoring_PCA.py` script from Chapter 5.

```
# saving PCA data for later re-use
import pickle
PCAmodelData = {"PCAmodel": pca,
                 "scaler": scaler,
                 "n_comp": n_comp,
                 "P_matrix": P_matrix,
                 "lambda_k_inv": lambda_k_inv,
```

```
"Q_CL": Q_CL,  
"T2_CL": T2_CL} # dictionary data structure uses key-value pairs
```

```
with open('PCAmodelData.pickle', 'wb') as f:  
    pickle.dump(PCAmo
```

If you re-run the ProcessMonitoring_PCA.py script, you will now see a PCAmodelData.pkl file in your project folder.



Let's now look at the runPCAmodel function below. You will notice that for ease of code maintenance and readability, we have separated out different tasks into separate helper functions. Note that if you prefer you can put the runPCAmodel function in a separate .py file and the import it into your FDD.py script. For simplicity, we will keep all the code in FDD.py file. As the explanatory comments indicate, we load the data in the saved pickle files into the workspace, fetch latest process data, transform latest data via PCA, compute monitoring indexes, check for the presence of process fault, and create some plots that we will eventually show on the user interface. The resulting FD result is returned. The monitoring index computation code is similar to that in Chapter 5 and therefore the helper functions' details are not shown here, but you are encouraged to check them out in the online script file.

```
# runPCAmodel function  
def runPCAmodel():  
    # read saved PCA model data  
    with open('PCAmo
```

```

[currentQ, currentT2, Q_contri, T2_contri] = computeMetricsContributions(PCAmoelData,
processData)

# detect fault
if (currentQ > PCAmoelData["Q_CL"]) or (currentT2 > PCAmoelData["T2_CL"]):
    processSate = 'Issue Detected'
else:
    processSate = 'All Good'

# generate and save metric plot containing historical metrics and current metrics
generateMetricPlots(currentQ, currentT2, PCAmoelData["Q_CL"], PCAmoelData["T2_CL"])
# saves 'metricPlot.png' in the working folder

# generate and save contribution plots for latest data
generateContributionPlots(Q_contri, T2_contri) # saves 'contributionPlot.png'

return processSate

```



To keep the illustration simple, the `getLatestProcessData` function fetches current process data from a local CSV file. In real environment, process data may be fetched from a database or from files residing on another machines. All these varied scenarios can be easily handled in Python through appropriate libraries. For example, for database connectivity, you can use `pypyodbc`.

Now you know how you can embed your ML code and model into a web application. But our web app is not complete yet. We still need to instruct our web app to send those generated metrics and contribution plots to the end users.

14.4 Building Front-end User Interface

Interactive and nice-looking user interfaces cannot be rendered with just plain text output returned from web servers. Along with different media objects (like figures), explicit instructions need to be provided to the browsers on how to structure the webpage. Those instructions are written using HTML (specifies the basic structure of the page) and CSS (handles the formatting/appearance aspects) languages that browsers understand. Let's begin by modifying our web app so that we obtain the following user interface.

Smart Process Monitoring Tool

Issue detected

The corresponding HTML code would look like this:

```
<html>
<style>
body
{
    font-family: Arial; text-align: center;
}

#topHeader
{
    padding: 15px; background: #1abc9c; color: white; font-size: 30px;
}
</style>

<body>
<h1 id='topHeader'>Smart Process Monitoring Tool</h1>
<p style="color:red;"><b>Issue detected</b></p>
</body>
</html>
```

You can in fact go ahead and type the above code in a notepad, save it as sample.html and open the file in a browser. You will see the above interface. We can embed this code in our getResults function as per the pseudo-code shown below

```
# getResults function returning HTML code
def getResults(self):
    processState = runPCAmodel()

    # generate front-end code
    html_code = """
        < put HTML code here >
        """
    return html_code
```

While the above code structure would technically be correct, it is not a best practice to keep the front-end code and core back-end code in the same script. The separation can be achieved as follows

```
# getResults function returning HTML code using a template
import os
from jinja2 import Environment, FileSystemLoader

path = os.path.abspath(os.path.dirname(__file__))
env = Environment(loader=FileSystemLoader(path))

def getResults(self):
    processState = runPCAmodel() # returns numeric flag: 0 => 'All good' or 1 => 'Issue detected'

    # generate front-end code
    frontEnd_template = env.get_template('frontEndTemplate.html') # loads a template file
    frontEndCode = frontEnd_template.render(state=processState) # generates front-end code
                                                               using the template
    return frontEndCode
```

where the contents of file *frontEndTemplate.html* look like this

```
<html>
<head>
<style>
    body {font-family: Arial; text-align: center;}
    #topHeader {padding: 15px; background: #1abc9c; color: white; font-size: 30px;}
</style>
</head>

<body>
    <h1 id='topHeader'> Smart Process Monitoring Tool </h1>
    {% if state == 0 %}
        <p style="color:green;"><b> All Good </b> </p> <br>
    {% else %}
        <p style="color:red;"><b> Issue Detected </b> </p> <br>
    {% endif %}
</body>
</html>
```

non-html code
parsed by Jinja

You will notice some additional elements (`{% ... %}`, `if` statements) in the `frontEndTemplate.html` file. These additional elements are not meant for browsers. This file is a template that will be used by the Jinja templating engine (imported in our `FDD.py` script) to generate the required HTML code. You can notice the logic that has been put in place to modify the central dashboard message and its text color depending upon the `processState` value. The templating framework is a beautiful solution to keep the front-end and back-end code separate and encourage code-readability.

Alright, now you understand how front-end code can be put in a web app. Let's move ahead and add our figures to the interface. We will add the following code into the body section of the template file for displaying figures.

```
<body>
...
<center>  </center> <br> <br>
<center>  </center>

</body>
```

All we did above was include image elements into the HTML code and provide the correct path. A small code is also inserted into `FDD.py` script to let the web app know where to find these image files.

```
# execution settings
cherrypy.config.update({'server.socket_host': '0.0.0.0'})
conf = {'/': {'tools.staticdir.on': True, 'tools.staticdir.dir': path}}
```

If you execute `FDD.py` and access your website, you would see the figures on the user interface now. Let's now come to the last and interesting part of adding some dynamics or interactivity to our interface. What we desire is that the contribution plots be shown only when the user clicks on 'Show Contribution Plot' button (these plots would not be much useful when monitoring indices are below the thresholds). To enable this interactivity, we will employ another cornerstone language of web development, Javascript. As shown below, a Javascript function is put in the head section of the template and a button is added to the user interface.

```
<head>
...
<script>
function toggleDisp(imgID) {
    var x = document.getElementById(imgID);
```

```

if (x.style.display == "none")
{
    x.style.display = "block";
}
else {
    x.style.display = "none";
}
</script>
..
</head>

<body>
...
<button type="button" onclick="toggleDisp('contri_img')"> Show/Hide Contribution Plot
</button> <br> <br> <br>
...
</body>

```

When the button is clicked, the Javascript function gets executed which toggles the display property of the contribution plots images. There remains one last piece of the puzzle. Currently, the latest monitoring results are displayed only when a user refreshes/reloads the webpage. This clearly is undesirable. What we want is that the interface refreshes itself automatically to display latest results at regular interval. This is accomplished by including the following in the head section which refreshes/re-loads the interface every 30 seconds.

```
<meta http-equiv="refresh" content="30">
```

Our web app is now complete. Go ahead and try it out. You should see the interface shown in Figure 14.1. The top metric plots will first show only a single data-point and new data-points will show up after every automatic refresh. The world of web development is huge, and we covered only a small portion of it in this chapter to develop a very basic user interface with limited functionalities. Technologies like Ajax, jQuery, Angular, React, Bootstrap exist that can be used to make extremely complex and functionality-rich websites.

Summary

In this chapter we studied the very last phase of the end-to-end development of a machine learning project, i.e., taking the results from ML models to the non-technical end-users via web applications. Concepts learnt in this chapter can help you rapidly deploy your tool and collect crucial user feedback. With this chapter, we have come to the end of our process data science journey. We hope that you enjoyed reading this book as much as we enjoyed writing it!

Appendix

Dataset Descriptions

This appendix section provides a quick overview of the process datasets used in the book. The ML techniques implemented on the datasets are also mentioned.

1) Polymer Manufacturing Process Data

The dataset (named Proc1a) is from a polymer manufacturing facility and was obtained from <https://landing.umetrics.com/downloads-other-downloads>. Data is available in this book's GitHub repository. The dataset contains 33 variables and 92 hourly observations. In the available dataset, variables are scaled and coded for confidentiality reasons⁸⁵.

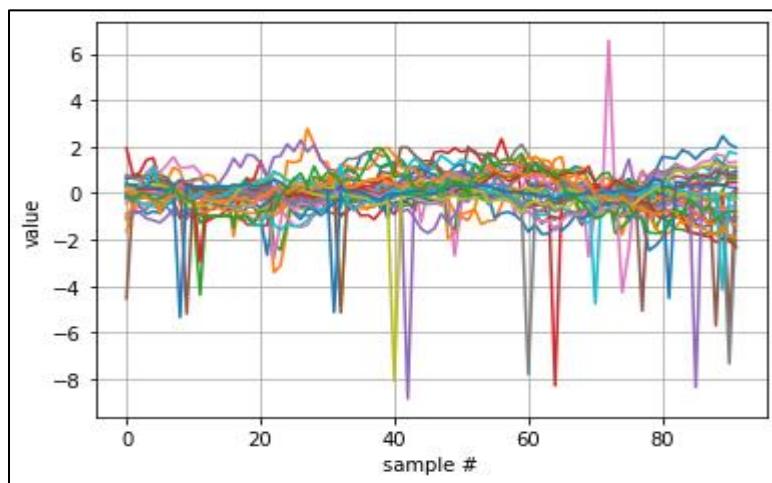


Figure A1: Polymer manufacturing data. Each curve corresponds to a process variable

For this dataset, it was reported that the process started behaving abnormally around sample 70 and eventually had to be shut down. Dimensionality reduction and process monitoring/fault detection & diagnosis using PCA was illustrated using this data in Chapter 5.

2) Pulp & Paper Manufacturing Process Data

The dataset comes from a paper pulp mill and is available at <https://openmv.net>. The dataset contains 301 hourly samples of 21 process variables from a Kamyr digester which, as shown in Figure A2, is a tubular reactor where wood chips react with white liquor (solution of sodium hydroxide and sodium sulfide) to remove lignin from cellulose fibers. Kappa number is the critical quality variable (provided in the dataset) in this process, and it quantifies the lignin content in the pulp. It is desirable to have an accurate prediction model for Kappa number for process monitoring, analyzer calibration monitoring, or in cases where real-time analyzers may not be available. A PLS-based inferential model is built in Chapter 5.

⁸⁵ https://www.academia.edu/38630159/Multivariate_data_analysis_wiki

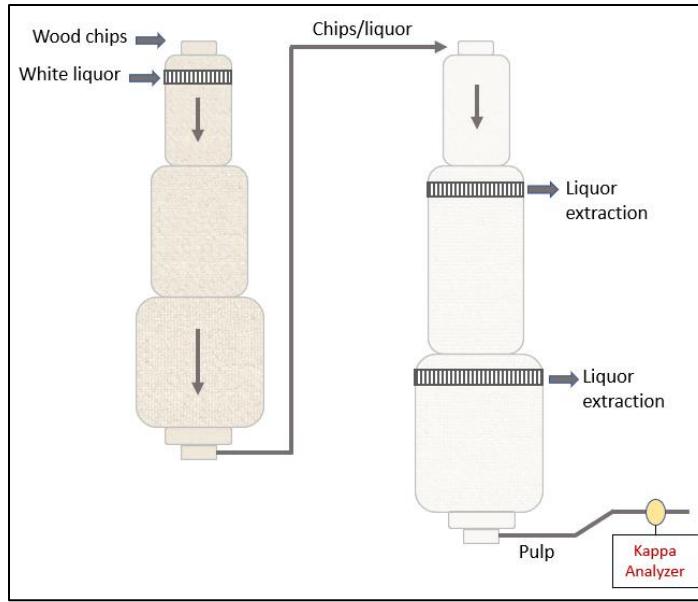


Figure A2: Kamyr digester for paper pulp production

3) Low-Density Polyethylene (LDPE) Process Data

The data comes from a LDPE production process using a simulated model that was tuned to match a typical industrial process⁸⁶. This dataset can be obtained from <https://openmv.net>. As shown in figure below, the process consists of a multi-zonal tubular reactor. The dataset contains 54 samples of 14 process variables and 5 product quality variables. The process variables include the reactor operating conditions (temperatures, pressure), feed flowrates, and the coolant temperatures. The quality variables include several molecular properties such as the average molecular weight of the polymer product.

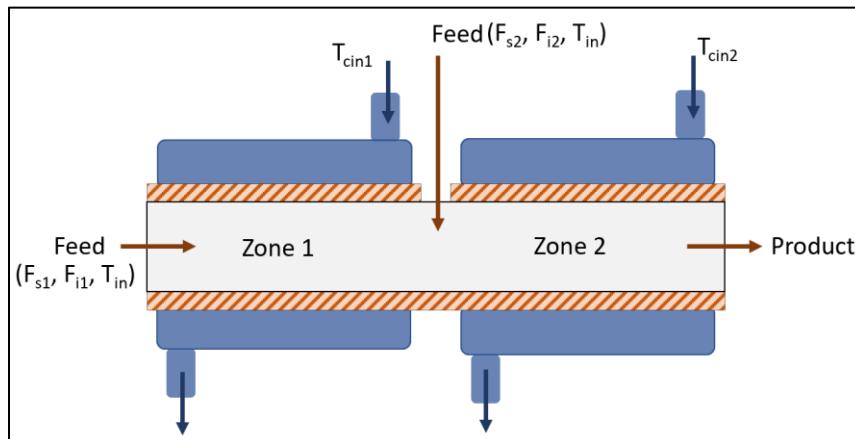


Figure A3: LDPE reactor

⁸⁶ MacGregor et al., Process monitoring and diagnosis by multiblock PLS methods, Process Systems Engineering, 1994

The feed and reactor operating conditions strongly influence the final product quality and therefore, an efficient FDD is highly desirable for such processes. For this dataset, a process fault occurs sample 51 onwards due to increase in feed impurity level. Figure A4 shows that fault detection using quality variables plot is not ‘obvious’. A couple of process variables do seem to show significant deviations, but it would be serendipitous if a plant operator was looking at these specific variables at the right time! An efficient and systematic fault detection mechanism using PLS is illustrated for this data in Chapter 5.

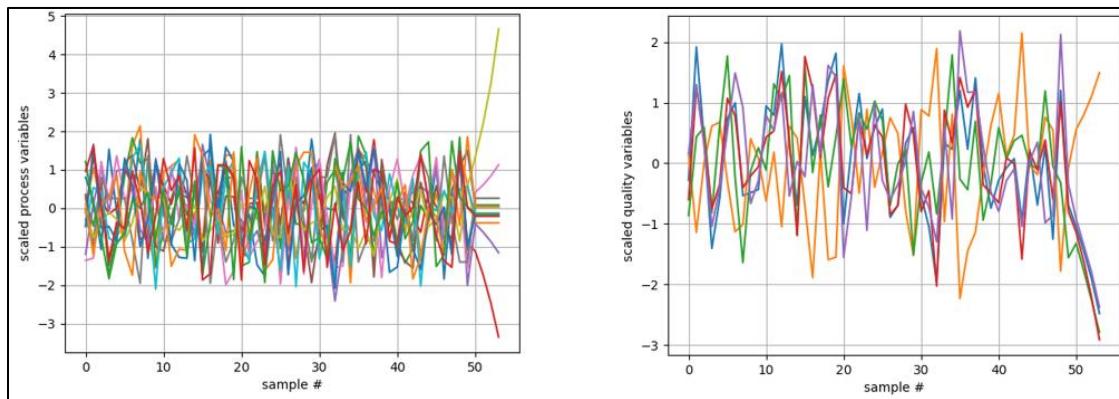


Figure A4: Scaled process (left) and quality (right) variables in LDPE dataset

4) Tennessee Eastman Process Data

This data is from a large-scale industrial chemical plant and is available at <https://github.com/camaramm/tennessee-eastman-profBraatz>. The process is called Tennessee Eastman process (TEP) and Figure A5 shows the process flowsheet.

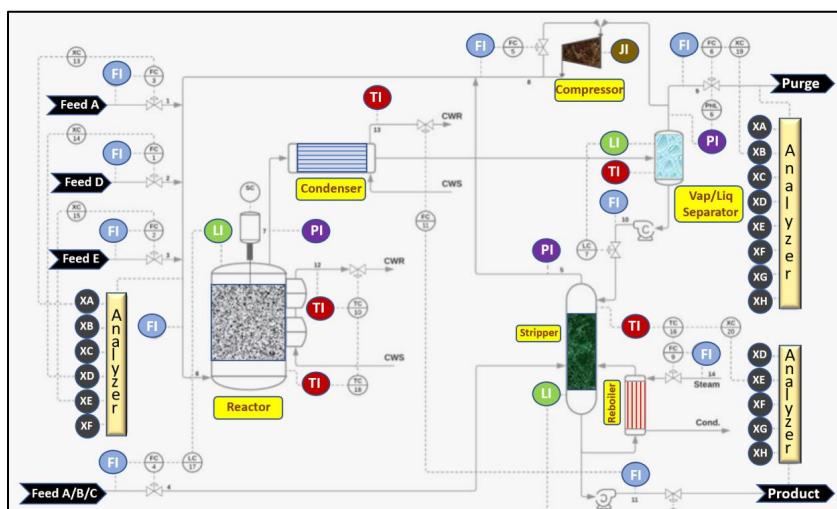


Figure A5: Tennessee Eastman process flowsheet⁸⁷

⁸⁷ 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/>).

This process consists of several unit operations: a reactor, a condenser, a separator, a stripper, and a recycle compressor. 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. 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’. This dataset has been used to build ICA and FDA-based fault detection tools in Chapter 6.

A bigger dataset for TEP has also been provided by Reith et al.⁸⁸ The dataset still contains 21 different fault classes (along with no-fault operation); however, for each fault class, 500 separate simulations runs were conducted for both training and test dataset. Each training simulation run contains 500 time samples from 25 hours of operation and each testing run contains 960 time samples from 48 hours of operation. LSTM-based fault classification has been demonstrated with this data in Chapter 12.

5) Semiconductor Manufacturing Process Data

This dataset is from a semiconductor manufacturing process and can be downloaded from <http://www.eigenvector.com/data/Etch>. The dataset was obtained from multiple batches from a metal etch 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. The dataset exhibits multimode operations and has been analyzed for fault detection in Chapter 7 using SVDD and Chapter 8 using GMM.

6) Polymer Pilot Plant Data

This dataset is from a Dupont polymer pilot plant and is available at <ftp://ftp.cis.upenn.edu/pub/ungar/chemdata/>. The dataset consists of 61 samples of 14 variables (10 predictors and 4 plant outputs). The predictors correspond to temperatures, feed rates, and so on. Although the specific details of these variables are not disclosed, the data contributors emphasize that this dataset is particularly good for testing the robustness of nonlinear modeling methods for irregularly sparse data. SVR-based soft sensor model is built for this data in Chapter 7.

⁸⁸ Reith, C.A., B.D. Amsel, R. Tran., and B. Maia. Additional Tennessee Eastman process simulation data for anomaly detection evaluation. Harvard Dataverse, Version 1, 2017

7) Debutanizer Column Data from a Petroleum Refinery

This dataset is from a debutanizer column in a petroleum refinery⁸⁹ and is provided as supplementary material at <https://link.springer.com/book/10.1007/978-1-84628-480-9>. 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. However, the location of the chromatograph sensor is downstream of the debutanizer column which introduces a delay in C4 content data availability. For better product quality control, real-time knowledge of C4 content is needed. For this, a soft sensor is desired that can predict C4 content using other process data (as shown in Figure A6) around the column which influence butane content in bottoms product.

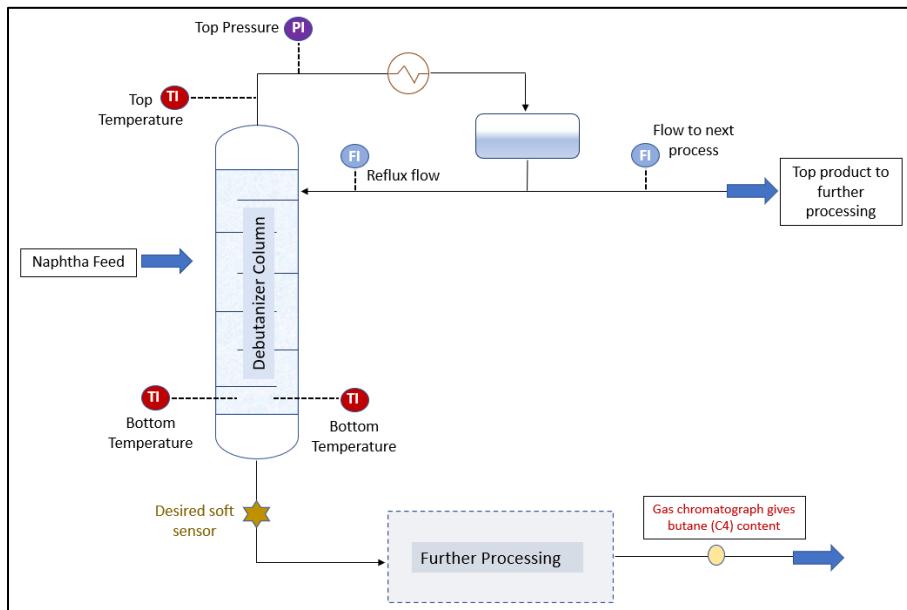


Figure A6: Debutanizer column with location of measured variables

The dataset contains 2394 samples of input-output process values. Note that the dataset is provided in its normalized form and the output values have been translated by 8 samples to compensate for the effect of the time delay. SVR and ANN have been used for soft sensor development in Chapters 7 and 11, respectively.

8) Concrete Compressive Strength Data

This dataset is available at the UCI machine learning repository (<https://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength>) and includes information about 1030 concrete samples. The UCI page emphasizes that the concrete

⁸⁹ Fortuna et. al., Soft sensors for monitoring and control of industrial processes, Springer, 2007

strength is a highly nonlinear function of concrete age and ingredients. In Chapter 9, a Random Forest regression model is built to capture this nonlinear relationship.

9) Wastewater Treatment Plant Data

The dataset (downloaded from UCI machine learning repository⁹⁰) comes from a wastewater treatment plant that uses activated sludge process to remove organic matter and suspended solids from municipal wastewater. In this process (Figure A7), the suspended solids are first physically settled (primary treatment) and then biologically treated to oxidize the biodegradable organic matter (secondary treatment). Data from on-line sensors at different stages of the process for 38 variables over 527 days of operation are provided. Seven out of the 38 variables characterize the effluent water quality. In Chapter 9, a XGBoost-based soft sensor model is built to predict output conductivity (COND-S) of the effluent stream.

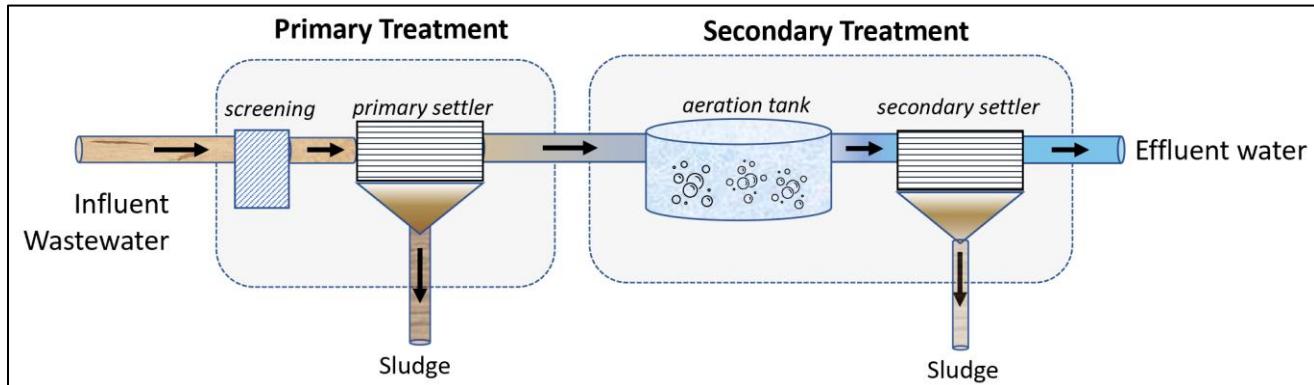


Figure A7: Simplified flow sheet of a typical wastewater treatment plant

10) Combined Cycle Power Plant Data

The dataset⁹¹ comes from a combined cycle power plant composed of gas turbine (GT), steam turbine (ST) and heat recovery steam generator as shown in Figure A7. 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. Measurements of ambient conditions (temperature, pressure, relative humidity) and ST exhaust steam pressure which influence the electrical output are also included. In Chapter 11, an ANN model is built to predict the electrical power as a function of the other signals.

⁹⁰ <https://archive.ics.uci.edu/ml/datasets/water+treatment+plant>

⁹¹ UCI Machine Learning Repository: <https://archive.ics.uci.edu/ml/datasets/combined+cycle+power+plant>

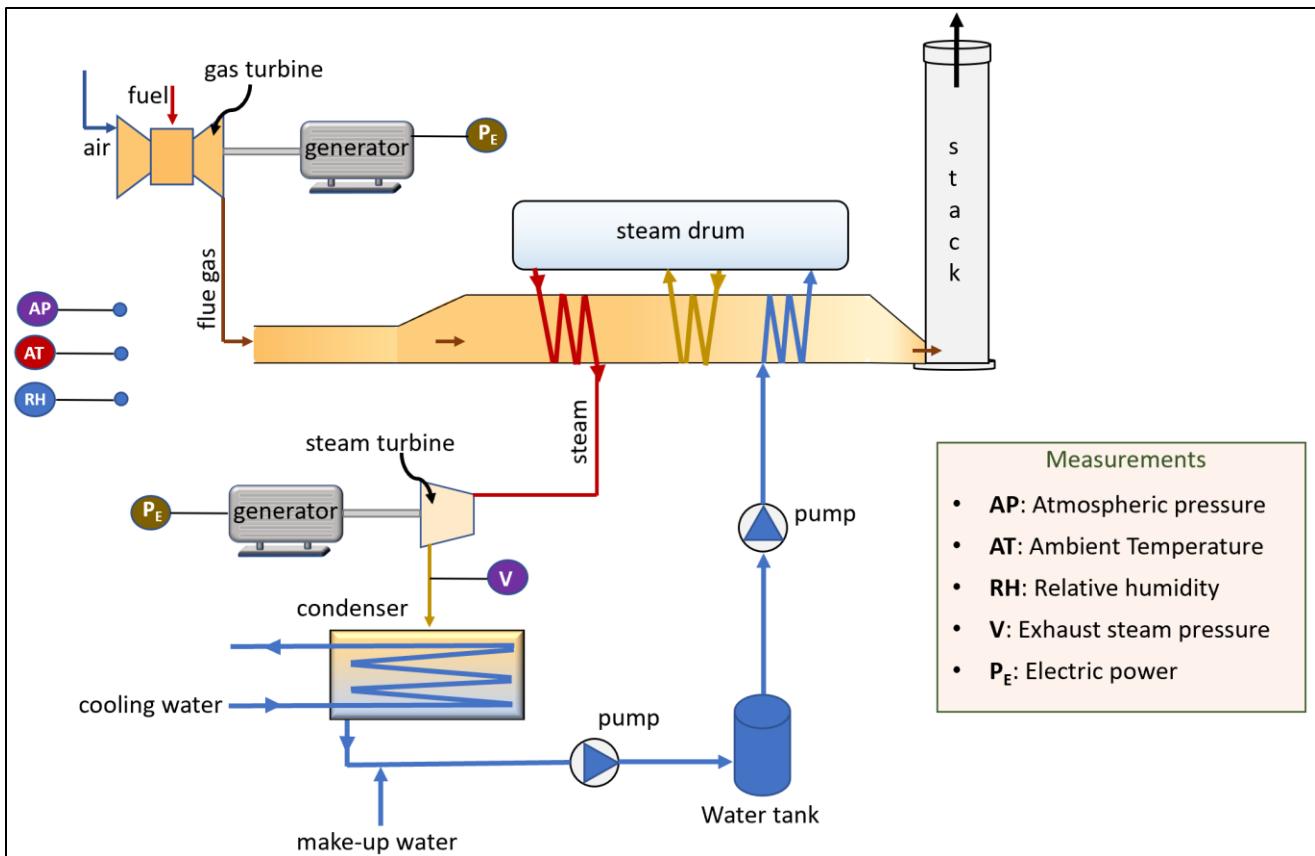


Figure A8: Simplified schematic of combined cycle power plant

11) SISO Heater System Data

This dataset is from a heater system where heater power output is manipulated to adjust the system temperature. The dataset contains 4 hours of training data and 14 minutes of test data (<https://apmonitor.com/do/index.php/Main/LSTMNetwork>). Data is sampled at every second. In Chapter 12, a LSTM model is built to predict the temperature using the current and past system data.

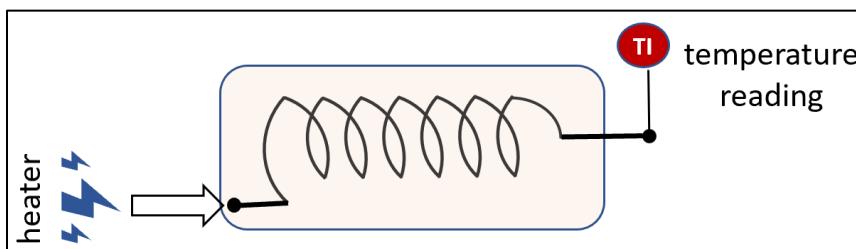


Figure A9: A heater system with a single temperature sensor

12) Gas Turbine Data

This dataset is available at NASA prognostics data repository⁹² and provides simulated data for aircraft gas turbine engines. The dataset 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 remaining useful lives for the engines have been provided for the test dataset. More details and step-by-step guide on building a predictive maintenance model is provided in Chapter 12.

End of the book

⁹² Turbofan engine degradation simulation data set, NASA. Originally obtained from <https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/>

By the process data scientists, for the process data scientists, of process data science



Machine Learning in Python for Process Systems Engineering

This book provides an application-focused exposition of modern ML tools that have proven useful in process industry and hands-on illustrations on how to develop ML solutions for process monitoring, predictive maintenance, fault diagnosis, inferential modeling, dimensionality reduction, and process control. This book considers unique characteristics of industrial process data and uses real data from industrial systems for illustrations. With the focus on practical implementation and minimal programming or ML prerequisites, the book covers the gap in available ML resources for industrial practitioners. The authors of this book have drawn from their years of experience in developing data-driven industrial solutions to provide a guided tour along the wide range of available ML methods and declutter the world of machine learning.

The following topics are broadly covered:

- *Fundamentals of machine learning workflow*
- *Practical methodologies for pre-processing industrial data*
- *Classical ML methods and their applications for process monitoring, fault diagnosis, and soft sensing*
- *Deep learning and its application for predictive maintenance*
- *Reinforcement learning and its application for process control*
- *Deployment of ML solution over web*