Machine Learning and Multivariate Statistics for Optimizing Bioprocessing and Polyolefin Manufacturing

Aman Agarwal

Dissertation submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

In Chemical Engineering

Y.A. Liu, Chair
Donald G. Baird
Sanket A. Deshmukh
Hongliang Xin

6 December, 2021
Blacksburg, VA

Keywords: foaming, antifoam profiles, fermentation, machine learning, multivariate statistics, ensemble methods, automated machine learning, deep learning

Machine Learning and Multivariate Statistics for Optimizing Bioprocessing and Polyolefin Manufacturing

Aman Agarwal

TECHNICAL ABSTRACT

Chemical engineers have routinely used computational tools for modeling, optimizing, and debottlenecking chemical processes. Because of the advances in computational science over the past decade, multivariate statistics and machine learning have become an integral part of the computerization of chemical processes. In this research, we look into using multivariate statistics, machine learning tools, and their combinations through a series of case studies including a case with a successful industrial deployment of machine learning models for fermentation. We use both commercially-available software tools, Aspen ProMV and Python, to demonstrate the feasibility of the computational tools.

This work demonstrates a novel application of ensemble-based machine learning methods in bioprocessing, particularly for the prediction of different fermenter types in a fermentation process (to allow for successful data integration) and the prediction of the onset of foaming. We apply two ensemble frameworks, Extreme Gradient Boosting (XGBoost) and Random Forest (RF), to build classification and regression models. Excessive foaming can interfere with the mixing of reactants and lead to problems, such as decreasing effective reactor volume, microbial contamination, product loss, and increased reaction time. Physical modeling of foaming is an arduous process as it requires estimation of foam height, which is dynamic in nature and varies for different processes.

In addition to foaming prediction, we extend our work to control and prevent foaming by allowing data-driven *ad hoc* addition of antifoam using exhaust differential pressure as an indicator of foaming. We use large-scale real fermentation data for six different types of sporulating microorganisms to predict foaming over multiple strains of microorganisms and build exploratory time-series driven antifoam profiles for four different fermenter types. In order to successfully predict the antifoam addition from the large-scale multivariate dataset (about half a million instances for 163 batches), we use TPOT (Tree-based Pipeline Optimization Tool), an automated genetic programming algorithm, to find the best pipeline from 600 other pipelines. Our antifoam profiles are able to decrease hourly volume retention by over 53% for a specific fermenter. A decrease in hourly volume retention leads to an increase in fermentation product yield.

We also study two different cases associated with the manufacturing of polyolefins, particularly LDPE (low-density polyethylene) and HDPE (high-density polyethylene). Through these cases, we showcase the usage of machine learning and multivariate statistical tools to improve process understanding and enhance the predictive capability for process optimization.

By using indirect measurements such as temperature profiles, we demonstrate the viability of such measures in the prediction of polyolefin quality parameters, anomaly detection, and statistical monitoring and control of the chemical processes associated with a LDPE plant. We use dimensionality reduction, visualization tools, and regression analysis to achieve our goals. Using advanced analytical tools and a combination of algorithms such as PCA (Principal Component Analysis), PLS (Partial Least Squares), Random Forest, etc., we identify predictive models that can be used to create inferential schemes.

Soft-sensors are widely used for on-line monitoring and real-time prediction of process variables. In one of our cases, we use advanced machine learning algorithms to predict the polymer melt index, which is crucial in determining the product quality of polymers. We use real industrial data from one of the leading chemical engineering companies in the Asia-Pacific region to build a predictive model for a HDPE plant. Lastly, we show an end-to-end workflow for deep learning on both industrial and simulated polyolefin datasets.

Thus, using these five cases, we explore the usage of advanced machine learning and multivariate statistical techniques in the optimization of chemical and biochemical processes. The recent advances in computational hardware allow engineers to design such data-driven models, which enhances their capacity to effectively and efficiently monitor and control a process. We showcase that even non-expert chemical engineers can implement such machine learning algorithms with ease using open-source or commercially available software tools.

Machine Learning and Multivariate Statistics for Optimizing Bioprocessing and Polyolefin Manufacturing

Aman Agarwal

GENERAL AUDIENCE ABSTRACT

Most chemical and biochemical processes are equipped with advanced probes and connectivity sensors that collect large amounts of data on a daily basis. It is critical to manage and utilize the significant amount of data collected from the start and throughout the development and manufacturing cycle. Chemical engineers have routinely used computational tools for modeling, designing, optimizing, debottlenecking, and troubleshooting chemical processes. Herein, we present different applications of machine learning and multivariate statistics using industrial datasets.

This dissertation also includes a deployed industrial solution to mitigate foaming in commercial fermentation reactors as a proof-of-concept (PoC). Our antifoam profiles are able to decrease volume loss by over 53% for a specific fermenter. Throughout this dissertation, we demonstrate applications of several techniques like ensemble methods, automated machine learning, exploratory time series, and deep learning for solving industrial problems. Our aim is to bridge the gap from industrial data acquisition to finding meaningful insights for process optimization.

Acknowledgment

I would like to take this opportunity to thank my advisor, Dr. Y.A. Liu for facilitating my industrial project journey, providing me with all the necessary resources for research, and helping me enhance my writing skills. I am very grateful that he primed me for research and helped me get ready for the industrial world with his insights.

I would also like to thank my committee members, Dr. Donald Baird, Dr. Sanket Deshmukh, and Dr. Hongliang Xin for their support, guidance, and availability. I thank Dr. Luke Achenie for his words of encouragement in the initial phase of my research. Additionally, I would like to thank all the professors and staff members in the Chemical Engineering Department at Virginia Tech.

I would like to thank my industrial project mentor, Dr. Chris McDowell, Novozymes Biologicals Inc., for giving me the opportunity, motivation, and valuable feedback on every step of my research. Thank you very much for your continuous mentorship throughout my Ph.D. I would also like to thank Luke Dooley, Novozymes Biologicals Inc., for his continuous support, availability, and cherished resourcefulness for all my projects with Novozymes. Additionally, I would like to thank Dr. Mads Thaysen, Novozymes A/S for his professional advice, availability, and continuous support.

I am very grateful to Aspen Technology and Novozymes Biologicals for their continuous support throughout my graduate studies at Virginia Tech. I would also like to thank MATRIC research institute for allowing me to explore industrial research as a summer intern. I would like to thank Steven Hedrick, Don Bunnings, Brooke Albin, Dana Jividen, Rob Nunley, and everyone from MATRIC for their hospitality and challenging learning opportunities. I would also like to thank Dr. Kevin Seavey, DOW Chemicals, for his availability and help with motivation. I would also like to thank my labmates and friends for their continuous support.

Lastly, I would like to extend my sincerest gratefulness to my loving parents, my father Surendra Agarwal and my mother Shanta Agarwal, for their unconditional love and guidance throughout my graduate studies. I would also like to thank my brother, Rishu Agrawal for his brotherly guidance and love. I dedicate my dissertation to my parents. Without them, my graduate journey would not have been possible.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction to the Dissertation

Machine Learning and Multivariate Statistics for Optimizing Bioprocessing and Polyolefin Manufacturing

1.1 Scope and organization of the dissertation

The contents of the dissertation are as follows. Chapter 2 demonstrates the usage of various multivariate statistical algorithms to monitor and quality control an LDPE plant. In this chapter, we look into the strength and weaknesses of common statistical machine learning algorithms and determine their viability for a particular regression task.

Chapter 3 includes building a data-driven soft-sensor for a continuous HDPE plant from a reputed chemical industry in the Asian-Pacific region. In this chapter, we demonstrate the possibility of building soft sensors for measurements that are difficult to extract in real-time. Chapter 4 proposes a novel method for fermenter classification and foaming prediction using ensemble-based machine learning. In this chapter, we demonstrate how we can use exhaust differential pressure (an indicator of foaming) to predict the onset of foaming in an ad hoc manner. We also validate the integration of the batch datasets with accurate classification models.

Chapter 5 is an extension to the research done in Chapter 4, as we work with large datasets for industrial implementation. In this chapter, we showcase innovative defoaming strategies using automated machine learning and exploratory time-series analysis. We further establish our proof-of-concept (PoC) by deploying machine learning (ML)-guided antifoam profiles in the fermentation plant and gauging the impact of the profiles in integrated and batch tests.

Chapter 6 provides a step-by-step guide to chemical engineers on deep learning applications for both continuous industrial process (time-independent) and a simulated dynamic batch process (time-dependent). In this chapter, we help readers walk through building deep neural networks including three different types of recurrent neural networks for an HDPE plant. Chapter 7 summarizes the dissertation and provides future outlooks for the research.

Appendix A serves as a tutorial for chemical engineers new to Python language with useful examples and basic concepts of the language. Appendix B is the supplementary material for Chapter 6 as it introduces chemical engineers to the fundamentals of neural network.

1.2 Significant novel contribution of the research

1. Ensemble-Based Machine Learning for Industrial Fermenter Classification and Foaming Control

- We build accurate ensemble-based classification models to differentiate fermenter types on the basis of known independent variables alone, without prior knowledge of fermenter design specifications, thus allowing for data integration of multiple plant data sets to build better prediction models.
- We accurately predict the onset of foaming based on exhaust differential pressure using both classification and regression models.

2. Large-Scale Industrial Fermenter Foaming Control: Automated Machine Learning for Antifoam Prediction and Defoaming Process Implementation

- We demonstrate the ability of automated machine learning (AML) to predict antifoam addition for multiple strains of microorganisms using large-scale industrial fermentation dataset.
- We establish proof of concept (PoC) by using time-series exploratory analysis to build strain-specific and deployment-ready antifoam profiles for four different fermenter designs. We confirm the industrial impact of the deployed profiles based on the initial results from the plant. From our initial deployment results, we see significant improvements in yield, shown by the significant decrease in hourly volume retention (up to 53%) and decrease in the count of exhaust differential pressure (a foaming indicator) exceeding a threshold of 100 mbarg, among other performance indicators.

3. Deep Learning Optimization on Industrial and Simulated Polyolefin Datasets

- We demonstrate how to build a steady-state (time-independent) model for a high-density polyethylene (HDPE) industrial plant to predict the melt index.
- We show how to build three different types of dynamic recurrent neural network from scratch to predict the melt index from a simulated time-dependent polymer dataset.

Chapter 2. Multivariate Data Analysis of Polyolefin Quality Parameters for a LDPE plant

2.1 Introduction

Multivariate data analysis is a form of predictive analytics which uses a training dataset to build a mathematical model for tasks such as process monitoring, identification of critical parameters, assessment of process variability, scale-up, and inferential control. One such application of multivariate data analysis is quality control. We often use quality parameters like melt flow rate (melt index), weight-average molecular weight, and number-average molecular weight, etc. to assess the polymer quality. The measurements for these quality parameters are often made off-line in a laboratory setting and are recorded on an hourly to a daily basis. Some of the biggest challenges of such measurements lie within the data collected. The measurements are often noisy, sparse, and infrequent, making them unreliable and imprecise. In this chapter, we look at using different multivariate statistics and machine learning algorithms to predict these off-line quality parameters using multi-output regression and different statistical methods to analyze a given set of process data.

Process data from chemical industries are generally high-dimensional, non-causal, sparse, and noisy. By using different algorithms like PCA (Principal Component Analysis), PLS (Partial Least Squares), decision trees, random forests, etc. and visualization plots like loading plots, biplots, barcharts, etc. we can successfully extract the desired insights from a raw dataset. In this chapter, we look at simulated LDPE data by MacGregor et al. generated by using a model of Kiparissides and Mavridis. [1-2] Polyolefins, such as low-density polyethylene (LDPE), high-density polyethylene (HDPE), and propylene (PP), are commercially significant thermoplastics manufactured throughout the world for food packaging, clothing, bags, toys, containers, pipes, etc. [3] LDPE are usually produced in a tubular reactor under high pressure. Properties like molecular weight and melt index are often very difficult to measure on a frequent interval to ensure final product quality. In order to mitigate the expense associated with the measurement, we use multivariate statistics and machine learning algorithms to predict the polyolefin quality parameters. The resulting multi-output regression model, built on the existing operational logs of temperature sensors, wall temperature probes, and solvent flow rate, provides us with frequent on-line quantitative estimates of the polyolefin quality parameters.

2.2 Literature Review

With the increasing interest in digitalization of the chemical industry and integration of data science in the chemical industry, many researchers have been exploring different pathways to apply multivariate statistics and machine learning algorithms for process monitoring and quality control. [4-7] Wang et al. use partial least squares (PLS), support vector machines (SVMs), and random forest machine learning

algorithms to predict several quality parameters of organic polymers. [8] Ersen et al. use multiple algorithms like K-NN (k- nearest neighbors), decision trees, random forest, etc. to analyze fracture behavior of polymer composites. [9] Hartman et al., used artificial neural networks to derive potential rate constants. [10] Similarly, Baughman and Liu, in their 1995 neural network book, describe in detail how artificial neural networks can be used for a variety of tasks including prediction, classification, diagnosis, and control applications. [11]

2.3 LDPE data structure

The dataset generated by the Kiparissides and Mavridis model include variations in the four variables: wall temperature (T_w), solvent-flow rate (S), heat transfer coefficient for the wall on the cooling side (H_w), and initial initiator concentration in the feed. The reactor is a tubular reactor and the feed includes a monomer ethylene, an initiator, and a chain transfer agent. These authors measure temperatures at 20 different locations in the reactor, and simulated polymer properties including the number-average molecular weight (M_n). weight-average molecular weight (M_w), frequency of long chain branching (LCB), frequency of short chain branching (SCB), the content of vinyl groups (VNL), and vinylidene groups (VND) in the polymer chain. Table 1 shows the input and output variables for the multi-output regression models.

The polymerization reaction is a highly exothermic reaction and therefore the reactor temperature rapidly increases as we move from T_1 to T_{20} and eventually begins to plateau as it reaches its peak. The reactor wall begins to slowly heat the reactor and generates heat until a maximum temperature is reached; the peak is known as the hot-spot, which occurs when the initiator is completely consumed. All the changes in the rate of heat transfer due to fouling of the inner walls and the change in the initiator concentration are recorded in the temperature profiles alongside the reactor. The dataset consists of 22 input variables (X) and 6 output variables (Y), which are listed below in Table 2.1.

Table 2.1: List of process variables and product quality variables for the LDPE reactor

Input (Process Variables)		**Output (Product Quality Variables)**	
T_1-T_{20}	Temperature measurements along the reactor	**MWI_e5**	Weight average molecular weight inverse (X 10^5)
		MNI_e6	Number average molecular weight inverse (X 10^6)
T_w	Wall temperature		
		LCB	Long chain branching frequency
S	Solvent Flow Rate		
		SCB	Short chain branching frequency
		VNL	Frequency of vinyl groups in polymer chain
		VND	Frequency of vinylidene groups in polymer chain

2.4 Data Visualization and Process Monitoring

Most process measurements acquired from distributed control systems (DCS) in industries are high dimensional, have low signal-to-noise ratios, and have missing data. Multivariate statistical monitoring using visualization techniques and projection of the raw dataset into a transformed plane helps to interpret the dataset. We use Aspen ProMV, a commercially available software readily available to academic users at low cost for building the Principal Component Analysis (PCA)/ Partial Least Squares(PLS) models and the required visualizations for anomaly detection, dimensionality reduction, and exploration of variable relations in section 2.4.

2.4.1 PCA algorithm and model building

PCA is defined as an orthogonal linear transformation that transforms the data to a new coordinate system by projecting the dataset into a new hyperplane. The scalar projection is done such that the

greatest variance comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.[12] Eq. (1) defines this transformation:

$$\mathrm{X} = \mathrm{TP}^T + \mathrm{E} \quad (1)$$

Here, X represents the data matrix, T represents the score matrix, P is the loading matrix, and E is the residual matrix. Re-writing Eq. (1) gives the relationship for the prediction error E:

$$\mathrm{E} = \mathrm{X} - \mathrm{TP}^T = \text{Observed value} - \text{Model Approximation} = \text{Prediction Error} \quad (2)$$

We build the PCA model on the 22 input variables and transform the high dimensional input set into three principal components as shown in Figure 2.1.

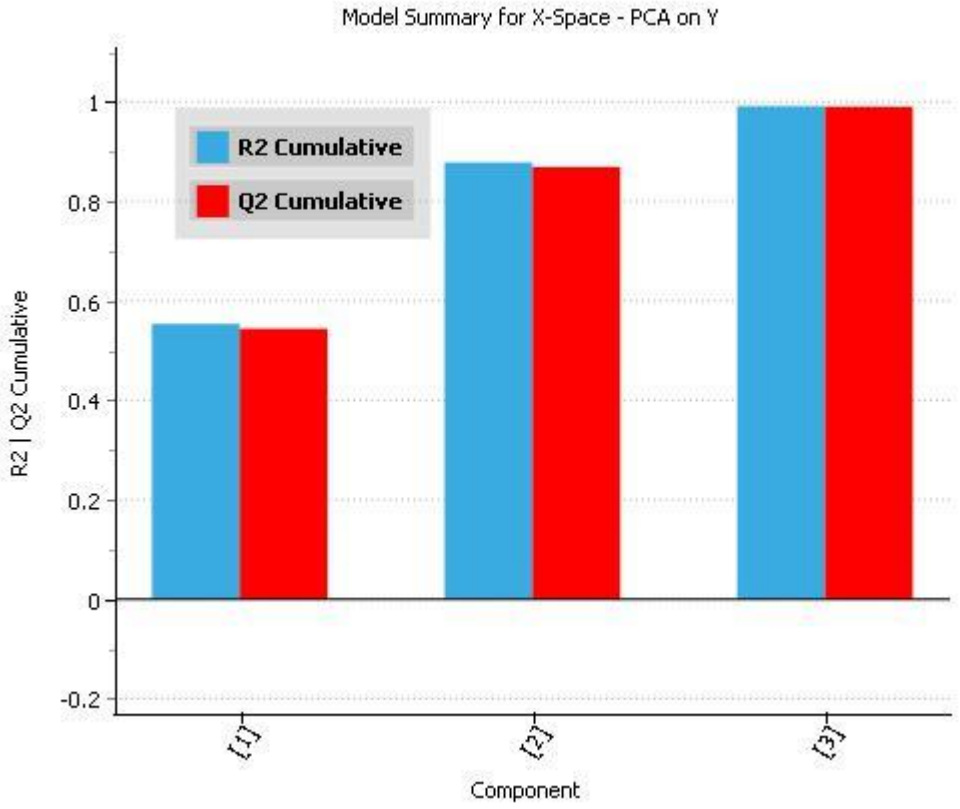


Figure 2.1. PCA plot with 3 principal components for X process variables (22 input variables).

The R2 cumulative score summarizes the fit error on all training data, while the Q2 cumulative score is the prediction ability over the test dataset. Equations. (3) and (4) define these scores:

$$R_X^2 = \frac{SS(X) - SS(E_{train})}{SS(X)} \quad (3)$$

$$Q_X^2 = \frac{SS(X) - SS(E_{test})}{SS(X)} \quad (4)$$

where SS=Sum of squares and E_{train} and E_{test} are the prediction errors for the training and test datasets.

Figure 2.1 shows that with three principal components, we are able to successfully fit the dataset (since both R_X^2 and Q_X^2 are close to 1).

2.4.2 Anomaly detection

After using PCA for dimensionality reduction, we can now use Hotelling's T^2 in junction with score plot for building a hyper-ellipse score plot. Figure 2.2 shows the PCA score plot along with the Hotelling's T2 tolerance ellipse of 95% confidence. Hotelling's T2 is a summary statistic for all the scores taken together, given by:

$$T_i^2 = \frac{t_{i1}^2}{s_1^2} + \frac{t_{i2}^2}{s_2^2} + \cdots + \frac{t_{iA}^2}{s_A^2} \tag{5}$$

where, t's are the latent (transformed) variables whose values are called scores., 's' represents variance of the latent variable in the model. From Figure 2, we can see that some observations depart from the cluster as outliers and observations 55 and 56 are outside the 95% confidence interval. Using this plot, we can see the trend from 'outlier' or 'anomaly' observation 51-56 and observation 37.

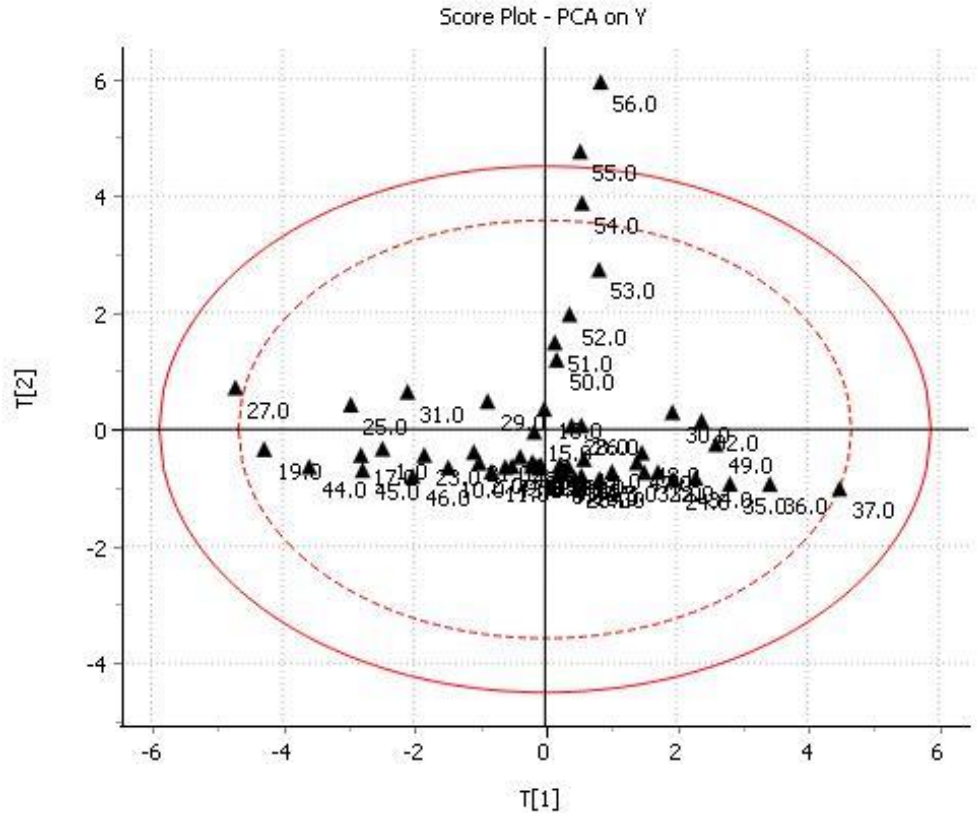


Figure 2.2. PCA hyper-ellipse score plot on product quality variables.

2.4.3 PLS algorithm

PCA calculates the latent variables based only on the input (X) variables, and Partial Least Squares (PLS) quantifies the latent variables based on both input and output (Y) variables. The overall objective function of PLS is given by:

$$Objective\ Function = argmax\ (Covariance) \tag{6}$$

Like before for X space in PCA, PLS modifies both the X-space and Y-space while ensuring maximum covariance as shown in Figure 2.3. t' s and u' s are the transformed variables and their values represent the projection scores. Equations. (7) and (8) define these scores as:

$$t_a = X_a w_a \text{ , for the X} - \text{space} \tag{7}$$

$$u_a = Y_a c_a \text{ , for the Y} - \text{space} \tag{8}$$

where, w_a represents the loading values for X, and c_a represents the loading values for Y.

The covariance is given by:

$$Cov(t_a, u_a) = \frac{1}{n-1}\sum_{i=1}^{n}\{(t_a - \bar{t}_a)(u_a - \bar{u}_a)\} \tag{9}$$

where, $\bar{t}_a$ is the mean of 't' projection scores and $\bar{u}_a$ is the mean of 'u' projection scores respectively.

In PLS, we try to maximize this covariance or the dot product of $t_a' u_a$.

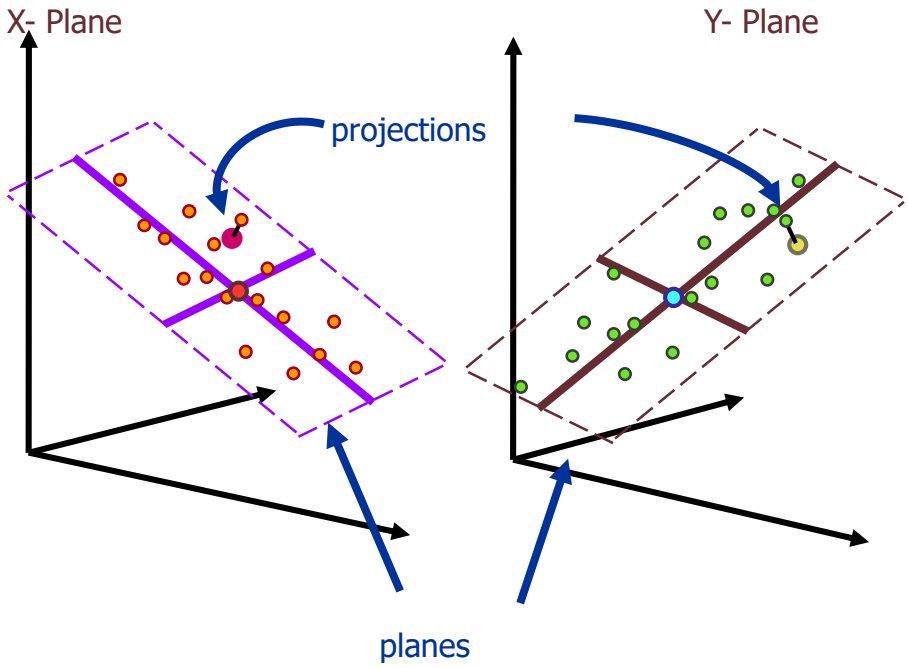


Figure 2.3. PLS projection plot.

2.4.4 PLS model building and variable relationships

As shown in Figure 2.4, we build a PLS model with 8 components as the Q_X^2 score reaches its maximum possible value for the dataset. We remove the seven outliers {(51-56), 37} identified above before building the PLS model. With 8 components, we are able to get a cumulative Q_X^2 score of 0.8, which is reasonable for the small dataset. To further visualize the predictions made by the model, we plot the actual observation vs predicted observation for the number-average molecular weight in Figure 2.5. We see that the predicted values for the number-average molecular weight (MN_e6) are close to the actual values with a high R_X^2 score and a low root-mean squared error estimate (RMSEE).
PLS can help us identify which variable is important through a variable important plot (VIP). Figure 2.6 shows the key process variables that impact the product quality parameters the most. We see that the solvent flow rate has a very high VIP score, which makes it an important variable to consider, while deciding to adjust the product quality. Along with the solvent flow rate, T7, T6, T_w, and T4 are also very important as they all have a score greater than or equal to one. VIP scores greater than one represent important variables.

Now that we know the important variables, we use the same model to visualize the relationships between X and Y variables. We make a loading plot, which has all the weights associated with the variables. Highly correlated variables have similar weights and since we do not look at the weights

directly, we see the effect and directionality of each of the original variables. From Figure 2.7, we see several relationships between the input and output variables and the variables by themselves. As expected, the temperatures closer to one side of the reactor are clustered together and are opposite to the temperature profiles on the other side. Also, in order to get higher frequency of LCB (long chain branching parameter), we have to make sure that the reactor inlet temperature is not too high and the solvent rate is low as well. However, to get a higher frequency of SCB (short chain branching parameter) and increased molecular weight, we need a high inlet temperature and higher solvent rate.

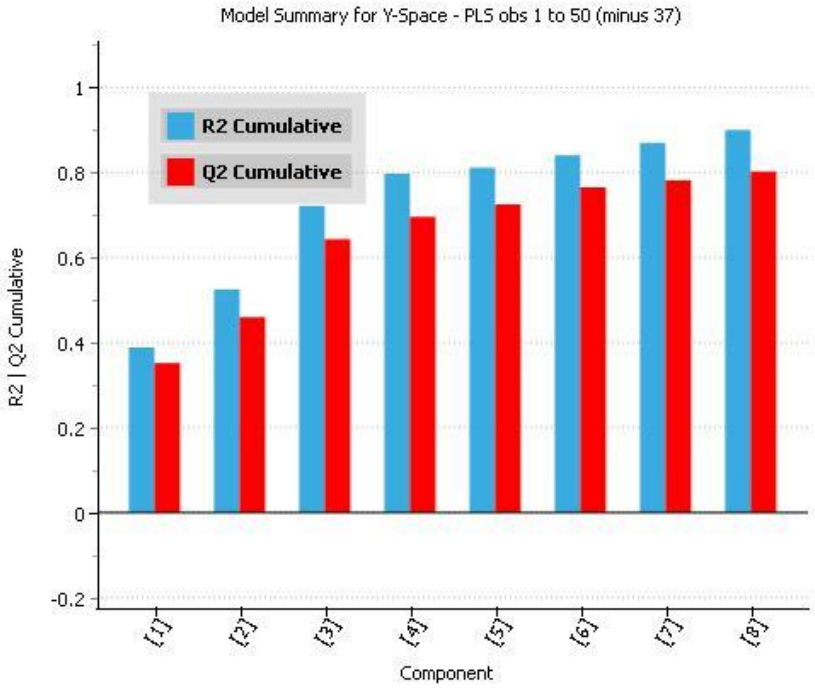


Figure 2.4. PLS plot with 8 components without outliers.

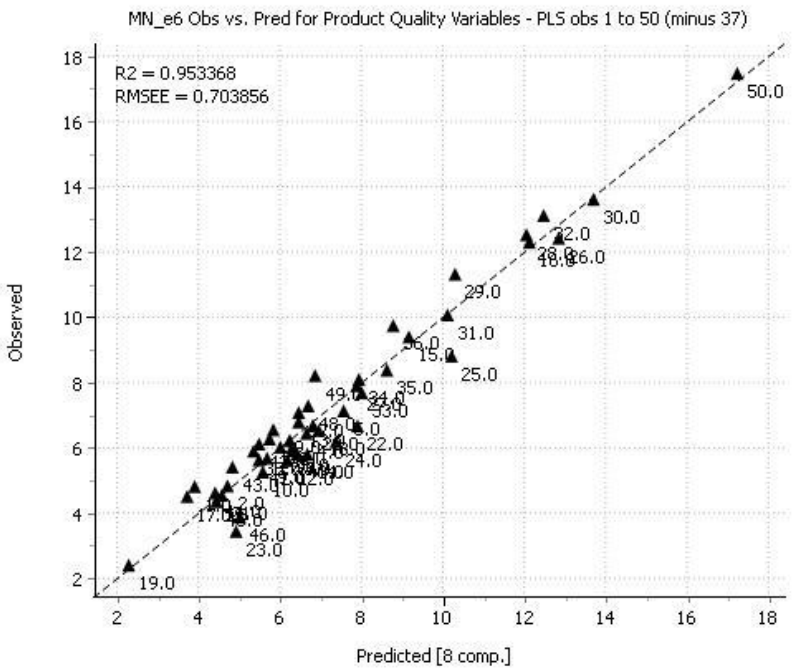


Figure 2.5. PLS observed vs predicted plot for the number-average molecular weight.

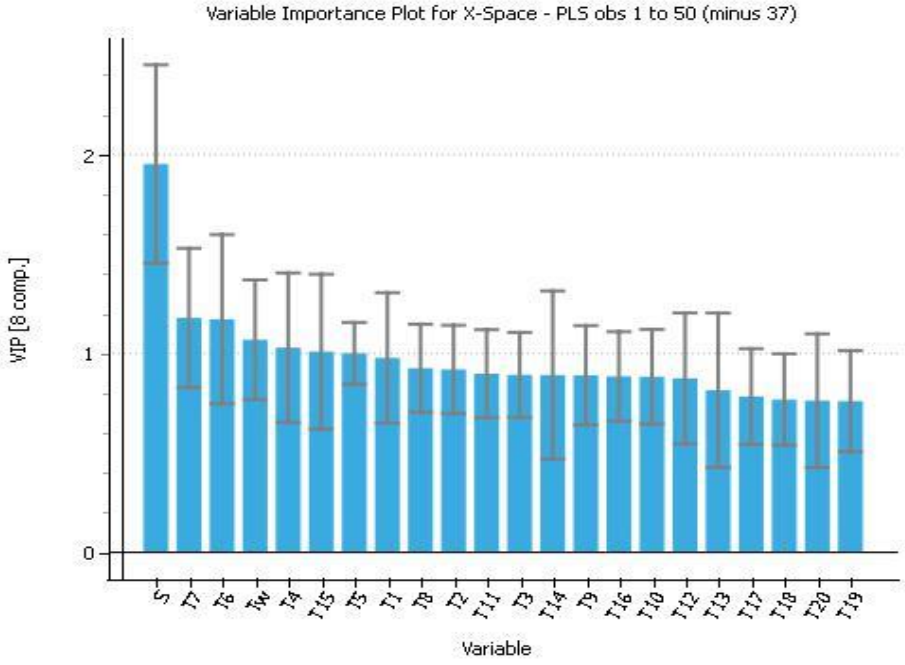


Figure 2.6. Variable importance plot based on PLS.

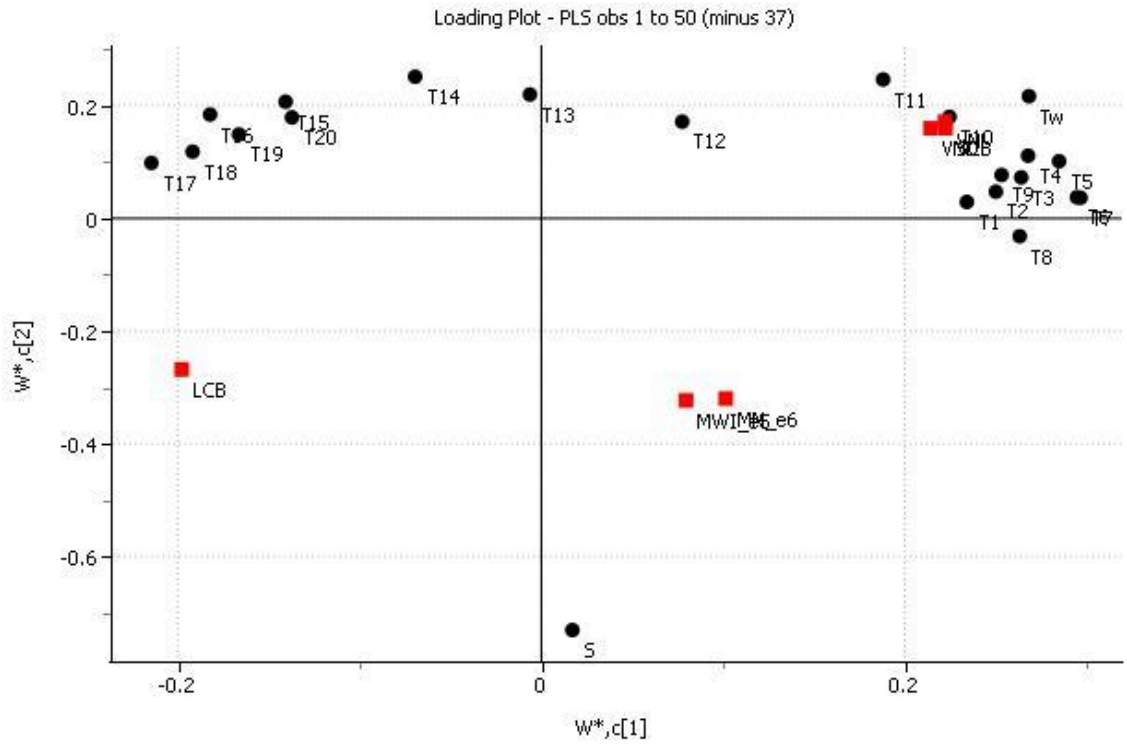


Figure 2.7. Loading plot (X&Y) based on PLS.

2.5 Multi-output regression model building and results

Multi-output regression models are problems that involve using multivariate statistics and machine learning algorithms to predict two or more output values given a dataset. Most machine learning algorithms do not natively support predicting multiple outputs and predict a single numerical value. Such algorithms (like support vector machine) require wrappers or linear sequences of models to give multiple output values. For our purpose, we look at some algorithms known for multi-output regression. We develop all the regression models built below using Python libraries, which are commercially free open-source resources.

2.5.1 Linear regression algorithm

Linear regression is the simplest machine learning algorithm used for multi-output regression. The objective of a linear regression model is to find a relationship between one or more features (independent variables) and a continuous target variable (dependent variable). We represent the linear regression by Eq. (10):

$$Y = \theta^T x \tag{10}$$

where, θ^T is the model parameter which includes the bias term and x is the feature vector.

2.5.2 K-Nearest neighbor algorithm

K-NN algorithm is a distance-based instant learning algorithm, which is non-parametric and gives results based on proximity. The algorithm uses several distance measures for calculating the distance between two points; for our model, we choose the standard Euclidean distance, defined by Eq. (11) for two points x and y:

$$Euclidean\ distance(d) = \sqrt{\sum_{i=1}^{k}(x_i - y_i)^2} \quad (11)$$

After calculating the distances, the K-NN algorithm finds a given observation value based on the values of the nearest neighbors. We choose the number of neighbors (k) based on grid search. The number of neighbors cannot be more than the number of instances in the test set.

2.5.3 Decision tree regression algorithm

Decision tree is a frequency-based algorithm which builds models in the form of a tree structure. A decision tree consists of three types of nodes:

- Root nodes or just nodes are the top decision node and represent an entire population or a sample.
- Decision nodes resulting from splitting sub-nodes into further sub-nodes.
- Leaf/Terminal nodes are the bottom nodes with no further split.

We apply regression trees for continuous quantitative target variables. For regression, the most common splitting criterion for the trees is the weighted variance of the nodes. The algorithm splits the nodes to minimize the variation in nodes after the split.

2.5.4 Random forest regression algorithm

Random Forest is simply a combination of the decision trees which follows the principle of bootstrap aggregating (or bagging) to reduce complexity by training each decision tree on a different data sample, where sampling is done with replacement. Random Forest has high tolerance for multidimensional data and yields a parallel ensemble of unstable learners, which together provide a strong learner, resulting in lower bias and variance.

The biggest difference between decision trees and random forest is that decision trees are prone to overfitting and do not handle noise as well as random forest. Random forest provides an aggregate solution which minimizes the variance errors and bias errors.

2.5.5 Model validation and results

We build five different regression models for multi-output regression prediction of the six polyolefin quality parameters. Before building the models, we split the dataset into training and testing sets (80-20

split). We also remove all the outliers identified in the earlier sections. In order to preserve the small dataset from further splits, we use cross-validation to divide the training set into folds for validation purposes. We use 10-fold cross validation, where the training set is divided into 10 folds and the model is trained on the 9-folds. We validate the resulting model on the remaining part of the data. By repeating the process, we use all of the training data to build the model. We choose 8 components for PLS based on the previous R^2 scores. Similarly, we build a grid of {3,5,7,8,9} neighbors and pick 9 nearest neighbors for best results from trial and error.

Table 2.2 shows the results obtained for each of the regression models. We compare the standard deviation of the output quality variables with the root mean square error (RMSE) of the predicted values. RMSE is the square root of the average of squared errors. We use the Q^2 score to represent the accuracy of the fit of the model on the test set. A negative Q^2 score indicates a fit worse than the standard deviation. We compare the results from five different algorithms: linear regression, k-nearest neighbor, decision trees, random forest, and partial least squares.

From Tables 2.2 and 2.3, we see that a single algorithm does not predict all the output variables accurately. Because of the complexity of a multivariate dataset, a simple linear regression aggression fails to predict all the variables with high accuracy. However, the linear regression predicts short chain branching (SCB) frequency and vinylidene frequency in polymer chain, fairly well. It is crucial to identify linear relationships between variables in datasets. However, dependent variables which show non-linear relations with the independent variables require more sophisticated non-linear algorithms.

We notice that the K-nearest neighbor algorithm gives poor performance for all output variables. We expect this result as the K-NN algorithm works better on large datasets and is prone to misleading results for a smaller dataset. Decision trees give the best results for weight-average molecular weight and long chain branching (LCB) frequency. Decision trees outperforms random forests for smaller datasets and are more prone to overfitting compared to random forests. Decision trees are, however, faster for relatively large datasets and easily interpreted. The random forest algorithm gives the best prediction of the frequency of vinyl group addition. Random forest algorithm is effective for noisy non-linear datasets and larger datasets. We can further fine-tune the random forest algorithm for better performance, while regularizing to restrict overfitting. The PLS algorithm gives the most accurate predictions for the number-average molecular weight, and this observation is similar to that obtained using AspenProMV for a different testing set and split between training and testing datasets.

Table 2.2. Multi-output regression results for different machine learning algorithms.

Model	**Y1**	**Y2**	**Y3**	**Y4**	**Y5**	**Y6**
Linear Regression	**MWI_e5**	**MNI_e6**	**LCB**	**SCB**	**VNL**	**VND**
Q2_score	-0.6744	0.7230	0.3548	0.8024	0.1643	0.9224
RMSE	1.2727	1.9610	0.0128	1.2994	0.0011	0.0031
STD	0.9836	3.7258	0.0160	2.9232	0.0012	0.0110
k-Nearest Neighbor	**MWI_e5**	**MNI_e6**	**LCB**	**SCB**	**VNL**	**VND**
Q2_score	-0.1966	-0.1840	-0.1628	-0.1508	-0.1740	-0.1504
RMSE	1.0929	4.1398	0.0178	3.2480	0.0013	0.0122
STD	0.9836	3.7258	0.0160	2.9232	0.0012	0.0110
Decision Tree	**MWI_e5**	**MNI_e6**	**LCB**	**SCB**	**VNL**	**VND**
Q2_score	0.2616	0.0192	0.8697	0.3892	0.2834	0.3820
RMSE	0.8452	3.6898	0.0058	2.2846	0.0010	0.0086
STD	0.9836	3.7258	0.0160	2.9232	0.0012	0.0110
Random Forest	**MWI_e5**	**MNI_e6**	**LCB**	**SCB**	**VNL**	**VND**
Q2_score	0.13769	-0.2042	0.783599	0.549067	0.579419	0.580188
RMSE	0.9133	4.0886	0.0074	1.9630	0.0008	0.0071
STD	0.9836	3.7258	0.0160	2.9232	0.0012	0.0110
PLS	**MWI_e5**	**MNI_e6**	**LCB**	**SCB**	**VNL**	**VND**
Q2_score	-3.7036	0.8476	-1.0569	0.4262	-0.9805	0.8594
RMSE	2.1331	1.4545	0.0229	2.2143	0.0016	0.0041
STD	0.9836	3.7258	0.0160	2.9232	0.0012	0.0110

Table 2.3. Summary Results

Product Quality Variables	Highest Q2_score
MWI_e5	Decision Tree
MNI_e6	PLS
LCB	Decision Tree
SCB	Linear Regression
VNL	Random Forest
VND	Linear Regression

2.6 Conclusion

Using different tools of machine learning and different statistical algorithms, we are able to successfully showcase ways to monitor and quality control a LDPE reactor. We can use simple methods of dimensionality reduction and visualization such as PCA for anomaly detection. Using methods like PLS, we are able to identify inter-variable relationships and relationships between input and output variables. Multi-output regression models enable us to predict the infrequently measured quality parameters for real time control and assessment of polymer quality. The choice of algorithm used for analysis depends on the nature of the data: linear or non-linear, sparse or complete, noisy or consistent, balanced or unbalanced, small data volume or large data volume, interpretation, time, and processing power. One algorithm does not fit every dataset or in this case every output variable. However, the knowledge of the available algorithms and limitations of the algorithms helps chemical engineers and other users make informed decisions.

References:

1. Skagerberg, B.; Macgregor, J. F.; Kiparissides, C. Multivariate Data Analysis Applied to Low-Density Polyethylene Reactors. *Chemometrics and Intelligent Laboratory Systems* **1992**, *14* (1-3), 341–356.
2. Kiparissides, C.; Mavridis, H. Mathematical Modelling and Sensitivity Analysis of High Pressure Polyethylene Reactors. *Chemical Reactor Design and Technology* **1986**, 759–777.
3. Achilias, D.; Roupakias, C.; Megalokonomos, P.; Lappas, A.; Antonakou, E. Chemical Recycling of Plastic Wastes Made from Polyethylene (LDPE and HDPE) and Polypropylene (PP). *Journal of Hazardous Materials* **2007**, *149* (3), 536–542
4. Yuan, X.; Li, L.; Wang, Y.; Yang, C.; Gui, W. Deep Learning for Quality Prediction of Nonlinear Dynamic Processes with Variable Attention-Based Long Short-Term Memory Network. *The Canadian Journal of Chemical Engineering* **2019**, *98* (6), 1377–1389.
5. Szymańska, E. Modern Data Science for Analytical Chemical Data – A Comprehensive Review. *Analytica Chimica Acta* **2018**, *1028*, 1–10.
6. Silva, D. J. D.; Wiebeck, H. Predicting LDPE/HDPE Blend Composition by CARS-PLS Regression and Confocal Raman Spectroscopy. *Polímeros* **2019**, *29* (1).

7. Ge, Z.; Song, Z.; Ding, S. X.; Huang, B. Data Mining and Analytics in the Process Industry: The Role of Machine Learning. *IEEE Access* **2017**, *5*, 20590–20616.
8. Wu, K.; Sukumar, N.; Lanzillo, N. A.; Wang, C.; Ramprasad, R. "R.; Ma, R.; Baldwin, A. F.; Sotzing, G.; Breneman, C. Prediction of Polymer Properties Using Infinite Chain Descriptors (ICD) and Machine Learning: Toward Optimized Dielectric Polymeric Materials. Journal of Polymer Science Part B: Polymer Physics **2016**, 54 (20), 2082–2091.
9. Balcıoğlu, H. E.; Seçkin, A. Ç. Comparison of Machine Learning Methods and Finite Element Analysis on the Fracture Behavior of Polymer Composites. *Archive of Applied Mechanics* **2020**.
10. Rizkin, B. A.; Hartman, R. L. Supervised Machine Learning for Prediction of Zirconocene-Catalyzed α-Olefin Polymerization. *Chemical Engineering Science* **2019**, *210*, 115224.
11. Baughman, D. R.; Liu, Y. A. Neural Networks in Bioprocessing and Chemical Engineering; Academic Press: San Diego, CA, **1995**.
12. Jolliffe, I. T. *Principal component analysis*; Springer: New York, 2002.

Chapter 3. Data-driven Soft Sensor for Process Monitoring and Control of a HDPE plant

3.1 Introduction

Soft sensors are inferential estimators that alleviate the need for installation of expensive hardware sensors by making accurate predictions of relevant process variables. Because of a surge in data volumes in industries, there has been a realization of the potential of data-driven sensors. In order to ensure proper process control and instant on-line measurements, many chemical plants, and bioprocessing facilities have opted for virtual soft sensors instead of hard sensors. In this chapter, we look at building such a soft-sensor for a HDPE plant. Soft sensors are generally of two types: model-driven soft sensors and data-driven soft sensors. Since the foundation of data-based sensors is based on real processing conditions, data-driven sensors are more adaptive than model-driven sensors.

Recently, because of an increase in storage capacity of data, data-driven soft sensors have gained popularity in the chemical industry. Low sampling rates and difficulties faced during off-line measurements in a laboratory setting have also added to the necessity for developing these inferential sensors. There are many reported applications of data-driven sensors for process monitoring and fault detection. The strategy to develop a data-driven sensor depends on the type of process as well. Most data-driven sensors are most effective for continuous processes, while model-based sensors are applicable to both continuous and discontinuous processes. When applying data-driven sensors to discontinuous or batch sensors, we must consider one additional feature vector to characterize the batch-to-batch variance. Another issue to consider while building a data-driven sensor is missing data. Data-driven sensors struggle with missing data; one way to overcome the limitation is by imputation of the missing data.

For our case study, we build a data-driven sensor for a continuous HDPE plant from a reputed chemical industry in the Asian-Pacific region.[1] For a polyolefin process, polymer melt index (MI) serves as an important quality control variable, which Is also hard to measure on-line. MI refers to the ease of flow of the melt of a thermoplastic polymer.

3.2 Literature Review

Data-driven and model-based soft sensors are growingly important in bio-processing and chemical industries because of their cost-reduction potential and their use in process control.[2-5] Soft-sensors pave a way to perform real-time analyzing, monitoring, and control by providing reliant calculation of parameters, where no physical hard sensor is available. Gao et al. use ensemble deep kernel learning (EDKL) to predict melt index.[6] Polyolefin manufacturing uses such data-driven soft sensors to predict quality properties like the melt index and it has been a hot research topic among many chemical

engineers for quite some time.[7-10] Zhang et al. use empirical mode decomposition (EMD), relevance vector machine (RVM), and least squares support vector machine (LSSVM) to predict the melt index.

3.3 HDPE reactor setup

High-density ethylene (HDPE) production is an important slurry polymerization process as shown in Figure 3.1. The polymerization process involves two different reactors in parallel or in series and the entire process is highly exothermic. We obtain an industrial HDPE plant dataset from Professor Park.[1] From the raw process data, we use 14 input variables to predict the melt index as the output variable as shown in Table 3.1.

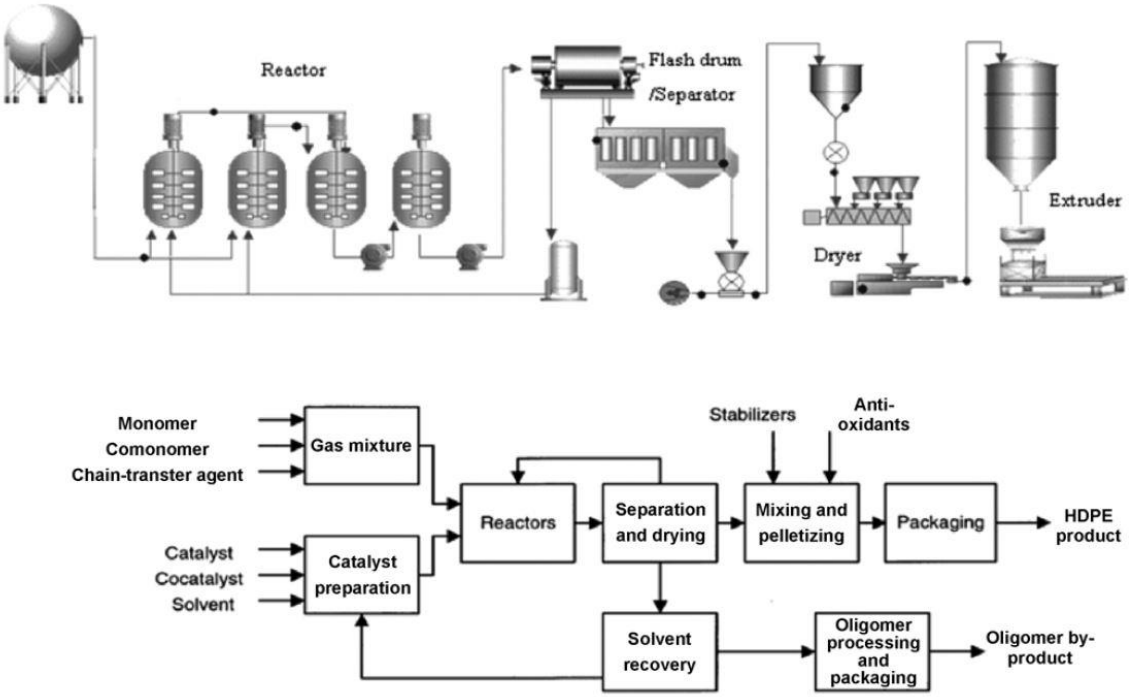


Figure 3.1. Schematic of the HDPE process used to build a soft sensor. Reproduced from reference (1)

Table 3.1. List of all the input variables for HDPE reactor

Input Variables	Symbol
Ethylene Feed Rate	ETH
Hexane Feed Rate	HDH
Recycled Hexane Feed Rate	HMH
Reactant with BUE ligand Feed Rate	PRL/BUE
Hydrogen Feed Rate	HYD
Reactor temperature (Bottom)	RT_BOTTOM
Reactor temperature (Middle)	RT_MIDDLE
Reactor temperature (Top)	RT_TOP
Reactor level	LEVEL
Reactor Pressure_1	RP1
Reactor Pressure_2	RP2
Agitator Speed	AGITATOR
Jacket Temperature_1	JT1
Jacket Temperaure_2	JT2

3.4 Algorithms and Model Building

We use three different models to predict the melt index using the 14 input variables. We split the dataset into training and testing sets (80-20 split). In order to preserve the dataset from further splits, we use cross-validation to divide the training set into 10 folds for training and validation, with 9 folds for training and the remaining one for validation. The algorithms we use are neural networks and deep neural networks, extreme gradient boosting (XGBoost), and support vector regression (SVR). Baughman and Liu show how to use neural networks for solving a variety process monitoring and control problems.[12] Similarly, SVR has found growing application in process control and monitoring [13] The use of XGBoost (extreme gradient boosting) machine learning algorithm in process applications appears to be relatively new.

3.4.1 Neural networks and deep neural networks

Neural Networks are a cluster of nodes, known as neurons, which are arranged into an ordered sequence of at least three groups known as layers. The first layer is called the input layer and has the same number of neurons as the number of input variables for the system, and the last layer is known as the output layer; the layers in-between are called hidden layers. The choice of number of layers and number of nodes is task-dependent and is mostly based on experience.[14] Deep neural networks are just neural networks with two or more than two hidden layers.

3.4.2 Extreme gradient boosting (XGBoost)

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. In gradient boosting, the algorithm predicts the residuals or errors of prior models and then adds them together to make the final prediction. It is called gradient boosting, because it uses a gradient descent algorithm to minimize the loss when adding new models. XGBoost is a powerful sequential ensemble technique because of its features such as: regularization for preventing overfitting, weighted quantile sketch for handling weighted data, and block structure for parallel learning for faster computing.[15]

3.4.3 Support vector regression (SVR)

SVR is a regression algorithm based on the principles of support vector machine (SVM), used to predict a continuous variable. Unlike many other algorithms, SVR tries to fit the best line within a predefined or threshold value instead of minimizing the error between predicted and actual values. It basically creates an error boundary space, which is a region separated by two parallel lines. The prediction lines which do not pass through the space are disregarded and the lines that pass become the support vectors for the model. SVR helps overcome the limitations associated with distributional properties of underlying variables, geometry of the data and the common problem of model overfitting. SVR algorithm uses a set of mathematical functions called kernel. The function of the kernel is to transform the input data in the right form. Kernels are of different types: linear, nonlinear, polynomial, radial basis function (RBF), and sigmoid. With the right choice of kernels, SVR can be a very powerful statistical tool.

3.4.4 Model discussion and results

We build two different neural network structures; one of the structure has one hidden layer, while the deep structure has two hidden layers. Figure 3.2 shows the structure for the deep neural network. As we can see from the structure, the deep neural network has 1 input layer with 14 neurons, 2 hidden layers with 10 neurons, and an output layer with 1 neuron. The simple neural network has the same

structure with one less hidden layer. We use rectified linear unit (ReLU) activation function, as the activation function for the hidden layers, which is the most commonly used activation function in deep learning models.[16] We use simple linear activation function for the output layer as we want a numerical regression output. We use 'Adam optimization' for training the neural net.[17] For SVR, we use the commonly used radial basis function (RBF) kernel.

Table 3.2 gives the results of the different algorithms. It is important to visualize how the predictions look compared to the original values. Figures 3.3-3.5 help us visualize the prediction error for the three algorithms along with the R2 score for each model. R2 scores characterize the fit of the model over the dataset. An R2 score close to one shows that most of the variance is explained by the data and indicates a good fit. We see that deep neural network outperforms the other two algorithms and is better than regular neural network. The RMSE values for D-NN are significantly lower (about four times) than SVM. We note that Park et. al[1] have also applied SVM to a melt index problem.

Table 3.2. Data-driven soft sensor model results for predicting melt index

Algorithm	**RMSE**
NN	1.0846
D-NN	0.9966
XGBoost	3.6163
SVM	4.0069
Std. Deviation	4.57

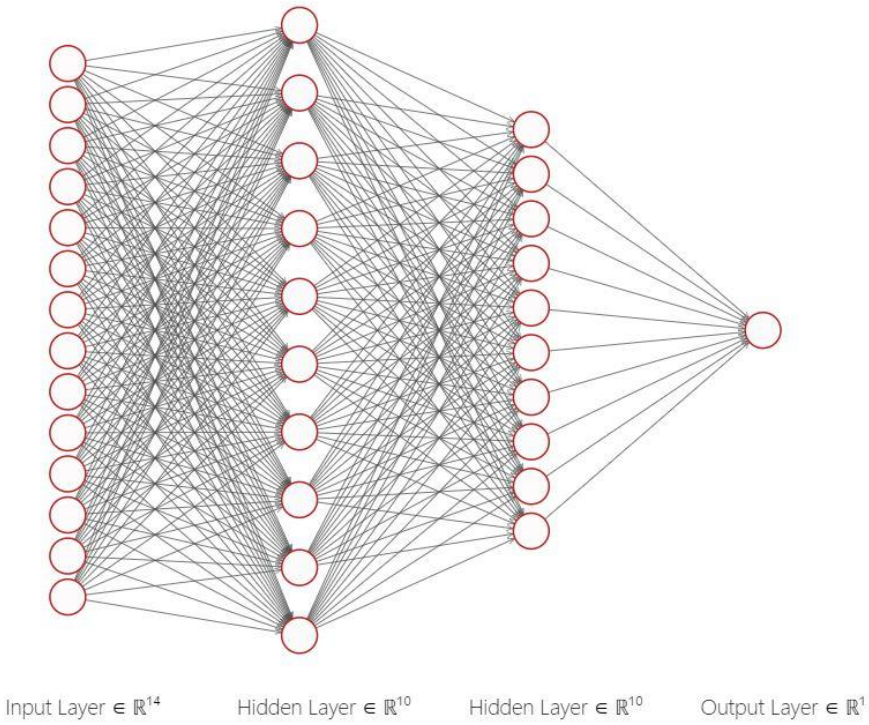


Figure 3.2. Basic deep neural network structure with two hidden layers.

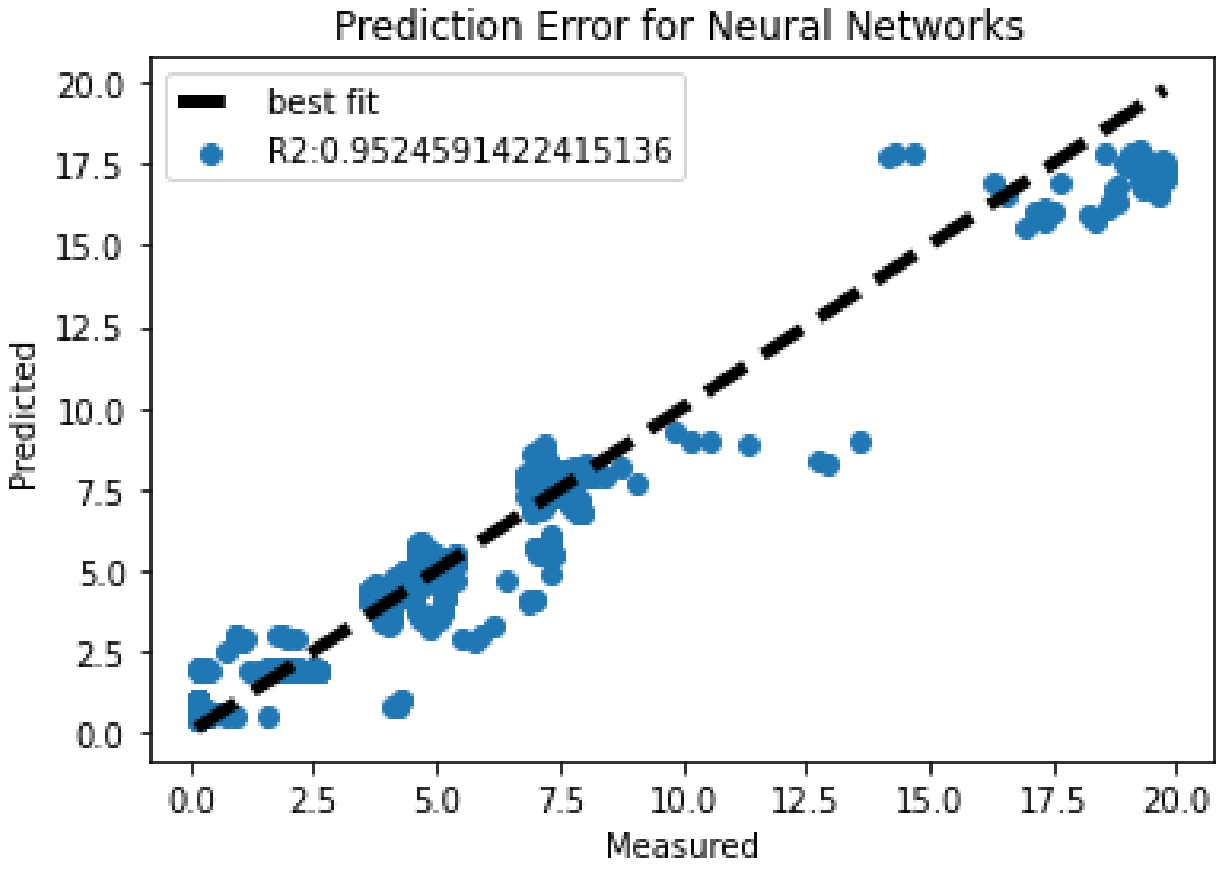


Figure 3.3 Prediction error plot for neural networks.

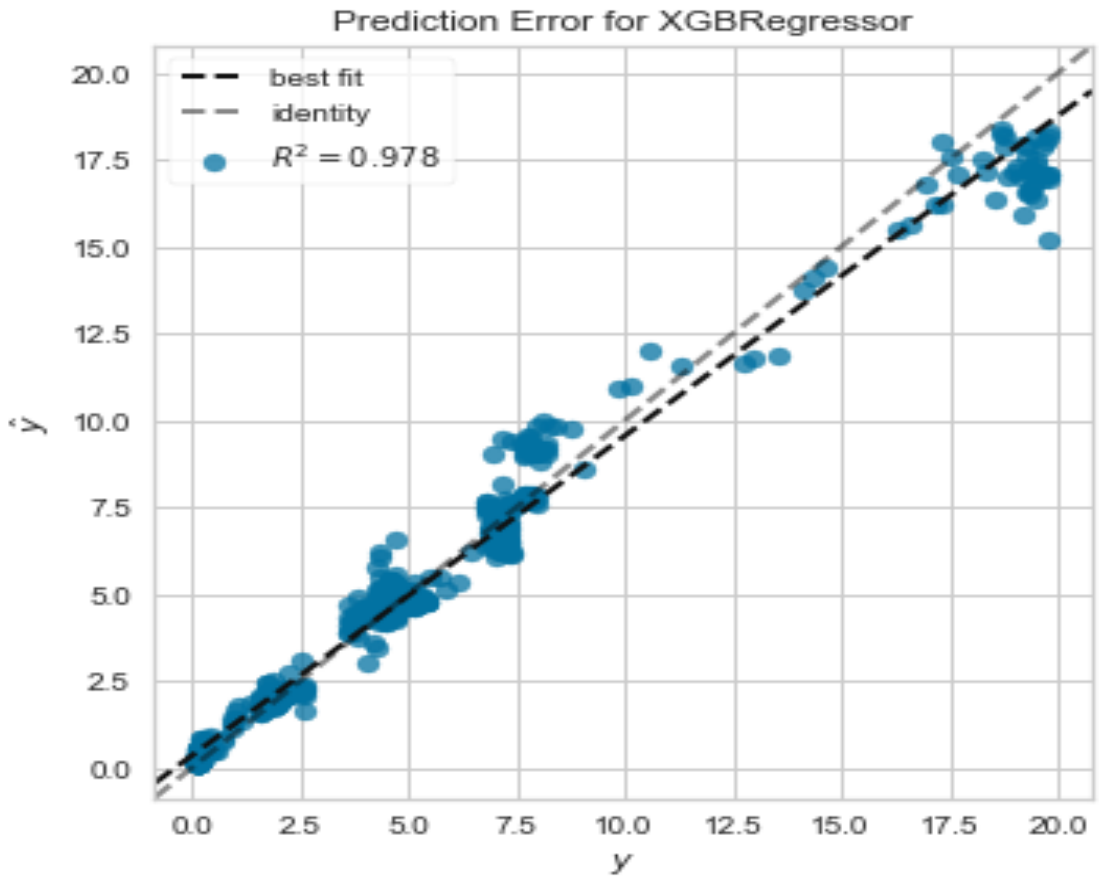


Figure 3.4. Prediction error plot for XGBoost regressor.

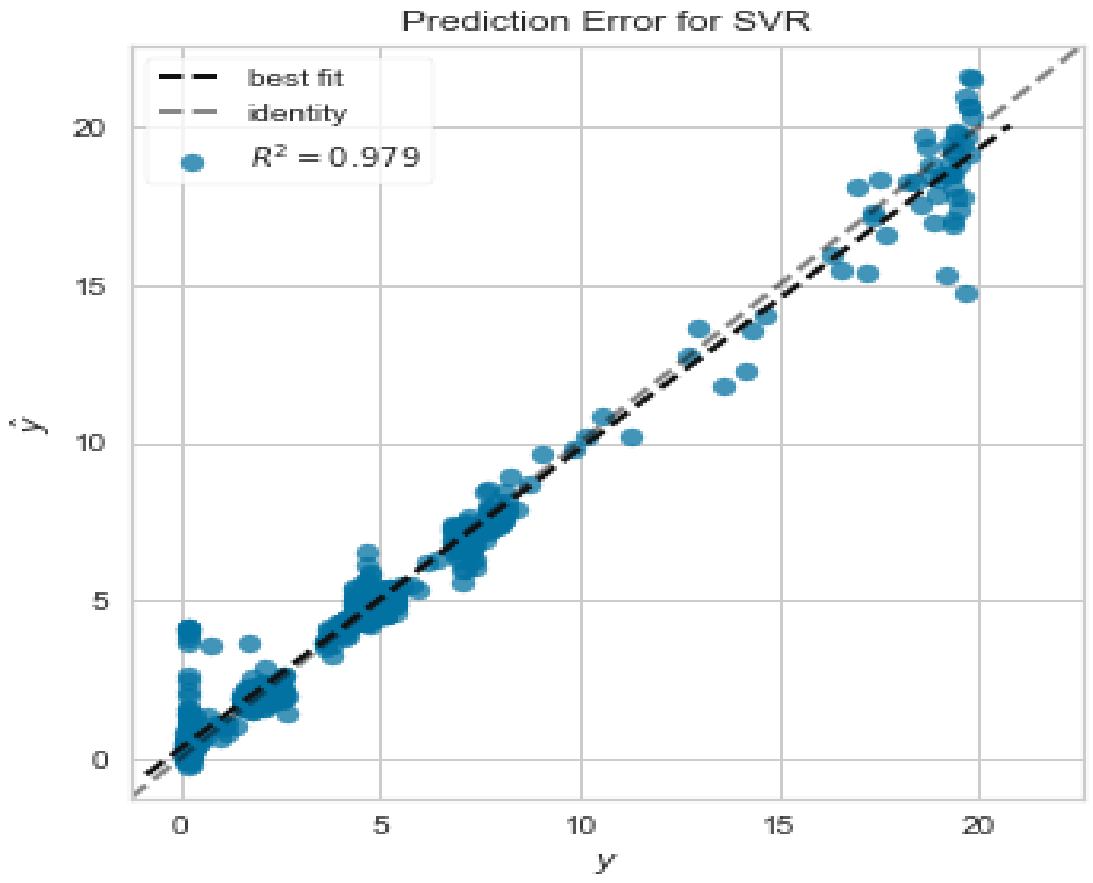


Figure 3.5. Prediction error plot for support vector regression (SVR).

3.5 Conclusion

In the polymerization process, for both LDPE and HDPE, machine learning algorithms can provide virtual assistance in many aspects of process monitoring and control. By providing accurate predictions, we can use soft-sensors to decrease capital investment costs and provide analysis and insights which otherwise may not be readily available. In this chapter, we build such a data-driven sensor and find that deep neural network provides us with the best predictions. With advances in computational technology and statistical algorithms, there are many avenues that await the adoption by chemical engineers to develop cost-effective soft sensors. Here, we showcase the benefits of one such algorithm to predict the melt index with high predictive power.

References

1. Park, T. C.; Kim, T. Y.; Yeo, Y. K. Prediction of the Melt Flow Index Using Partial Least Squares and Support Vector Regression in High-Density Polyethylene (HDPE) Process. *Korean Journal of Chemical Engineering* **2010**, *27* (6), 1662–1668.
2. Kaneko, H.; Arakawa, M.; Funatsu, K. Novel Soft Sensor Method for Detecting Completion of Transition in Industrial Polymer Processes. *Computers & Chemical Engineering* **2011**, *35* (6), 1135–1142.

3. Cheng, Z.; Liu, X. Optimal Online Soft Sensor for Product Quality Monitoring in Propylene Polymerization Process. *Neurocomputing* **2015**, *149*, 1216–1224.
4. Liu, Y.; Chen, J. Integrated Soft Sensor Using Just-in-Time Support Vector Regression and Probabilistic Analysis for Quality Prediction of Multi-Grade Processes. *Journal of Process Control* **2013**, *23* (6), 793–804.
5. Jiang, X.-L.; Guo, X.-Q. Soft-Sensor Modeling of Quality Control Based on Support Vector Machine. *Journal of Computer Applications* **2009**, *28* (9), 2382–2385.
6. Liu, Y.; Yang, C.; Gao, Z.; Yao, Y. Ensemble Deep Kernel Learning with Application to Quality Prediction in Industrial Polymerization Processes. Chemometrics and Intelligent Laboratory Systems **2018**, 174, 15–21.
7. Han, I.-S.; Han, C.; Chung, C.-B. Melt Index Modeling with Support Vector Machines, Partial Least Squares, and Artificial Neural Networks. *Journal of Applied Polymer Science* **2004**, *95* (4), 967–974.
8. Liu, Y.; Gao, Z. Industrial Melt Index Prediction with the Ensemble Anti-Outlier Just-in-Time Gaussian Process Regression Modeling Method. *Journal of Applied Polymer Science* **2015**, *132* (22).
9. Liu, Y.; Yang, C.; Liu, K.; Chen, B.; Yao, Y. Domain Adaptation Transfer Learning Soft Sensor for Product Quality Prediction. Chemometrics and Intelligent Laboratory Systems **2019**, 192, 103813.
10. Zhang, M.; Liu, X.; Zhang, Z. A Soft Sensor for Industrial Melt Index Prediction Based on Evolutionary Extreme Learning Machine. Chinese Journal of Chemical Engineering 2016, 24 (8), 1013–1019.
11. Zhang, M.; Zhou, L.; Jie, J.; Liu, X. A Multi-Scale Prediction Model Based on Empirical Mode Decomposition and Chaos Theory for Industrial Melt Index Prediction. Chemometrics and Intelligent Laboratory Systems 2019, 186, 23–32.
12. Baughman, D. R.; Liu, Y. A. Neural Networks in Bioprocessing and Chemical Engineering, **1995**, Elsevier, Atlanta, GA.
13. Kecman, V. (2005). Support Vector Machines – An Introduction. Support Vector Machines: Theory and Applications Studies in Fuzziness and Soft Computing, 1-47. doi:10.1007/10984697_1
14. Bramer, M. (2020). An Introduction to Neural Networks. In Principles of Data Mining (pp. 427-466). Springer Verlag London.
15. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD 16, San Francisco, CA, August, 2016

Chapter 4. Ensemble-Based Machine Learning for Industrial Fermenter Classification and Foaming Control

Aman Agarwal and Y.A. Liu
AspenTech-PetroChina Center of Excellence in Process System Engineering,
Department of Chemical Engineering, Virginia Polytechnic Institute and State University,
Blacksburg, Virginia 24061, United States

Christopher McDowell
Novozymes Biologicals Inc., 5400 Corporate Circle, Salem, Virginia 24153, United States

4.1 Introduction

4.1.1 Foaming control and dataset collection

The process of foaming refers to the formation of a dispersed media by trapping pockets of gas in a network of thin liquid films or solids. A foam generation process can be further described by two distinct events: air entrapment and bubble breakup.[1] Despite its simplicity, the foaming process affects the industrial sector in a substantial way due to its dynamic nature, making it a major technological hurdle. [2-9] In a bioprocess, due to the extensive aeration and presence of active microorganisms that reduce the surface tension, foams can build up to a serious level resulting in several issues such as microbial cell stripping and contamination.[10]

Some of the common methods of controlling foaming involves addition of chemical antifoam agents (AFA) to prevent the adverse effects of foaming. Increased usage of industrial AFA tends to decrease cell viability and the effects are intensified with increased exposure and higher concentration of AFA.[11] The current methods of AFA addition are based on empirical methods or operational experiences. This study aims at mitigating the adverse effects of excessive AFA addition by using advanced machine learning algorithms to predict the onset of foaming. Conventionally, we estimate the amount of foaming by using the empirical foaming parameters: foamability (maximum height reached by the foam after CO2 injection), Bickerman coefficient (bubble average lifetime), and surface tension of the liquid.[12] However, this estimation is time-consuming and not viable for dynamic process control.

This paper presents a way to estimate foaming based on the prediction of differential pressure using ensemble-based machine learning algorithms. As illustrated in Figure 4.1, the exhaust differential pressure (DP1) is the pressure difference between the fermenter head pressure (PT01), measured from the headspace of the fermenter, and the exhaust pressure (PT02), measured from the exhaust line downstream of the fermenter.

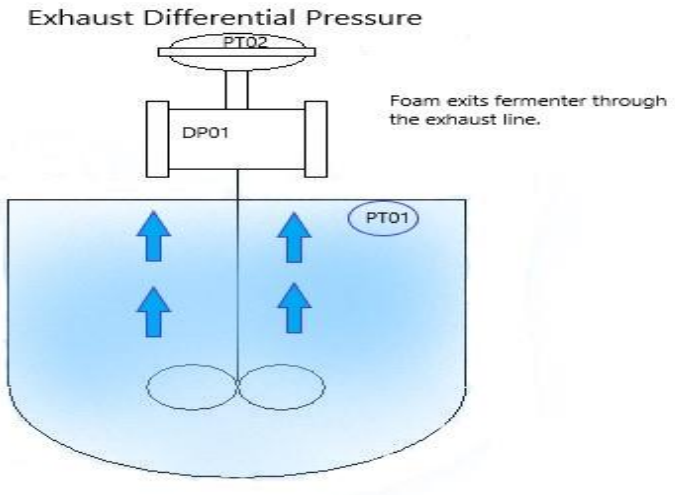


Figure 4.1. Fermenter design with pressure notations for exhaust differential pressure calculations.

Fermentation practices in industry shows that the exhaust differential pressure can be a strong indicator of foaming. Developing a model which can accurately predict foaming would require large industrial dataset with operating variables over a wide range of values to accommodate different fermenter designs and multiple strains of microorganisms. No such model is currently available, because of lack of relevant industrial data and application of conventional methods to predict foaming.

Physical modeling of foaming is an arduous process as it requires estimation of foam height which is dynamic in nature and differs for different bioprocesses.[13] Machine-learning based modeling helps mitigate the necessity for foam height estimation, and it can be generalized for any process as it uses the available operational data for prediction. Recently, machine-learning based methods have found several applications in sectors where mechanistic modeling is precluded by the inability to develop a model or generalize it for a process.[14-18]

Figure 4.2 shows a schematic diagram of an experimental setup to study the dynamic measurement of foam behavior for a continuous fermenter over a range of key process variables[19]. This setup and the additional details in reference [19] suggest that the key measurement considerations in foaming control

include: probes (pH, temperature and dissolved oxygen); additions of antifoam, media (air), acid and caustic for pH control; volume of fermenter, agitator speed, and hour (time of foaming). In developing the dataset for the current study, we collect the key plant data following reference [19], and include an identical list of 11 independent variables. Figures 4.3 to 4.5 display these 11 independent variables (X), together with our quality variable (Y), the exhaust differential pressure. In particular, our dataset consists of 64 batches with four fermenter designs, with 11 independent variables (X) and one quality variable (Y), totaling over 183,000 instances. These data require cleaning, integration and transformation before being used for model building.

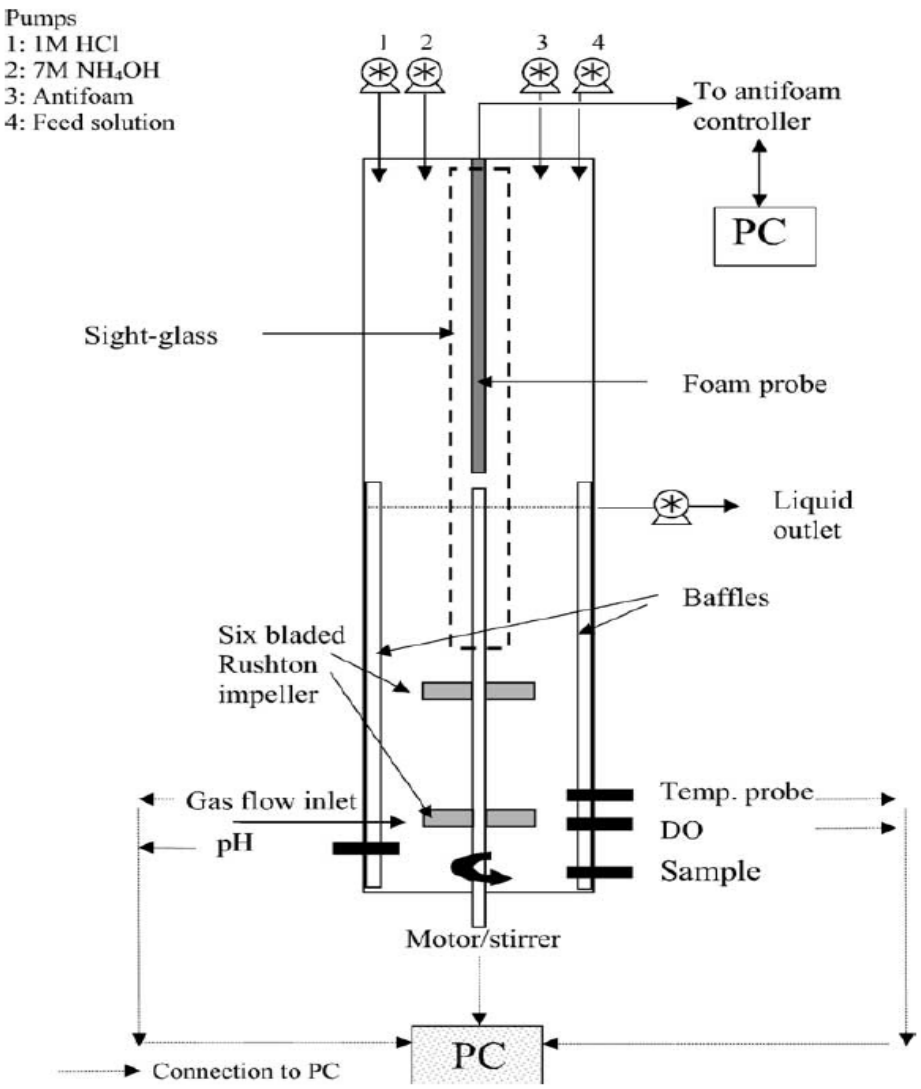


Figure 4.2. Fermenter setup for dynamic study of foaming behavior [19]

4.1.2. Ensemble methods for modeling foaming

The proposed model is based on ensemble methods, which are meta-algorithms that offer a systematic solution by combining the predictive power of several learners. Ensemble methods provide an extra degree of freedom in the classical bias-variance tradeoff, where model complexity is carefully calibrated so that the fit on the training sample reflects performance out-of-sample, and have allowed for solutions to many complicated problems.[20-25] The book by Zhou[26] gives an excellent introduction to the foundations and algorithms of ensemble methods. Bootstrap aggregating (bagging) and boosting are two popular ensemble methods which can be used with several statistical models, predominantly with decision trees. In this paper, we use two popular ensemble methods: 1) extreme gradient boosting (XGBoost), an efficient boosting sequential ensemble framework designed by Chen et al [27] and 2) random forest (RF), an established parallel bagging ensemble framework developed by Brieman et al.[28]

4.2 Ensemble Framework and Methodology

The four steps in our proposed scheme for the ensemble-based foaming prediction model are: data acquisition, data preprocessing and visualization, ensemble-based model building, and foaming prediction. For the data acquisition step, we acquire real-industrial annual batch data for different fermenters from a fermentation plant. For consistency, we collect the data for a specific strain of bacteria over time in different fermentation setups. The data preprocessing involves data cleaning, integration, and transformation. In the data visualization step, we justify the usage of ensemble methods with the help of heat-mapping based on linear correlations. Heat-mapping helps us visualize the existing bivariate correlations between independent variables and a chosen dependent variable. We then apply the ensemble methods for classification of different types of fermenters, and for both classification and regression models for foaming prediction. We quantify the model results based on different averaging techniques and metrics for our multiclass classification models and root mean squared error (RMSE) for regression models. We also perform model validation using 10-fold cross validation, which is explained ahead in Section 4.2.2.

4.2.1 Data preprocessing

Data preprocessing include cleaning, integration and transformation. Data cleaning identifies irrelevant and inaccurate records, and removes noisy and redundant data, which may occur because of usage of multiple temperature and pressure sensors, multiple dissolved oxygen (DO) probes, multiple identifiers for the same independent variables, and different fermenter design setups. The raw data consist of several columns with redundant data due to presence of backup probes. We remove all the

columns with identical data for temperature, pressure, dissolved oxygen, etc. Similarly, we remove some data columns, which play no role in foaming, like inoculation date, speculative batch count, etc. In order to ensure the robustness of the prediction model for exhaust differential pressure, we remove the directly correlated variables like fermenter head pressure and exhaust line pressure, which are also measured along with exhaust differential pressure.

Data integration involves combining datasets from different fermenters to form a generalized dataset with over 183,000 instances. Data transformation involves standardization of the data to center the data (shift it to have a zero mean and remove bias). It also scales the data so the values are in a standardized unit and the data have unit variance (so variables with large values do not dominate the analysis results).[29] The transformation also involves changing the cumulative values of some independent variables (such as acidic flow, caustic flow, antifoam flow, and volume of the reactor) to noncumulative values.

4.2.2 Data visualization and model selection

In order to find the real underlying sources of variation and select the right features, we use the multivariate data analysis software Aspen ProMV (available through university programs of Aspen Technology Inc., Bedford, MA) to rank the relative importance of 11 independent variables. The software generates the variable importance plot of Figure 4.3 based on partial least square (PLS) to determine the important features for model building. The y-axis represents the Variable Importance in Projection (VIP) score, computed for a selected variable in multiple principal components following an equation given by Wold et al.[30] This analysis considers an independent variable as important when its VIP score is close to or greater than one in a given model. As expected, hour (time of foaming), pH, dissolved oxygen, and fermenter temperature are the most important variables for the model.

We use Python 3.6 and its various packages (e.g., Scikit-learn machine learning python library) for the following visualizations and the proposed models.[31-34] Before model building, we visualize the data using bar graphs for each variable as shown in Figure 4.4 for the selected variables.

By visualizing the data, we can see the spread of each of the independent variables and make a preliminary feature selection, where similar or non-variant attributes are removed. For some variables with multiple values, like pH, it is important to see if the values are reasonable. In an industrial fermentation setup, it is very common to have backup probes which share identical values with the active probes, or are inactive during regular operations. We check the entire dataset to identify such similar or non-variant variables, so they can be removed before data processing.

After visualizing the spread of the data and feature selection, it is important to see if an independent variable and our quality variable (foaming indicator; exhaust differential pressure) are linearly correlated. A good way to visualize any such correlation is to create a correlation heat map. Figure 4.5 shows the correlation of all 11 independent variables with the exhaust differential pressure. Because of restriction of industrial plant data, we represent the y-axis of the figure in max-min normalized log scale for the variable. We choose hour (time of foaming) as an independent variable, as none of the other independent variables is linearly correlated with time and the fermenter data are collected for multiple batches in the same period. In cases, where time linearly changes with other variables or if data are collected in different periods for multiple batches, we should remove hour as this independent variable becomes redundant or an observational ID.

We see a positive correlation of the exhaust differential pressure with hour, pH, fermenter temperature, and agitation speed. Similarly, we see a negative correlation against dissolved oxygen. This observation aligns with our understanding of foaming[19]. This analysis shows that we can apply visualization tools such as annotated heat mapping for quick analysis of large fermentation datasets to identify patterns among several batches with possible multiple strains of microorganisms and different fermenter designs. We also see that none of the independent variables shares a high (>|0.5|) positive or negative linear correlation with the dependent variable. Thus, our reasoning for using ensemble methods, known for handling data with nonlinear correlations, instead of simple linear models, is justified.[21]

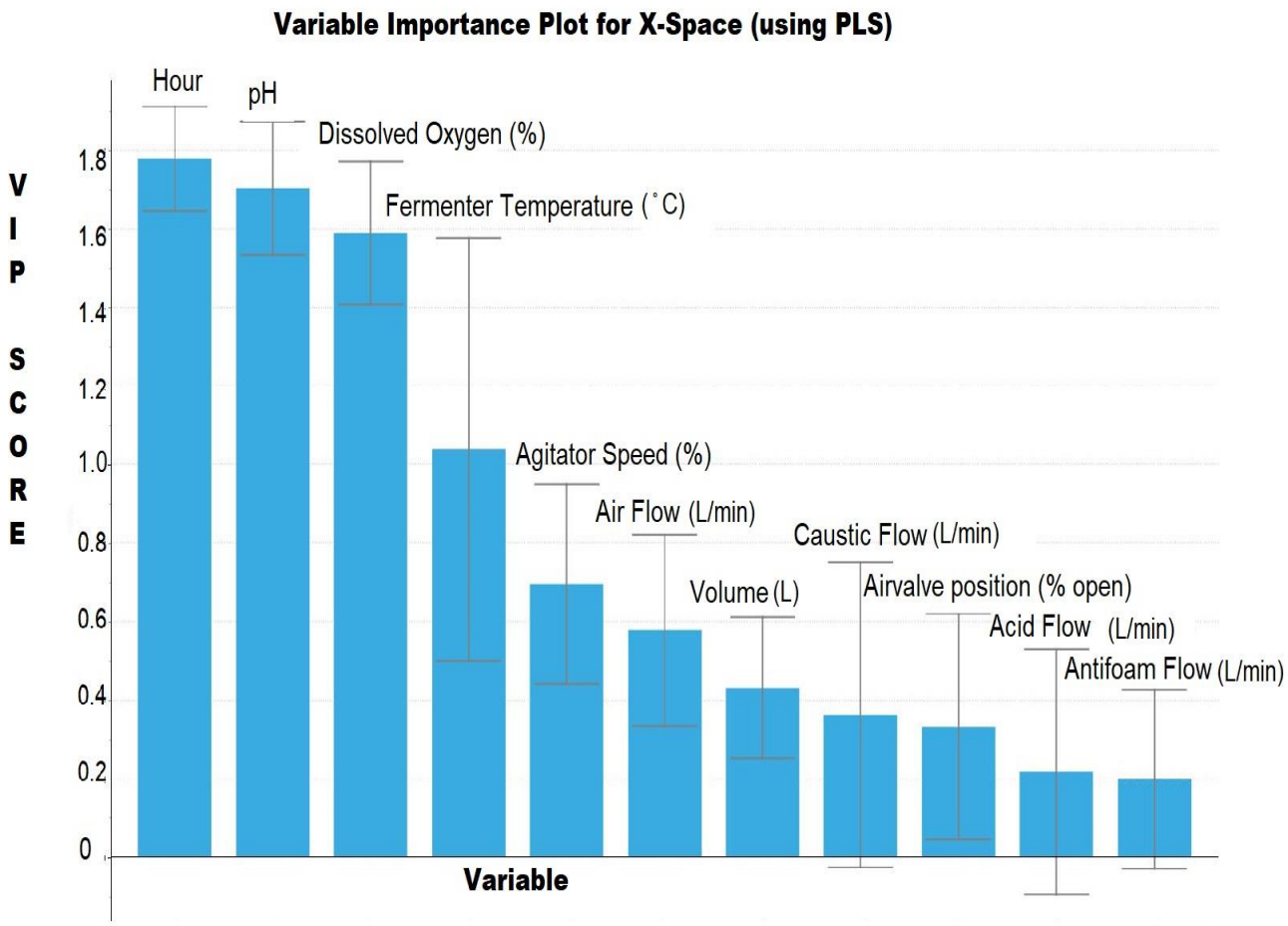


Figure 4.3. Variable importance plot based on partial least square (PLS) created using Aspen ProMV.

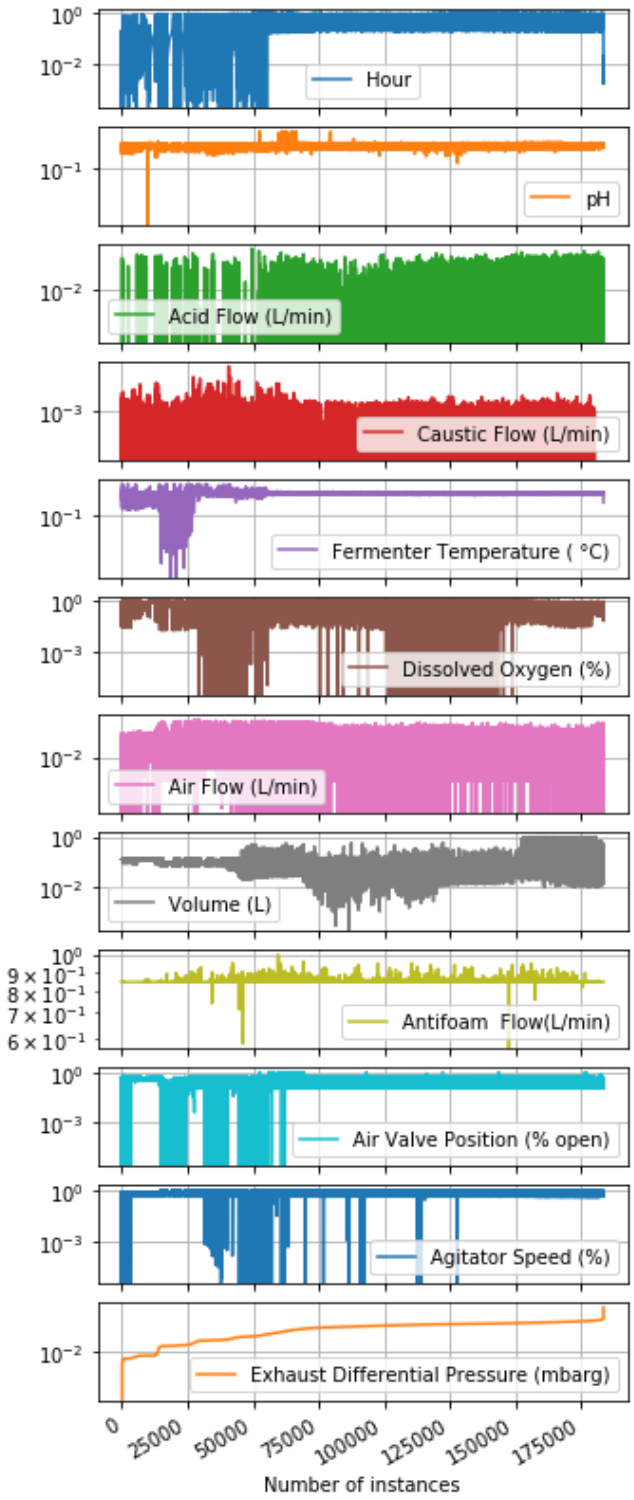


Figure 4.4. Bar graph representation of selected features for the entire dataset (about 183,000 instances); the y-axis is in max-min normalized log scale for the variable.

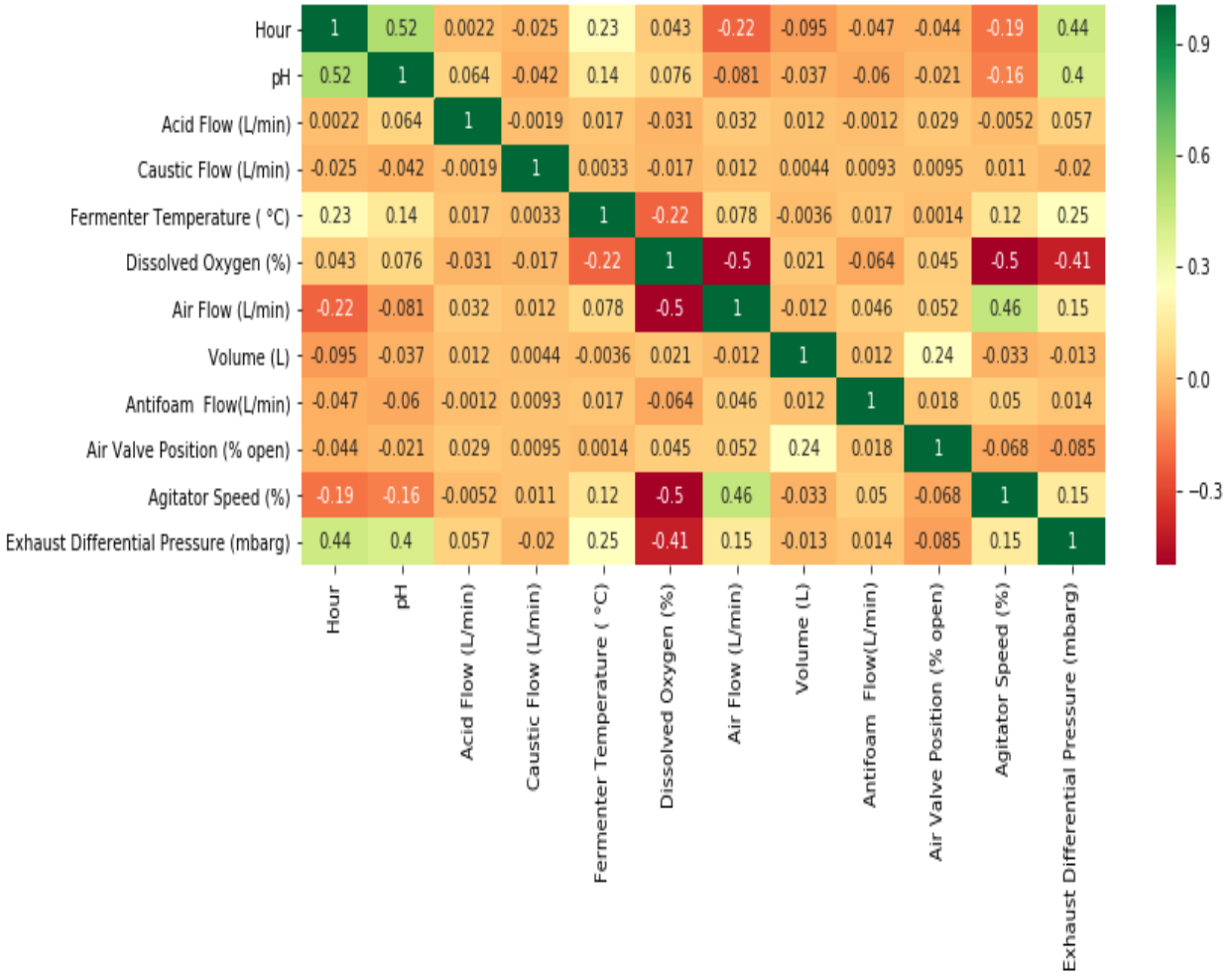


Figure 4.5. Linear correlation heat mapping to identify positive and negative relationships.

4.3 Model Building and Model Evaluation

The process of fermentation differs from other industrial processes as the design of fermenter and its mode of operation depends highly on the choice of microorganism used for fermentation.[35] Our fermentation plant currently has four fermenter types (designated types A, B, C and D), each with different material, geometric and equipment specifications. Some of these specifications are: straight wall height, diameter, aspect ratio, total volume, fill volume, stainless steel type, jacketing, aeration method, agitator mounting position, number of impellers, impeller diameter and impeller type, etc.

In particular, while these specifications may contribute to generating different fermentation performance, we wish to investigate if we could develop an ensemble-based fermenter classification model that: (1) evaluates all the datasets for the four fermenter types together in an integrated fashion;

(2) considers the 11 independent variables for each batch alone without having to incorporate explicitly the design specifications for the specific fermenter type for the batch; and (3) can correctly differentiate or classify the fermenter type with an accuracy of over 99% for our integrated dataset of over 183,000 instances. If this development of fermenter classification model is successful, it can help to generalize the foaming prediction for different types of fermenters and integrate the multiple datasets for a better model. We discuss the development of a XGBoost-based [27] fermenter classification model in Section 4.4.

After developing a model for fermenter type prediction, we build classification and regression models for foaming prediction based on the exhaust differential pressure prediction. For the classification model, we bin the exhaust differential pressure into four thresholds: lower threshold limit, close to threshold limit, threshold limit, and upper threshold limit, which is explained in Section 4.4.2. We use XGBoost-based [27] and RF-based [28] ensemble models for comparison. Finally, we build regression models using both of the meta-algorithms to predict the exact value of exhaust differential pressure in mbarg.

The model evaluation step involves partitioning the entire dataset into a training set and a test set to determine the predictive power of each model. For our models, we use 80-20 training-test split for model building. The training set is useful in the formulation of the model, while the test set is independent and plays no part in the process. We use the test set for validating the model predictions, and reserve one-fifth of the training set for hyperparameter tuning (validation set). This ensures an unbiased evaluation of model fit on training set, while tuning the hyperparameters. The k-fold cross-validation technique is a well-established and flexible technique for selecting the ideal model (which avoids overfitting and over-optimization).[36] It involves partitioning the data into k disjoint subsets and using each one for validation and the remainder for training. The k parameter is adjustable according to performance and predictive evaluation considerations. For our models we use 10-fold cross-validation, which involves randomly dividing the dataset into 10 groups, or folds, of approximately equal size. The first fold is treated as a test set, and the method is fit on the remaining 9 folds.

For our regression model, we use root mean square error (RMSE), which is the square root of the average of squared errors. It is measured in the same unit as the target variable, which in our case is mbarg (for exhaust differential pressure). Similarly, we use four different evaluation metrics for our classification models: accuracy, precision, recall, and f1-score. The accuracy of a classification model can be defined as the ratio of the total number of correct predictions by the total number of predictions. Precision is the ability of a classification model to return only relevant instances. Recall refers to the

ability of a classification model to identify all relevant instances. In addition, f1-score is the single metric that combines recall and precision using the harmonic mean. [29]

The evaluation metrics (precision, recall, and f1-score) used for multiclass classification are calculated based on different ways to average binary metric calculations for different class labels. The three common ways of averaging are: micro-averaging, macro-averaging, and weighted-averaging. Micro-averaging gives each class an equal contribution to the overall metric. Macro-averaging uses the mean of the binary metrics, giving equal weight to each class. Finally, weighted-averaging accounts for class imbalance by incorporating the average of binary metrics in which each class's score is weighted by its presence in the data sample. [31] Table 4.1 shows how these measures can be calculated by these different averaging techniques. [31] If all the classes are included, micro-averaging gives same values for precision, recall, and f1-score, which are identical to the accuracy score for the model.

Table 4.1. Calculation of evaluation metrics using different averaging techniques for multiclass classification models. [31]

Averaging Technique	Precision (P)	Recall (R)	F1-score
Micro	$P(y,\hat{y}) \coloneqq \frac{\lvert y \cap \hat{y}\rvert}{\lvert y\rvert}$	$R(y,\hat{y}) \coloneqq \frac{\lvert y \cap \hat{y}\rvert}{\lvert \hat{y}\rvert}$	$F(y,\hat{y}) \coloneqq 2\left[\frac{P(y,\hat{y}) \times R(y,\hat{y})}{P(y,\hat{y}) + R(y,\hat{y})}\right]$
Macro	$\frac{1}{\lvert C\rvert}\sum_{c\in C} P(y_l,\hat{y}_l)$	$\frac{1}{\lvert C\rvert}\sum_{c\in C} R(y_l,\hat{y}_l)$	$\frac{1}{\lvert C\rvert}\sum_{c\in C} F(y_l,\hat{y}_l)$
Weighted	$\frac{1}{\sum_{c\in C}\lvert\hat{y}_l\rvert}\sum_{c\in C}\lvert\hat{y}_l\rvert P(y_l,\hat{y}_l)$	$\frac{1}{\sum_{c\in C}\lvert\hat{y}_l\rvert}\sum_{c\in C}\lvert\hat{y}_l\rvert R(y_l,\hat{y}_l)$	$\frac{1}{\sum_{c\in C}\lvert\hat{y}_l\rvert}\sum_{c\in C}\lvert\hat{y}_l\rvert F(y_l,\hat{y}_l)$

4.3.1 Algorithms

4.3.1.1 Extreme gradient boosting (XGBoost)

Extreme gradient boosting is an implementation of gradient boosting decision trees, which is a powerful sequential ensemble technique because of its features such as: regularization for preventing

overfitting, weighted quantile sketch for handling weighted data, and block structure for parallel learning for faster computing.[26]

Let f_k be the prediction from a decision tree and suppose we have K trees; the model is a collection of trees given by:

$$\text{Model } = \sum_{k=1}^{K} f_k \quad (1)$$

After collecting all decision trees, we make prediction at the t-th step by

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k\,(x_i) \quad (2)$$

where x_i is the feature vector for the t-th data point.

For training the model, we need to optimize a loss function (L) and add a regularization term (Ω) to form a training objective function (*Obj*):

$$Obj = L + \Omega \quad (3)$$

For an iterative algorithm, we redefine the objective function as

$$Obj^{(t)} = \sum_{i=1}^{N} L(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega\,(f_i) = \sum_{i=1}^{N} L(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \sum_{i=1}^{t} \Omega\,(f_i) \quad (4)$$

To optimize with gradient descent, we need to calculate the gradient; in order to achieve high performance, we consider both first-order and second-order gradients. Since we do not have the derivative of every objective function, we calculate its second-order Taylor approximation and remove constant terms. This leads to:

$$Obj^{(t)} = \sum_{i=1}^{N} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \quad (5)$$

where, $g_i = \partial_{\hat{y}^{(t-1)}} l\left(y_i, \hat{y}^{(t-1)}\right)$, $h_i = \partial^2{}_{\hat{y}^{(t-1)}} l\left(y_i, \hat{y}^{(t-1)}\right)$

Now we define a tree:

$$f_t(x) = w_{q(x)} \quad (6)$$

where $q_{(x)}$ is a directing function that assigns every data point to the q(x)-th leaf

We define the index set as:

$$I_j = \{i | q(x_i) = j\} \quad (7)$$

Then, the objective function after expanding the regularization term and indexing becomes:

$$Obj^{(t)} = \sum_{j=1}^{T}[\sum_{i\in I_j} g_i w_j + \frac{1}{2}(\sum_{i\in I_j} h_i + \lambda)w_j^2] + \gamma T \quad (8)$$

After substituting the best w_j to optimize the objective function, we get:

$$Obj^{(t)} = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i+\lambda} + \gamma T \quad (9)$$

Let $I_L$and I_R be the sets of indices assigned to two new leaves. Then, we can write

$$Gain = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i+\lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i+\lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i+\lambda}\right] - \gamma \quad (10)$$

where gamma (γ) is a pseudo-regularization hyperparameter (Lagrange multiplier) used for pruning (reducing the size) the tree. Ultimately, the algorithm chooses the final structure by selecting splits with maximized gain.

4.3.1.2 Random forest

Random forest is a parallel ensemble technique which uses both bootstrap aggregation (bagging) and random variable selection for tree building. To obtain low-bias trees, each tree is unpruned (grown fully); in the meantime, bagging and random variable selection ensure low correlation between individual trees. The method yields an ensemble of unstable individual learners, which together can achieve both low bias and low variance. Some of the features of random forest which makes it a prominent method are high tolerance for multidimensional data, good performance for multiclass classification problems, handles overfitting internally, and works well with noisy data.[26]

Let us assume an ensemble of E trees $\{T_1(X), \dots, T_E(X)\}$, where $X = \{x_1, \dots, x_n\}$ is a n-dimensional vector of independent variables associated with a dependent variable. The ensemble method produces E outputs $\{\hat{Y}_1 = T_1(X), \dots, \hat{Y}_E = T_E(X)\}$, where $\hat{Y}_e, (e = 1, \dots, E)$ is the prediction for a dependent variable by the e-th tree. The outputs from all trees are aggregated to produce a final prediction, $\hat{Y}$. For the classification models, $\hat{Y}$ is the class predicted by the majority of the trees; while for regression models, it is the average of the individual tree predictions.

Given a dataset of n instances for training, $D = \{(X_1, Y_1), \dots, (X_n, Y_n)\}$, where $X_i, (i = 1, \dots, n)$ is a vector of independent variables and Y_i is the dependent variable of interest. From the training data of n instances, the algorithm chooses a random sample with replacement. For each bootstrap sample, a tree

is grown with the following modification: at each node, the best split is chosen among a randomly selected subset of independent variables using GINI index, defined by

$$Gini(S) = 1 - \sum_i p_i^2 \quad (11)$$

where p_i is the probability of an item with label *i* in a set S belongs to a class C

The tree is grown to the maximum size and not pruned back. The steps are repeated until E such trees are grown. The standard tree growing algorithm used in the RF method is based on classification and regression trees (CART) model.

4.4 Model Results and Discussion

We develop three different models using ensemble methods for fermenter-batch prediction and foaming prediction. These include: (1) fermenter type classification model; (2) foaming prediction classification model; and (3) foaming prediction regression model.

4.4.1 Fermenter type classification model

Four different fermenter batch types are differentiated using a classification ensemble model based on XGBoost algorithm. Figure 4.6 shows a confusion matrix of the results produced by the fermenter classification model, which is able to differentiate the batch type with 99.49% accuracy. The diagonal in the confusion matrix represents the number of correct predictions. Table 4.2. shows the evaluation metrics obtained by micro-averaging, macro-averaging, and weighted-averaging. For fermenter batch prediction, we get identical results for all three different types of averaging techniques.

We thereby infer that, the operating variables do incorporate all the necessary design differences and our ensemble approach can successfully detect these hidden patterns. By using the operating variables as input and no prior knowledge about the fermenter design (number of impellers, aspect ratio, etc.) and the microorganism, the model is able to distinguish one batch type from another based on the operating conditions alone. Separation of batch types is very essential for data integration and generalization of any model which will be used for foaming prediction. These results demonstrate the potential of ensemble methods in aiding big data analytics by allowing for data integration from multiple fermenters and multiple organisms.

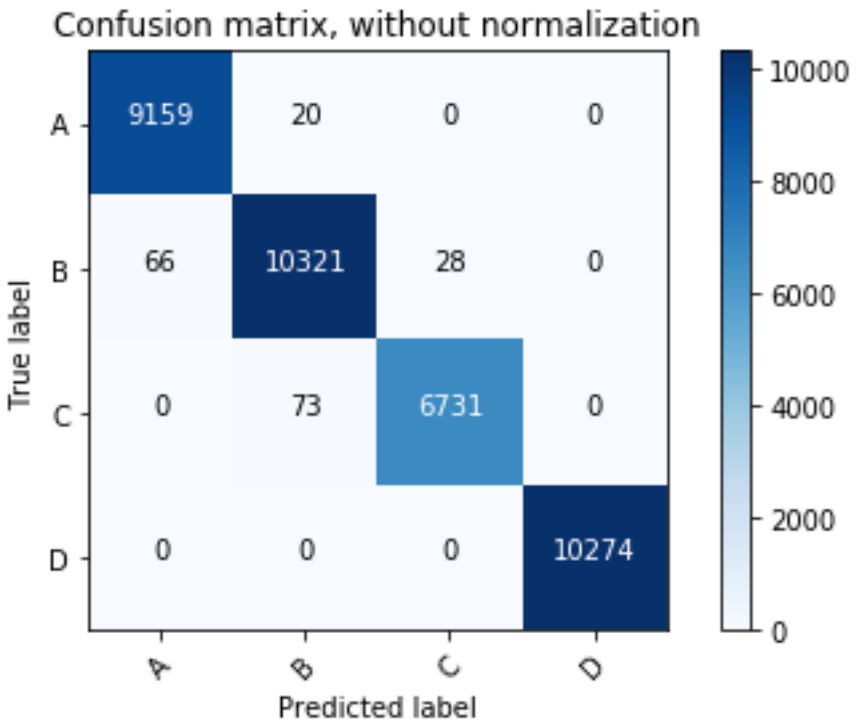


Figure 4.6. Confusion matrix for XGBoost-based fermenter classification model for 4 different fermenter types (A, B, C, and D).

Table 4.2. Fermenter batch classification model evaluation summary using XGBoost and different averaging techniques for metric calculation.

Averaging Technique	**Accuracy (in %)**	**Precision (P)**	**Recall (R)**	**F1-score**
Micro	99.49	0.99	0.99	0.99
Macro	99.49	0.99	0.99	0.99
Weighted	99.49	0.99	0.99	0.99

4.4.2 Foaming prediction classification model

Now that we have successfully separated the batch types, we integrate the datasets for a larger training dataset for the foaming prediction model. The foaming prediction model is based on the prediction of the exhaust differential pressure, which serves as an indicator of foaming and is observed *when the differential pressure is above 100 mbarg* based on past industrial fermentation experience. For the classification models, the numeric pressure values are binned to different thresholds so the model can predict a particular class type (threshold). The idea behind binning is to see how successful the models will be at predicting a certain threshold to allow for alleviatory steps such as addition of AFA. The threshold limits (arbitrarily set) are as follows:
L: Lower Threshold (below 90 mbarg), C: Close to Threshold (between 90-100 mbarg), T: Threshold (at 100 mbarg), and U: Upper Threshold (above 100 mbarg).

Figure 4.7 and Figure 4.8 show the respective confusion matrices for the foaming classification model based on XGBoost and RF, respectively. Like before, the diagonal in the confusion matrix represents the number of correct predictions. From Table 4.3. and Table 4.4., we can see that for foaming prediction micro-averaging results in better evaluation scores than macro-averaging. This makes sense as micro-averaging is a measure of effectiveness on larger classes in a test set. While, macro-averaging is a measure of effectiveness on the overall test set. By taking the weighted-average, we see that the weighted-average results are close to the micro-average results, which leads us to believe that class imbalance has no significant impact in our models. The average accuracy for the XGBoost classification model after tuning is 73.69% and the same for RF is around 82.39%. The results from XGBoost and RF are not that impressive as many instances which are at lower threshold are misclassified as upper threshold which will lead to false alarms in a real world scenario. The performance of RF is slightly better, but leaves room for improvement.

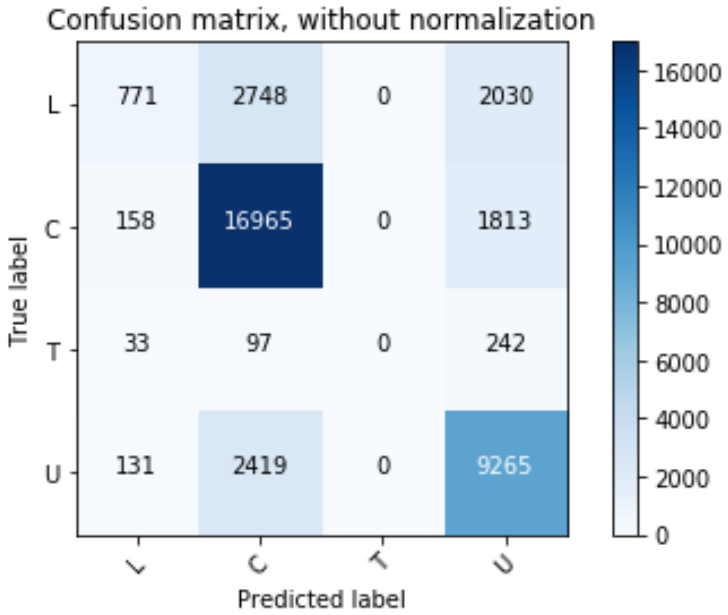


Figure 4.7. Confusion matrix for XGBoost-based foaming classification model for 4 thresholds.

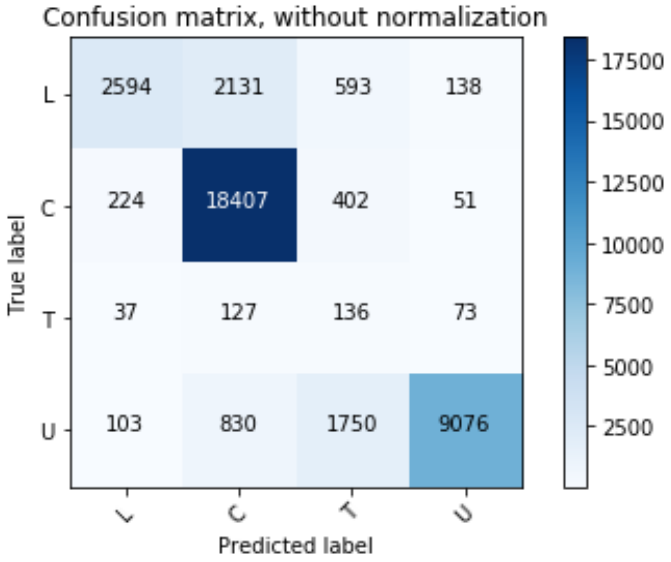


Figure 4.8. Confusion matrix for RF-based foaming classification model for 4 thresholds.

Table 4.3. Foaming prediction classification model evaluation summary using XGBoost and different averaging techniques for metric calculation.

Averaging Technique	Accuracy (in %)	Precision (P)	Recall (R)	F1-score
Micro	73.69	0.74	0.74	0.74
Macro	73.69	0.54	0.45	0.45
Weighted	73.57	0.72	0.74	0.70

Table 4.4. Foaming prediction classification model evaluation summary using Random Forest and different averaging techniques for metric calculation.

Averaging Technique	Accuracy (in %)	Precision (P)	Recall (R)	F1-score
Micro	82.39	0.82	0.82	0.82
Macro	82.39	0.69	0.65	0.62
Weighted	82.39	0.89	0.82	0.84

4.4.3 Foaming prediction regression model

After attempting to classify the foaming by ensemble methods, we build regression models which would exactly predict the numeric exhaust differential pressure at a particular instant. We use the same two ensemble methods for model building and choose RMSE values as performance indicators. As shown in Table 4.5, RF outperforms XGBoost again with an RMSE of 12.25 mbarg with comparison to XGBoost with an RMSE value of 18.61 mbarg. RF outperforms XGBoost for both classification and

regression models, which can be attributed to easier parameter tuning in RF ensemble methods. Both XGBoost and RF are generally expected to give similar results, which are relatively better than ordinary boosting and bagging methods and singular strong algorithms.

To compare with other popular algorithms for regression, we build two other models based on popular methods like neural networks and support vector regression (SVR). To compare with other popular algorithms for regression, we build two other models using popular methods like neural networks and support vector regression (SVR). For neural network, we build a network with 2 hidden layers (with 11 and 5 neurons respectively). Each hidden layer uses the rectified linear unit (ReLU) activation function and the neural net uses Adam optimization for training.[37-38] For SVR, we use radial basis function (RBF) kernel due to the non-linear nature of the data.[39] A grid-search technique is applied for tuning the hyperparameters: Gamma (γ), which is the regularization parameter and Sigma^2 (σ^2), which is the RBF kernel function parameter. The results show that ensemble methods perform better than these singular robust algorithms for our dataset. While, further optimization can be done to enhance the performance of these algorithms, ensemble methods provide us with the desired output.

Table 4.5. Model evaluation summary using ensemble methods and other popular methods for regression.

Ensemble Technique	**Model Type**	**Description**	**RMSE**
XGBoost	Regression	Foaming Prediction (without tuning)	45.72 mbarg
XGBoost	Regression	Foaming Prediction (with tuning)	18.61 mbarg
Random Forest	Regression	Foaming Prediction	12.25 mbarg
Neural Network	Regression	Foaming Prediction	40.99 mbarg
SVR (with RBF kernel)	Regression	Foaming Prediction	52.46 mbarg

4.5 Model implementation

We can implement the ensemble-based classification and regression models for similar datasets by following the steps shown in Figure 4.9. Python, an open-source programming language, includes

packages for both XGBoost and Random Forest ensemble methods.[31-34] We can do preprocessing with the help of packages like Scikit-Learn and Pandas.[31-32] The criterion for the training to test split is normally based on two conditions for the test set: i) the test set is large enough to yield statistically meaningful results, and ii) the test set is representative of the data set as a whole. In case of unbalanced datasets, we may consider stratified sampling, where each strata (subgroup) of a given dataset is adequately represented.

Depending on the size and nature of the dataset, training and hyperparameter tuning can take significant process time. However, we can use parallel computing techniques to reduce the processing time. In XGBoost, such parallelization is done within a single tree since it is a sequential ensemble method. By contrast, in Random Forest, parallelization is done for separate trees at the same time since it is a parallel ensemble method. A hyperparameter is a parameter whose value is used to control the learning process of a base learner. Hyperparameter optimization is essential to find the optimal model which minimizes the loss function. A grid search exhaustively generates candidates from a grid of parameter values and all possible combinations of parameter values are evaluated and the best combination is retained. For XGBoost, we tune the learning rate, maximum depth, and the number of boosting rounds, etc. While, for random forest, we tune number of estimators, maximum depth, minimum sample split, etc.

After a model is optimized with the ideal hyperparameters, we perform model evaluation with established methods like k-fold cross validation as explained in Section 4.2.2. Out-of-bag (OOB) error is one of the methods for predicting error in random forests. It avoids the need for an independent validation dataset, but often underestimates actual performance and the optimal number of iterations. Model performance metrics like accuracy can be used alongside confusion matrix to demonstrate prediction results of a classifier. For regression models, we can apply metrics like root mean squared error (RMSE) to quantify the model accuracy.

Figure 4.9. Model implementation of ensemble-based methods.

4.6 Conclusions, Limitations, and Future Research

In this paper, we describe the potential of ensemble learning methods like XGBoost and Random Forest in real-industrial data. We propose a novel application of such meta-algorithms in fermenter classification and foaming prediction. Our experience shows that regression-based ensemble methods can successfully be implemented to build an antifoam addition profile; such a profile will eliminate the need for speculative addition of defoamers and overcome the limitations associated with foaming which causes problems in several bioreactors and is not limited to biological plants. With the help of ensemble-based machine learning, we can use the already available industrial operating data to minimize issues such as foaming and maximize the yield of the product.

The real-industrial data acquired is based on a single strain of organism for consistency. It would be interesting to see if the model gives a good performance for different strains or different microorganisms. The hyperparameter tuning for XGBoost and RF is cursory and an in-depth parameter tuning leaves room for further improvements in the models, especially for XGBoost, after its underperformance with comparison to RF. We compared the ensemble methods with some strong singular algorithms like neural networks and SVR. Even though ensemble methods outperformed these algorithms, we believe they can still provide competitive results for a different dataset.

We plan to implement the foaming prediction results to build an antifoam profile for future batch runs which would allow us to have more control over foaming. The passage for data integration opened by successful classification of fermenters allows us to dive into big fermenter data analysis as we are able to combine the data from every batch to produce a large dataset which helps us build more robust models.

.

Symbols

English and Greek Symbols

y = set of predicted (sample, label) pairs

$\hat{y}$= set of true (sample, label) pairs

y_l= subset of y with class c

$\hat{y}_l$= subset of $\hat{y}$ with class c

C= set of classes

L= loss function

Ω= regularization term

f_k = decision tree prediction

x_i = feature vector for t-th datapoint

$q_{(x)}$= directing function

I_L = indices assigned to left leaves

I_R = indices assigned to right leaves

γ= pseudo-regularization parameter (Lagrange multiplier)

$\hat{Y}_e$= prediction by the e-th tree

$\hat{Y}$= final prediction

X_i= vector of independent variables

Y_i = dependent variable of interest

$\mathrm{p}_i = \text{the probability that an item in S belongs to class C}$

γ_r = regularization parameter in SVR

σ^2 = RBF-kernel function parameter

References

1. Golemanov, K.; Tcholakova, S.; Denkov, N. D.; Ananthapadmanabhan, K. P.; Lips, A. Breakup of bubbles and drops in steadily sheared foams and concentrated emulsions. *Physical Review E* **2008**, *78,* 051405.
2.Maza-Márquez, P.; Vílchez-Vargas, R.; Boon, N.; González-López, J.; Martínez-Toledo, M.; Rodelas, B. The ratio of metabolically active versus total Mycolata populations triggers foaming in a membrane bioreactor. *Water Research* **2016**, *92*, 208.
3. Maza-Márquez, P.; Vilchez-Vargas, R.; Kerckhof, F.; Aranda, E.; González-López, J.; Rodelas, B. Community structure, population dynamics and diversity of fungi in a full-scale membrane bioreactor (MBR) for urban wastewater treatment. *Water Research* **2016**, *105*, 507.
4. Cosenza, A.; Bella, G. D.; Mannina, G.; Torregrossa, M. The role of EPS in fouling and foaming phenomena for a membrane bioreactor. *Bioresource Technology* **2013**, *147*, 184.
5. Coutte, F.; Lecouturier, D.; Yahia, S. A.; Leclère, V.; Béchet, M.; Jacques, P.; Dhulster, P. Production of surfactin and fengycin by Bacillus subtilis in a bubbleless membrane bioreactor. *Applied Microbiology and Biotechnology* **2010**, *87*, 499.
6. Bella, G. D.; Torregrossa, M. Foaming in membrane bioreactors: Identification of the causes. *Journal of Environmental Management* **2013**, *128*, 453.
7. Kougias, P. G.; Francisci, D. D.; Treu, L.; Campanaro, S.; Angelidaki, I. Microbial analysis in biogas reactors suffering by foaming incidents. *Bioresource Technology* **2014**, *167*, 24.
8. Routledge, S. J. Beyond de-foaming: the effects of antifoams on bioprocess productivity. *Computational and Structural Biotechnology Journal* **2012**, *3,* 10001.
9. Kar, T.; Destain, J.; Thonart, P.; Delvigne, F. Scale-down assessment of the sensitivity of Yarrowia lipolytica to oxygen transfer and foam management in bioreactors: investigation of the underlying physiological mechanisms. *Journal of Industrial Microbiology & Biotechnology* **2011**, *39*, 337.
10. Delvigne, F.; Lecomte, J.-P. Foam Formation and Control in Bioreactors. *Encyclopedia of Industrial Biotechnology*; ACS: Washington, DC, 2010; p 1.
11. Nielsen, J. C.; Lino, F. S. D. O.; Rasmussen, T. G.; Thykær, J.; Workman, C. T.; Basso, T. O. Industrial antifoam agents impair ethanol fermentation and induce stress responses in yeast cells. *Applied Microbiology and Biotechnology* **2017**, *101*, 8237.
12. López-Barajas, M.; López-Tamames, E.; Buxaderas, S.; Tomás, X.; Torre, M. C. D. L. Prediction of Wine Foaming. *Journal of Agricultural and Food Chemistry* **1999**, *47*, 3743.
13. Birk, W.; Arvanitidis, I.; Jonsson, P.; Medvedev, A. Physical modeling and control of dynamic foaming in an LD-converter process. *IEEE Transactions on Industry Applications* **2001**, *37*, 1067.
14. Oyetunde, T.; Bao, F. S.; Chen, J.-W.; Martin, H. G.; Tang, Y. J. Leveraging knowledge engineering and machine learning for microbial bio-manufacturing. *Biotechnology Advances* **2018**, *36*, 1308.
15. Hansen, K.; Biegler, F.; Ramakrishnan, R.; Pronobis, W.; Lilienfeld, O. A. V.; Müller, K.-R.; Tkatchenko, A. Machine Learning Predictions of Molecular Properties: Accurate Many-Body Potentials and Nonlocality in Chemical Space. *The Journal of Physical Chemistry Letters* **2015**, *6*, 2326.
16. Timoshenko, J.; Lu, D.; Lin, Y.; Frenkel, A. I. Supervised Machine-Learning-Based Determination of Three-Dimensional Structure of Metallic Nanoparticles. *The Journal of Physical Chemistry Letters* **2017**, *8*, 5091.
17. Costello, Z.; Martin, H. G. A machine learning approach to predict metabolic pathway dynamics from time-series multiomics data. *NPJ Systems Biology and Applications* **2018**, *4,* 1.
18. Oyetunde, T.; Liu, D.; Martin, H. G.; Tang, Y. J. Machine learning framework for assessment of microbial factory performance. *Plos One* **2019**, *14*, 1.

19. Varley, J.; Brown, A. K.; Boyd, J. W. R.; Dodd, P. W.; Gallagher, S. Dynamic Multi-Point Measurement of Foam Behavior for a Continuous Fermentation over a Range of Key Process Variables. *Biochemical Eng. J.,* **2004**, *20*, 61.
20. Brown, G. Diversity Creation Methods: A Survey and Categorisation. *Information Fusion* **2004,** *6,* 5.
21. Alfaro, E.; Gámez, M.; García, N. Ensemble Classification Methods with Applications in R. Wiley & Sons: New York, NY, 2018, p 31.
22. Dev, V. A.; Datta, S.; Chemmangattuvalappil, N. G.; Eden, M. R. Comparison of Tree Based Ensemble Machine Learning Methods for Prediction of Rate Constant of Diels-Alder Reaction. *Computer Aided Chemical Engineering 27th European Symposium on Computer Aided Process Engineering* **2017**, *40*, 997.
23. Borysik, A. J.; Kovacs, D.; Guharoy, M.; Tompa, P. Ensemble Methods Enable a New Definition for the Solution to Gas-Phase Transfer of Intrinsically Disordered Proteins. *Journal of the American Chemical Society* **2015**, *137*, 13807.
24. Gulyani, B. B.; Fathima, A. Introducing Ensemble Methods to Predict the Performance of Waste Water Treatment Plants (WWTP). *International Journal of Environmental Science and Development* **2017**, *8*, 501.
25. Amozegar, M.; Khorasani, K. An ensemble of dynamic neural network identifiers for fault detection and isolation of gas turbine engines. *Neural Networks* **2016**, *76*, 106.
26. Zhou, Z. H., Ensemble Methods: Foundations and Algorithms, CRC Press, Boca Raton, FL., **2012**.
27. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD 16,* San Francisco, CA, August, **2016**.
28. Breiman, L. Random Forests. *Machine Learning* **2001**, *45*, 5.
29. Tan, P.-N.; Steinbach, M.; Karpatne, A.; Kumar, V. *Introduction to Data Mining*; Pearson Education, Inc.: New York, NY, **2019**, p 64.
30. Wold, S.; Ruhe, A.;Wold, H.;Dunn, W. The Collinearity Problem in Linear Regression: The Partial Least Square (PLS) Approach to Generalized Inverse. *Soc.Ind.Appl.Math*, **1984**, *5*, 735.
31. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* **2011**, *12*, 2825.
32. McKinney, W. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference,* Austin, TX, June, **2010.**
33. Hunter, J. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering* **2007**, *9*, 90.
34. Waskom, M.; Botvinnik, O.; Okane, D.; Hobson, P.; Halchenko, Y.; Lukauskas, S.; Cole, J.; Warmenhoven, J.; Ruiter, J.; Hoyer, S.; Vanderplas, J.; Villalba, S.; Kunter, G.; Quintero, E.; Martin, M.; Miles, A.; Meyer, K.; Augspurger, T.; Yarkoni, T.; Bachant, P.; Williams, M.; Evans, C.; Fitzgerald, C.; Wehner, D.; Hitz, G.; Ziegler, E., Qalieh, A.; Lee, A. Seaborn: v0.7.0, 2016. URL: https://doi.org/10.5281/zenodo.54844.
35. Jagani, H.; Hebbar, K.; Gang, S. S.; Raj, P. V.; H., R. C.; Rao, J. V. An Overview of Fermenter and the Design Considerations to Enhance Its Productivity. *Pharmacologyonline* **2010**, *1*, 261.
36. Kohavi, R. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. *International Conference on Artificial Intelligence*, Montreal, Canada, Aug. **1995**.
37. Baughman, D. R.; Liu, Y. A. Neural Networks in Bioprocessing and Chemical Engineering, **1995**, Elsevier, Atlanta, GA.
38. Kingma, D.; Ba, J. Adam: A Method for Stochastic Optimization, *International Conference on Learning Representations*, San Diego, CA. **2015**.
39. Cristianini, N.; Shawe-Taylor, J. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods, Cambridge University Press, London, UK, 2000*, pp.149–161.

Chapter 5. Large-Scale Industrial Fermenter Foaming Control: Automated Machine Learning for Antifoam Prediction and Defoaming Process Implementation

5.1 Introduction

5.1.1. Foam Formation and Foaming in Bioreactors

Foam is a form of dispersed medium with a well-defined complex structure composed of gas pockets separated by liquid membranes. Foaming behavior is usually affected by liquid properties such as surface tension, viscosity, and ionic strength. Foams can be classified into different categories based on their behaviors; they can be unstable (transient), metastable, or persistent.[1] Metastable and persistent foams, which are a result of relatively concentrated surfactant mixtures, have the longest lifetime if left undisturbed.[1]

A foam structure is usually polydisperse in nature with a varying particle size distribution [2] and is influenced by the liquid fraction, which is the proportion of fluid in the foam. An increase in the liquid fraction results in structural change, allowing the foam to adopt a spherical configuration as it transitions from dry foam to wet foam.[3] Foaming in bioreactors can occur because of multiple factors and the foaming agent(s) can come from both the media components used for fermentation and the metabolic activities of the microorganism used for fermentation.[4] Different operational factors can impact foam formation inside a bioreactor.[5,6] Higher airflow rate along with several other operational and design variables like temperature, pH, microorganism type (sporulating/ non-sporulating), choice of impellers, positioning of baffles, aeration zones, etc. impact foaming in one way or the other.

Some of the operational changes which can mitigate foaming include reduced aeration, lower temperature for broth, the addition of anti-foaming agents, reduced agitation, minimal broth usage, and usage of mechanical foam-breakers[5]; however, these techniques can have negative impacts on a fermentation process. Therefore, to maximize process efficiency without any severe repercussions, it is important to optimize the addition of antifoam in a bioreactor.

5.1.2 Antifoam Addition and Defoaming Practices

High film elasticity, high surface and bulk viscosity, and high solid content are some foam stabilizing factors. Antifoams prevent foaming in a system in one of two ways. They either displace the foam stabilizing component from the bubble wall or locally burst the bubbles. A typical antifoam consists of oil, hydrophobic solid particles, or a mixture of both. Traditionally, optimization of an antifoam addition profile for a particular bioprocess requires a thorough understanding of the surface activities which

impact the stabilization of foam, specifications for the different fermenter designs, and the knowledge of the physico-chemical properties of the foam itself.[7]

Antifoam addition can alter the dynamics of the bioprocess both directly (by preventing foaming) and indirectly (by impacting oxygen transfer, clogging filters downstream, interacting with cell membranes and other components of the chemical broth, etc.) Several studies are available that quantify the different impacts of foaming and antifoam addition on a bioprocess and other industrial sectors.[7-13] Velugula-Yellela et. al.[8] use change in the local dissolved oxygen variability to predict foaming; in their study, they observe the impact of antifoam and media selection on cellular health and production. In a similar study, Nielsen et. al.[9] show that industrial antifoam agents compromised the growth rates and the glucose uptake rates for an ethanol production process.

The addition of suitable antifoam agents effectively defoams a bioprocess, but excessive addition and post hoc addition of antifoam can severely degrade the product quality without controlling foaming efficiently. It is critical to use the optimal initial antifoam volume fraction (volume of antifoam mixed with media) for the desired defoaming actions.[10] Adding the right amount of antifoam is also important to maintaining a sufficiently high oxygen transfer rate (crucial for a fermentation process); McClure et al.[11] in their experiment quantify the impact of commercial antifoams in terms of foam suppression, oxygen transfer rate reduction, and time-dependent deactivation.

Typically, bioreactors are equipped with different foam sensors or use various foam detection methods to automatically add antifoam, or the antifoaming agents are added manually in response to a foaming indication. Over time, there have been several reported defoaming strategies to mitigate the adverse effect of excessive antifoam addition, such as pH adjustment and high-temperature neutral stripping[12], mechanical foam breakers[13], etc. In this work, we propose a unique defoaming strategy that uses data analytics tools to control the antifoam addition. Using our defoaming strategy, we attempt to remove the post hoc aspect of antifoam addition and develop specific (targeted) profiles for antifoam addition. In our previous work, we demonstrated that fermenters can be classified without any prior information on fermenter design and type of microorganism; we also predicted the onset of foaming using ensemble methods.[14]

Through this work, we take our work to the next step by demonstrating the use of automated machine learning and by using exploratory time-series data analytics for industrial antifoam profile deployment. We use exhaust differential pressure as an indicator of foaming. As shown in Figure 5.1, exhaust differential pressure is the difference between the fermenter head pressure and the exhaust

line pressure. We consider foaming to occur when the *exhaust differential pressure crosses 100 mbarg* based on past industrial fermentation experience.

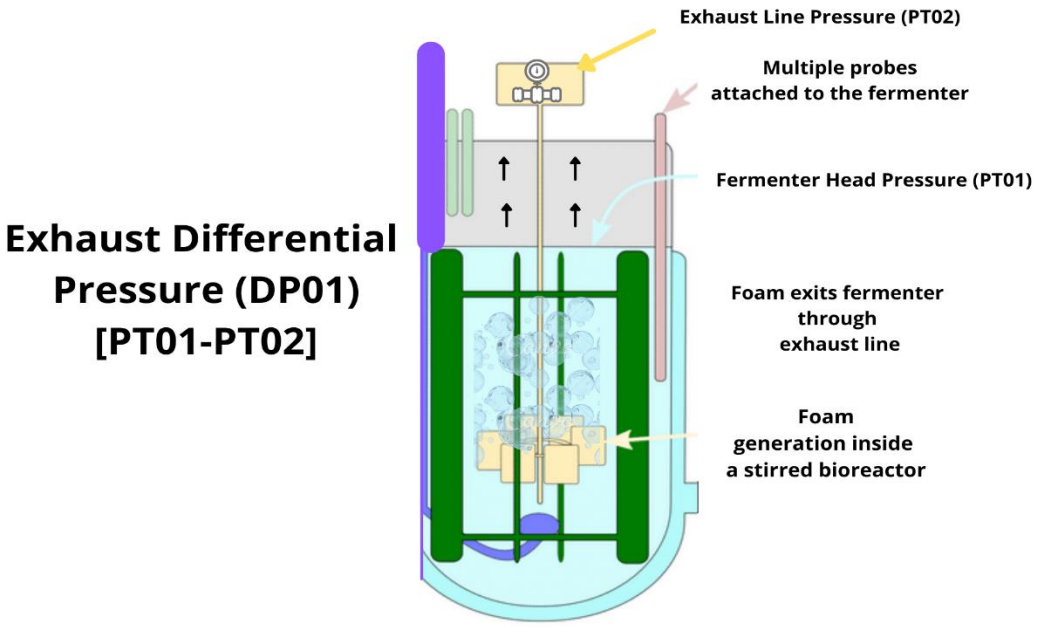


Figure 5.1. Fermenter design with pressure notations for exhaust differential pressure calculations, created using Canva and adapted from reference [5]

5.2 Dataset Extraction

We extract the key process variables to build the antifoam prediction model based on an experimental setup to study the dynamic measurements of foaming behavior for a continuous fermenter in Figure 5.2.[6] The setup and additional details in reference 6 suggest the key measurement considerations in foaming control include probes (pH, temperature, and dissolved oxygen), the volume of a fermenter, agitator speed, and time for fermentation. In building the dataset for our current study, we use the key plant data following references 5 and 6. Table 5.1 summarizes 12 independent variables (X) and 1 quality variable (Y). Our dataset consists of 163 batches with four fermenter designs, totaling about half a million instances.

We choose hours (fermentation progress time) as an independent variable because of two reasons. First, none of the other independent variables is linearly correlated with time, and second, the fermenter data are collected for a multi-batch process for the same time-periods. For cases where time linearly changes with other variables or if data are collected in different periods for multiple batches, we should remove time as an independent variable as it becomes redundant or an observational ID.

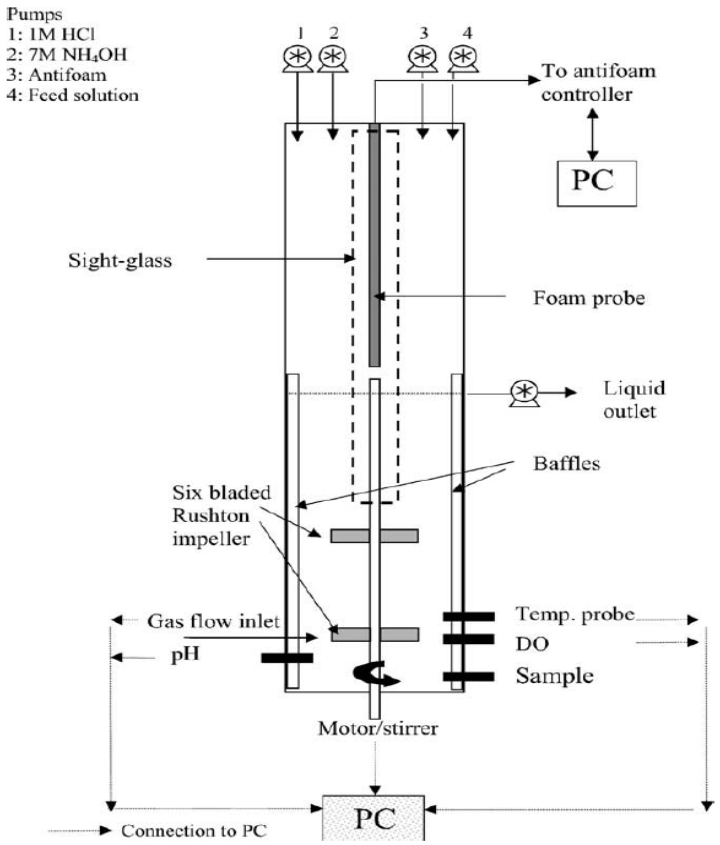


Figure 5.2. Foaming control setup in a bioreactor, Reprinted from ref. 5. Copyright 2004 Elsevier.

Table 5.1. List of dependent and independent variables

Independent Variables	Dependent Variable
Hours	Antifoam Flow (in L/min)
pH	
Acid Flow (in L/min)	
Caustic Flow(in L/min)	
Fermenter Temperature (in Celsius)	
Dissolved Oxygen(%)	
Air Flow (in L/min)	
Volume (in L)	
Air Valve Position (% open)	
Agitator Speed (%)	
Exhaust Differential Pressure (in mbarg)	
Maximum Antifoam Addition (in L)	

5.3 Defoaming Strategy

From our last report, we were able to integrate fermenter data based on independent variables alone, without specifying fermenter specifications such as straight wall height, aspect ratio, total volume, number of impellers, etc.[14] We also accurately predicted the onset of foaming using ensemble-based methods and compared them to other ML regression methods like Support Vector Machine (SVM) and neural networks.[14] This paper demonstrates two different approaches to mitigate foaming:
1) In the first approach (Section 5.4), we use the integrated data for each fermenter type and the combined dataset from all the fermenters to predict antifoam addition directly using an automated regression algorithm (TPOT) and compare it with ensemble-based methods like Random Forest and Extreme gradient boosting (XGBoost). We do not use this approach for deployment in the industry because of its limitations, but showcase the possibility of future usage as discussed later in Section 5.4.4 and Section 5.5.2.
2) For the second approach (Section 5.5), we use exploratory time series analysis and stepwise addition to build generalized profiles for all fermenter types and targeted deployment-ready profiles, which are specific to each organism in each fermenter type. We test these profiles on the fermenters and discuss the results in this paper later in Section 5.5.4 and Section 5.6.

5.4 Automated Machine Learning for Antifoam Prediction

A successful ML model requires scanning through an exhaustive list of appropriate machine learning algorithms with multiple steps for feature selection, preprocessing, and hyperparameter tuning. Because of the plethora of algorithms available and the necessity for domain expertise, finding and

using the right ML tool can be quite challenging, especially in an industrial setting. For our goal of building antifoam profiles, we use an automated-machine learning (AML) model called TPOT (Tree-based Pipeline Optimization Tool). [15] By using AML, we can test hundreds of pipelines of successful algorithms and select the best pipeline based on the evaluation metric. AML tools use pre-designed data analytics structures, following the traditional sequential steps of training, tuning, and testing a data analysis pipeline. Such a tool helps researchers save quality time and provides the right pathway towards finding accurate results. Recently, AML has found its footing in several industrial and research projects in bioindustries and other chemical engineering sectors. [16-20] In this paper, we use TPOT to enhance our workflow after data acquisition, data cleaning and visualization, and data integration and transformation.

5.4.1 TPOT Framework and Methodology

TPOT, developed by Olson et al[15], is an AML tool that uses genetic programming to explore thousands of pipelines to find the best ML framework. Genetic programming is a well-known evolutionary computation technique that has three properties: selection, crossover, and mutation. In the selection step, we search for all possible random solutions to a given problem (referred to as the population) and evaluate how fit each solution is for a given fitness function. The next step is crossover, where we select the fittest solution for every iteration in the optimization process (referred to as a generation) and perform crossover to create a new population. The last step is mutation, where we take the new population and mutate them with random modifications and repeat the process.
These stochastic changes can have positive or negative effects on the performance of the pipelines, thereby allowing the algorithm to explore pipelines that were not considered before. After every crossover step, the worst-performing pipelines are removed from the population; after a fixed number of generations, TPOT recommends the best pipeline.[17] The pipelines generated by TPOT include several different robust models from the Sklearn library such as Logistic Regression, SVM, K-nearest neighbor, Random Forest, etc. TPOT also occasionally learns pipelines that stack these estimators to create a new pipeline.

Figure 5.3 illustrates the TPOT framework for building an AML model. After the initial exploratory analysis and data integration, the TPOT algorithm automatically constructs and optimizes all the later steps.

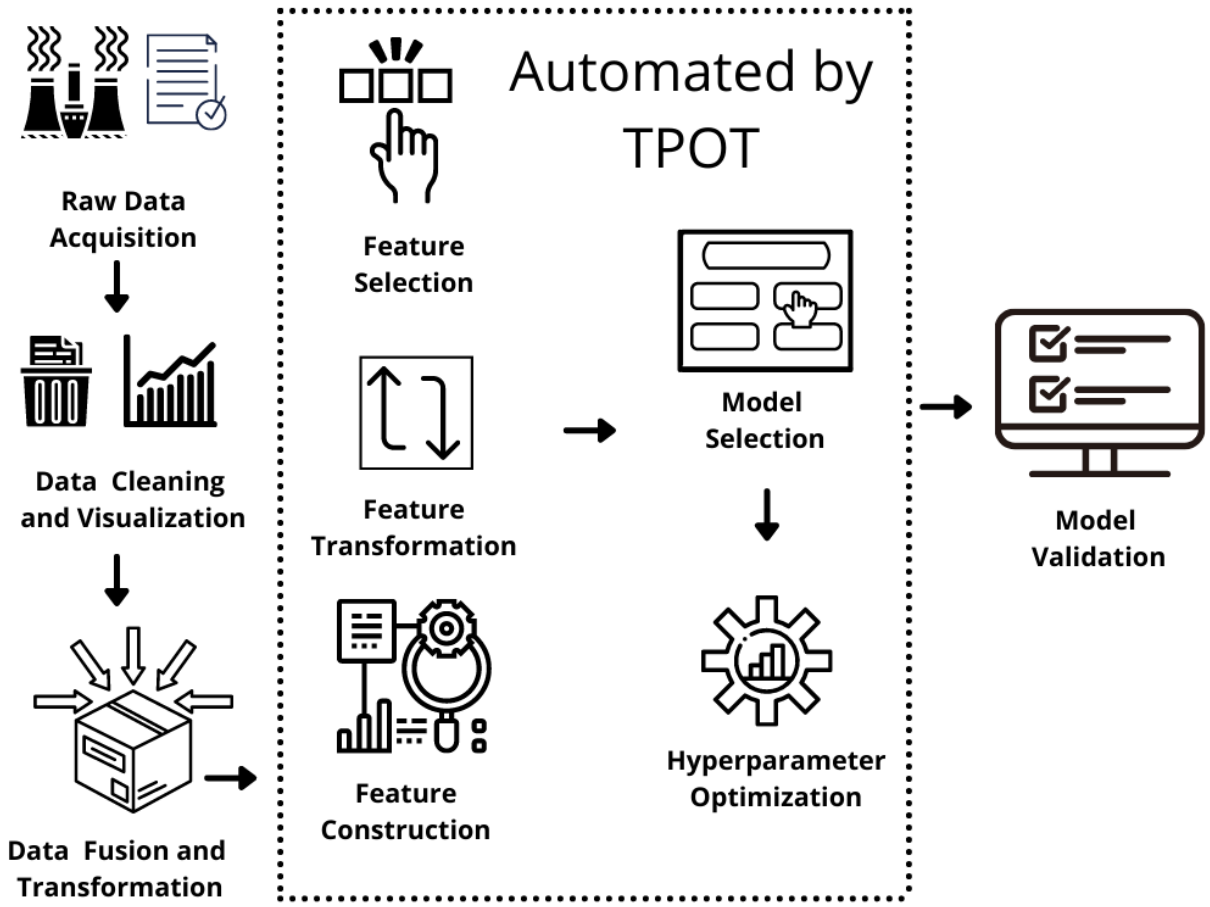


Figure 5.3. TPOT framework for model building, created using Canva and adapted from reference [15]

5.4.2 Data Preprocessing before Automation

Data preprocessing includes cleaning, visualization, fusion, and transformation. Data cleaning identifies misleading and mislabeled records, and removes noisy and redundant data, which may occur because of usage of multiple temperature sensors and pressure gauges, multiple dissolved oxygen (DO) probes, multiple identifiers for the same independent variables, and different fermenter design setups. The raw data consist of several columns with redundant data because of the presence of backup probes. We remove all the columns with identical data for temperature, pressure, dissolved oxygen, etc. Similarly, we remove some data columns, which play no role in antifoam addition profiling, like inoculation date, speculative batch count, etc.

5.4.2.1 Data Visualization

We use Python 3.8 and its various packages (e.g., Scikit-learn machine learning python library) for the following visualizations and the proposed models.[21-24] SHAP (Shapley Additive Explanations)

proposed by Lundberg et al.[24] is a method to explain individual predictions based on game theory. We use random forest feature importance along with TreeSHAP to get the plot shown in Figure 5.4. The variables with the highest importance are placed at the top and the lowest variables are placed at the bottom. Some advantages of SHAP-based plotting over traditional importance plots are: (1) SHAP-based plots can highlight both the importance of the independent variables and the positive and negative relationships of the independent variables with the dependent variable. (2) the SHAP-based plot includes every single observation as shown by each dot in the plot. Traditional importance plots only show the trend on a generalized basis and do not account for individual cases.

From Figure 5.4, we see that exhaust differential pressure is the most important feature along with hour, volume, air valve position, etc. This is expected as the antifoam addition should rely heavily on the current exhaust differential pressure. We can also see the directionality of the importance as SHAP values in the x-axis indicate whether the independent variable will result in a positive (higher) or negative (lower) effect on the dependent variable. In our case, we see that the exhaust differential pressure has both positive and negative impacts on the current antifoam addition. We can also see two extreme observational cases from the plot: a) lower volume resulting in lower antifoam addition. b) lower airflow also resulting in lower antifoam addition. Such anomalies in observations can be excluded from the training dataset to ensure better model performance and in some cases, they also help us identify problems in the fermentation process.

To identify the anomalies in our batches, we use the multivariate data analysis software Aspen ProMV (available through university programs of Aspen Technology Inc.) to identify the underlying sources of variation. The software helps us generate a PCA-based (principal component analysis) hyper-ellipse score plot, as shown in Figure 5.5. For this particular figure, we use a batch from Fermenter A with about 3000 observations to generate the ellipse plot. We see that some of the observations depart from the cluster as outliers. For instance, observations from 660-678 show deviation from the entire dataset as they fall outside the 95% confidence interval along with some other observations as shown in Figure 5.5. We can remove these observations from our training dataset for better model results.

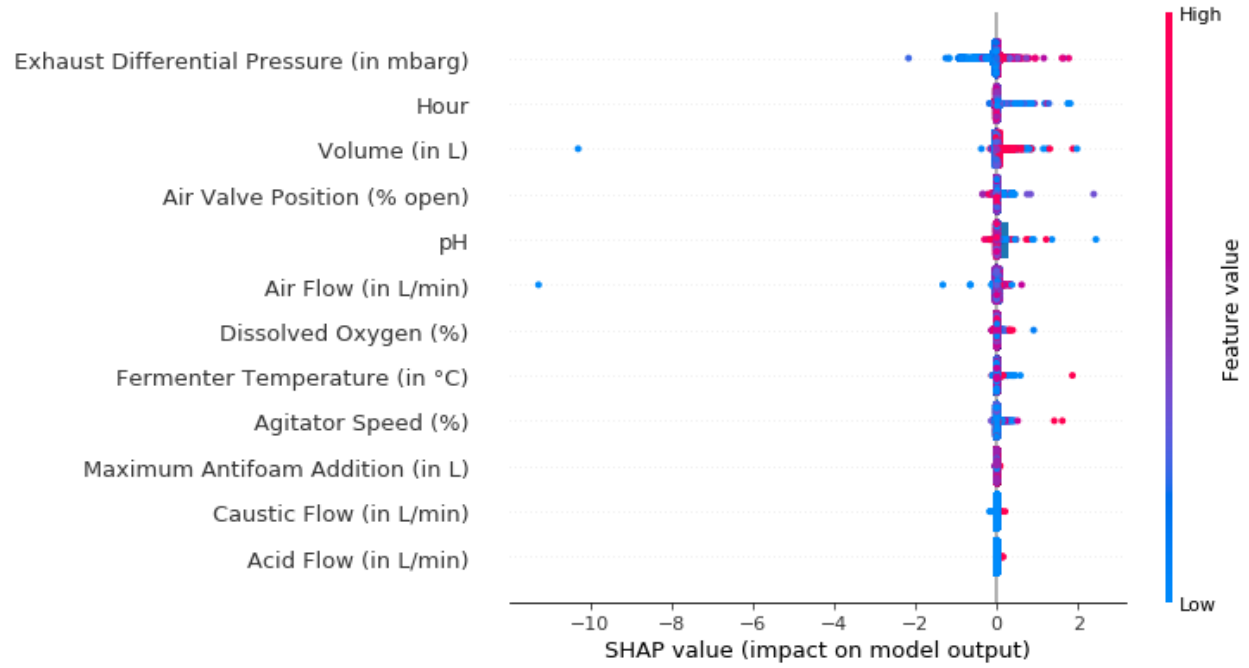


Figure 5.4. Random Forest Feature Importance computed using SHAP values.

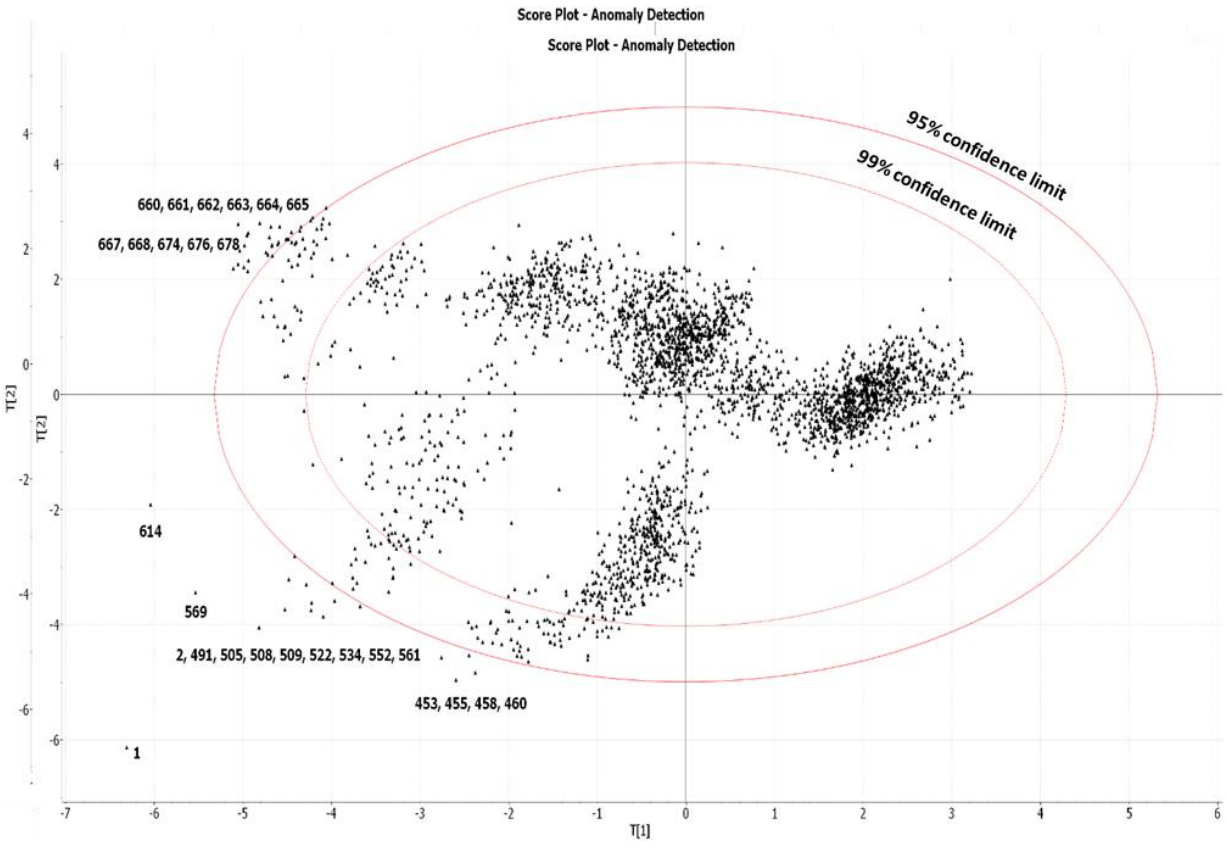


Figure 5.5. PCA-based hyper ellipse score plot

5.4.2.2 Data Fusion and Transformation

Data fusion involves combining datasets from different fermenters to form a generalized dataset for each fermenter type. We then segregate datasets based on the six different types of strains to profile the antifoam addition for each strain. Data transformation involves changing the cumulative values of some independent variables (such as acidic flow, caustic flow, antifoam flow, and volume of the reactor) to non-cumulative values. Data transformation within the automated TPOT algorithm involves maximum absolute value scaling of the data as prescribed by the TPOT algorithm. This scaling ensures that the value of each feature in the model is in the range of [-1,1].

5.4.3 TPOT Antifoam Addition Model Results

Table 5.2 shows the results for the antifoam addition predictive models using TPOT and ensemble-based methods. The table compares the root-mean-squared error (RMSE) values of the ensemble-methods (e.g., XGBoost and Random Forest) and TPOT method for both combined and individual datasets. We see that the RMSE values from TPOT are very similar to those of the ensemble methods and TPOT outperforms the methods with a slight margin for the combined fermenter data. The table also shows that the TPOT RMSE values for Fermenters A and D are very low (0.08-0.09). Lower RMSE values indicate a small error margin and a high accuracy. Similarly, Fermenters B and C, show a relatively high RMSE value, which possibly results from the limited amount of data available for these fermenters.

Interested readers can find the details on ensemble methods and their comparison with other regression methods from our previous work,[14] where we build regression predictive models for exhaust differential pressure. In this work, we predict the amount of antifoam addition to the fermenter directly.

Now that we have established a proper method of predicting the right amount of antifoam, we want to ensure that our method is viable in an industrial setting. By comparing the proprietary average antifoam flow rate in our industrial site with the antifoam addition RMSE, we find that our predictions have a marginal error of 3-4% per minute. While numerically our RMSE is very low, having a margin of 3-4% is not very practical for an improved profile. In general, a machine learning model suggests a maximum antifoam addition without any limit constraints. We believe that such prediction models can be used for building micro-scale profiles, but for a complete profile, they require a maximum addition constraint.

Table 5.2. Model Comparison between automated machine learning (TPOT) and ensemble methods

Algorithms	**Antifoam Addition Prediction RMSE**
Combined Dataset	
XGBoost	0.1276
Random Forest	0.13
TPOT	0.1249
Individual Dataset	
TPOT (Fermenter A)	0.09
TPOT (Fermenter B)	0.18
TPOT (Fermenter C)	0.24
TPOT (Fermenter D)	0.08

5.4.4 TPOT Prediction Model Conclusion and Limitations

With our TPOT regression model, we show that we can use automated machine learning (AML) to predict antifoam addition with comparable accuracy to fully-tuned ensemble methods. This result is very useful as it ensures that we can build models using AML algorithms like TPOT and get high accuracies as good as those given by established robust methods like random forest or extreme gradient boosting. The usage of such saves time for an engineer and removes the barrier of complexity formed by several robust machine learning algorithms.

We also acknowledge a limitation of our AML-guided TPOT antifoam prediction model. If we use the profiles suggested by the model, we cannot account for the limit in antifoam addition. To use the model successfully, we need to find a way to restrict the total antifoam addition for a given system. Furthermore, another limitation is the size of the individual dataset. We need a larger dataset for each fermenter type to improve the antifoam addition prediction model in Fermenters B and C. In the next section, we discuss how we built new deployment-ready antifoam profiles using a unique approach.

5.5. Antifoam Profile Building using Time-Series Based Exploratory Analysis and Stepwise Addition

To improve the antifoam profiles, we first explore the realized antifoam dosing strategy for each of the four fermenter types over 163 batches. We do this by grouping them by the mean value for each unique time observational ID. Table 5.3 shows the top ten peaks for the merged current antifoam addition rates for each fermenter type. From Table 5.3, we see that for fermenter A most of the addition occurs around the 13^{th} hour. Similarly, we find that most of the addition is during the 11^{th} and 37^{th} hour

for fermenter B, 15th hour for fermenter C, and 35th hour for fermenter D. Observing the overall trend of the current antifoam addition helps us understand the generalized antifoam addition pattern for each of the fermenter types.

Table 5.3. Merged antifoam profiles (largest 10 peaks) separated for each fermenter type for the entire dataset (of ~half a million instances) grouped by time.

Fermenter A		Fermenter B		Fermenter C		Fermenter D	
Time (in hrs)	Antifoam Addition (in L)	Time (in hrs)	Antifoam Addition (in L)	Time (in hrs)	Antifoam Addition (in L)	Time (in hrs)	Antifoam Addition (in L)
13.3	11.07	37.06	21.49	14.88	15.75	34.68	25.01
12.54	5.89	10.68	21.15	14.78	14.48	34.66	18.76
0.98	4.39	36.98	13.16	0.02	14.06	13.3	7.47
40.64	4.09	10.63	7.46	14.7	10.12	13.22	6.58
12.46	3.86	13.2	7.26	12.41	6.54	13.28	6.49
35.17	3.81	36.9	6.95	14.62	5.99	0.08	5.66
19.27	3.54	36.75	6	12.33	3.93	31.55	5.61
40.44	3.53	13.13	5.77	12.15	3.54	30.16	3.74
12.51	3.46	13.08	5.08	14.53	3.42	11.53	3.71
12.59	3.39	21.38	5.05	12.23	2.61	17.3	3.12

5.5.1 Generalized Fermenter Profiles

A typical industrial fermentation process can vary from a few days to months in some cases, depending on the type of reactor and the fermentation process. [25-27] For our case, we only consider the primary fermentation period of about 2-3 days. By combining all the strains and observing a generalized trend for a fermenter type, we get a base addition profile for a specific fermenter type. Since all the fermenters have different peak times, we conclude that the four different fermenters require separate antifoam addition profile analysis for the same microorganism strains. Figures 5.6-5.9 show the averaged current antifoam additions for each fermenter type for the entire 163 batches for six different strains. We use these base profiles to mark antifoam requirements for each fermenter type based on time. We conclude from these profiles that antifoam addition patterns are very different for each fermenter type.

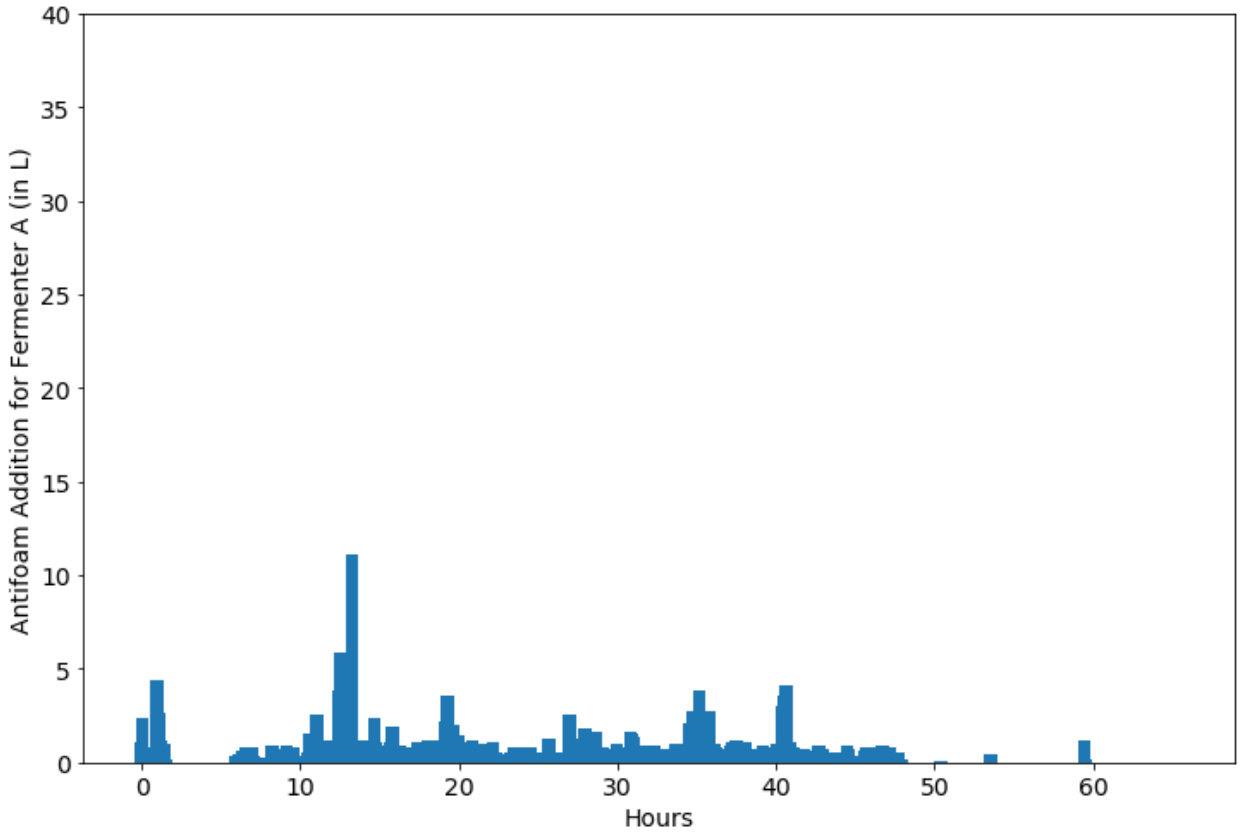


Figure 5.6. Merged current antifoam addition profile for fermenter A.

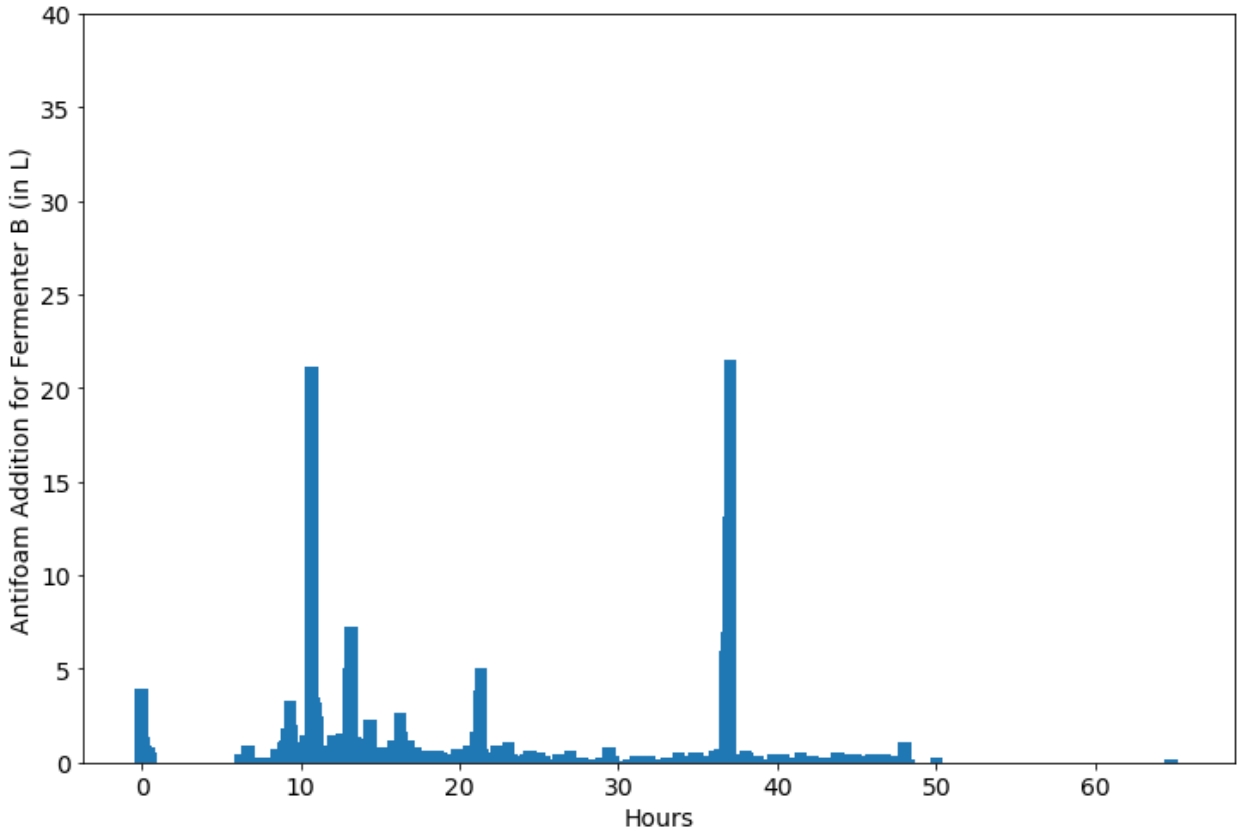


Figure 5.7. Merged current antifoam addition profile for fermenter B.

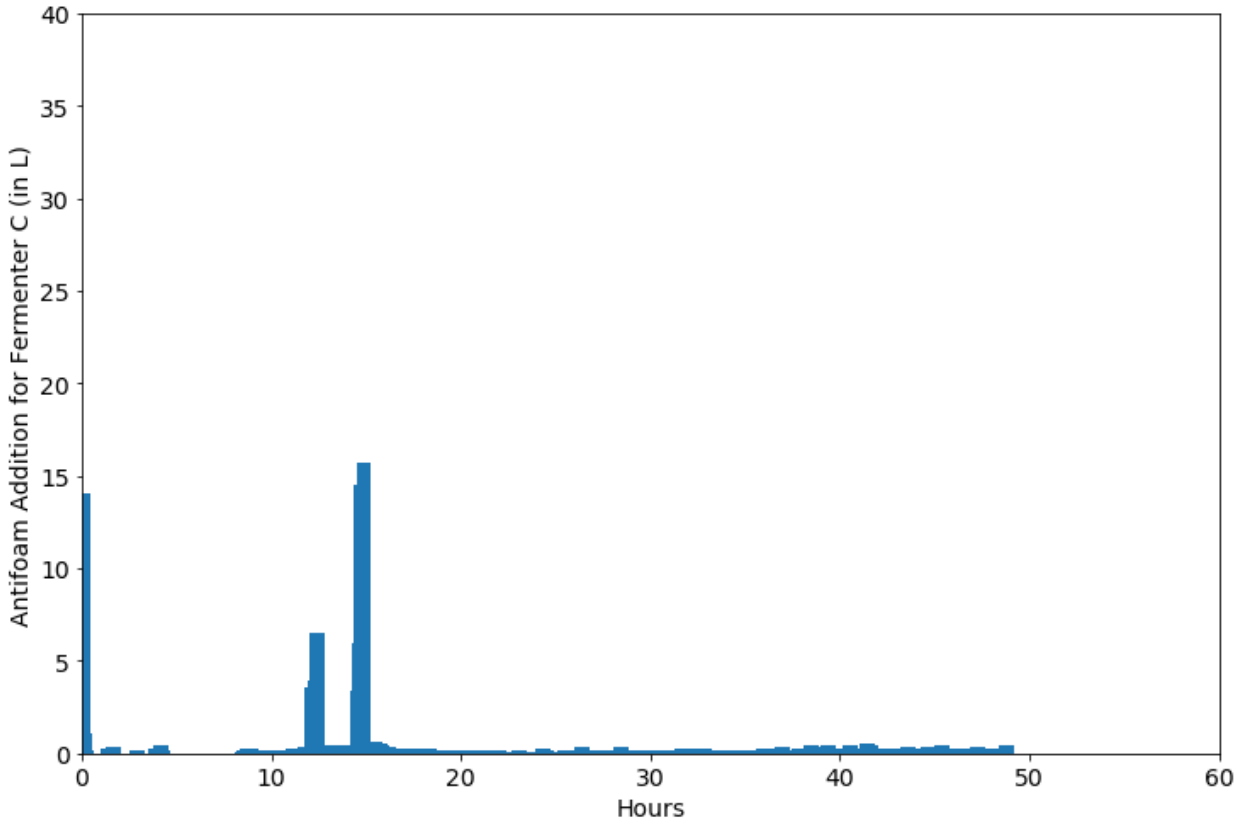


Figure 5.8. Merged current antifoam addition profile for fermenter C.

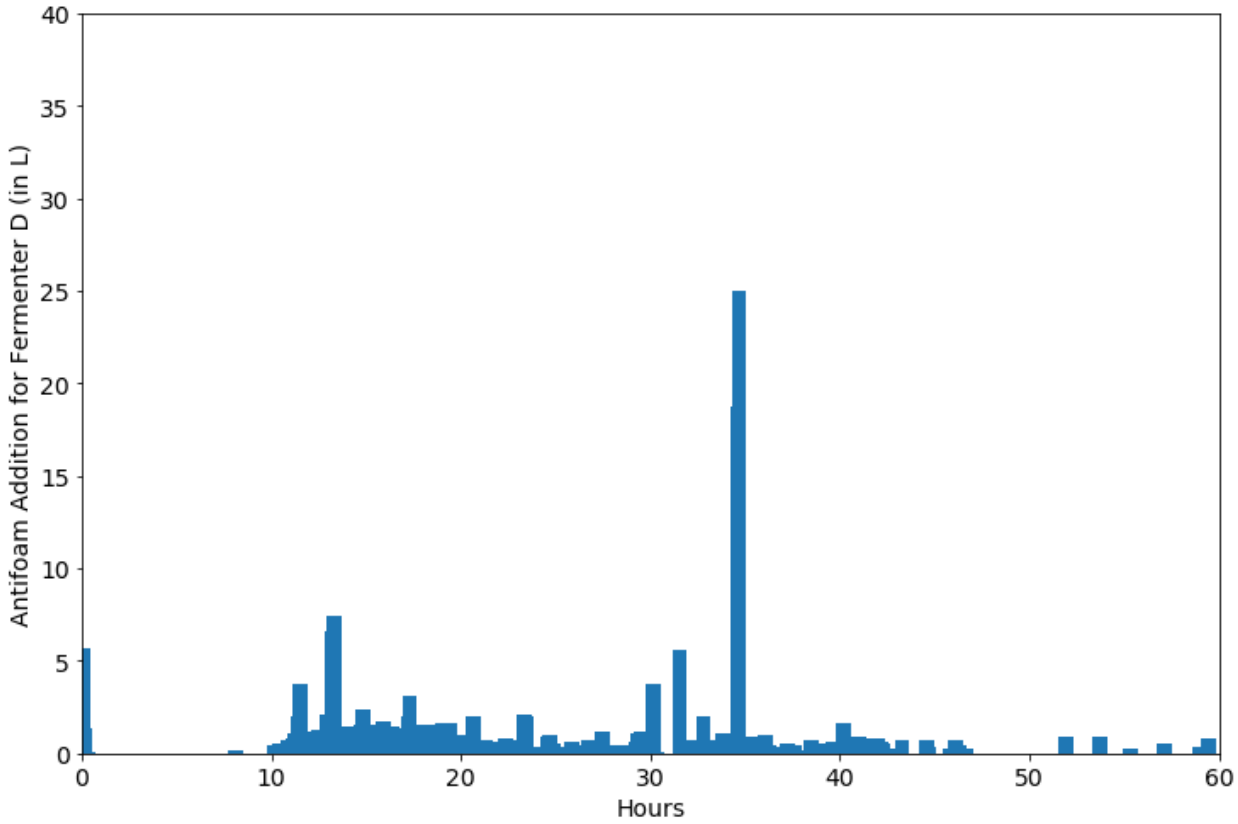


Figure 5.9. Merged current antifoam addition profile for fermenter D.

5.5.2 Deployment-Ready Targeted Profiles

The antifoam addition process is very dynamic; thus, the antifoam profile must be specified towards specific fermenter design and the microbial strain used. For reporting our analysis, we based our results on one of the six organisms, on three of the total four fermenter types, and both integrated and single batch tests.

To build new antifoam profiles, we use time-based exploratory analysis with stepwise addition. This method involves two important segments:

a) The first segment involves mapping out the current maximum antifoam addition profiles and the current maximum observed exhaust differential pressures for a specific organism in each of the fermenters. Figure 10 shows an example of the top 100 antifoam additions with corresponding exhaust differential pressure. By doing this, we mark the time levels which require adjustments in levels of antifoam addition. We see from Figure 10 that the current profile for Organism 5 on Fermenter A, has maximum antifoam addition times around the 12^{th} and 13^{th} hours, followed by the 19^{th}, 35^{th}, and 41^{st} hours. By doing this, we mark the time levels which require antifoam addition in the current profile. Then, from Figure 11, we see the observed maximum exhaust different pressure spikes occur around the

19^{th} and 25^{th}-27^{th} hours. Based on these, we mark the time levels which require antifoam additions that are absent from the current profile.

b) The second segment involves the management of antifoam addition constraints using a stepwise addition. Typically, in a fermentation process, there is a limit to the antifoam addition because of quality concerns. Most control systems for antifoam addition adhere to the limit, but the antifoam limit is reached much early in the fermentation process, resulting in poor quality towards the end. We use the time levels observed in the first segment to split the antifoam addition constraint leading to a stepwise addition of antifoam.

Figure 12 demonstrates how a sample profile for organism 5 on Fermenter A is submitted. The dotted lines indicate the stepwise constraints. In Figure 5.12, we add 25L of antifoam till the 18^{th} hour. We then add 15L of antifoam from the 18^{th} to the 27^{th} hour. Similarly, we add 10L for the remaining hours. The time levels obtained from segment 1 help us decide when to partition the antifoam constraints. By splitting the constraint into three different steps and by using time-series-based partitions, we make sure that: (1) we do not run out of antifoam towards the end; (2) we are adding more antifoam preemptively during the time slots which seem to require a higher dosage of antifoam.

We note that the micro-profiles shown in Figure 5.12, represented by the curvy lines are controlled by the on-site foaming control mechanism. It would be possible to tune these micro-profiles in the future using our TPOT-based antifoam addition prediction approach.

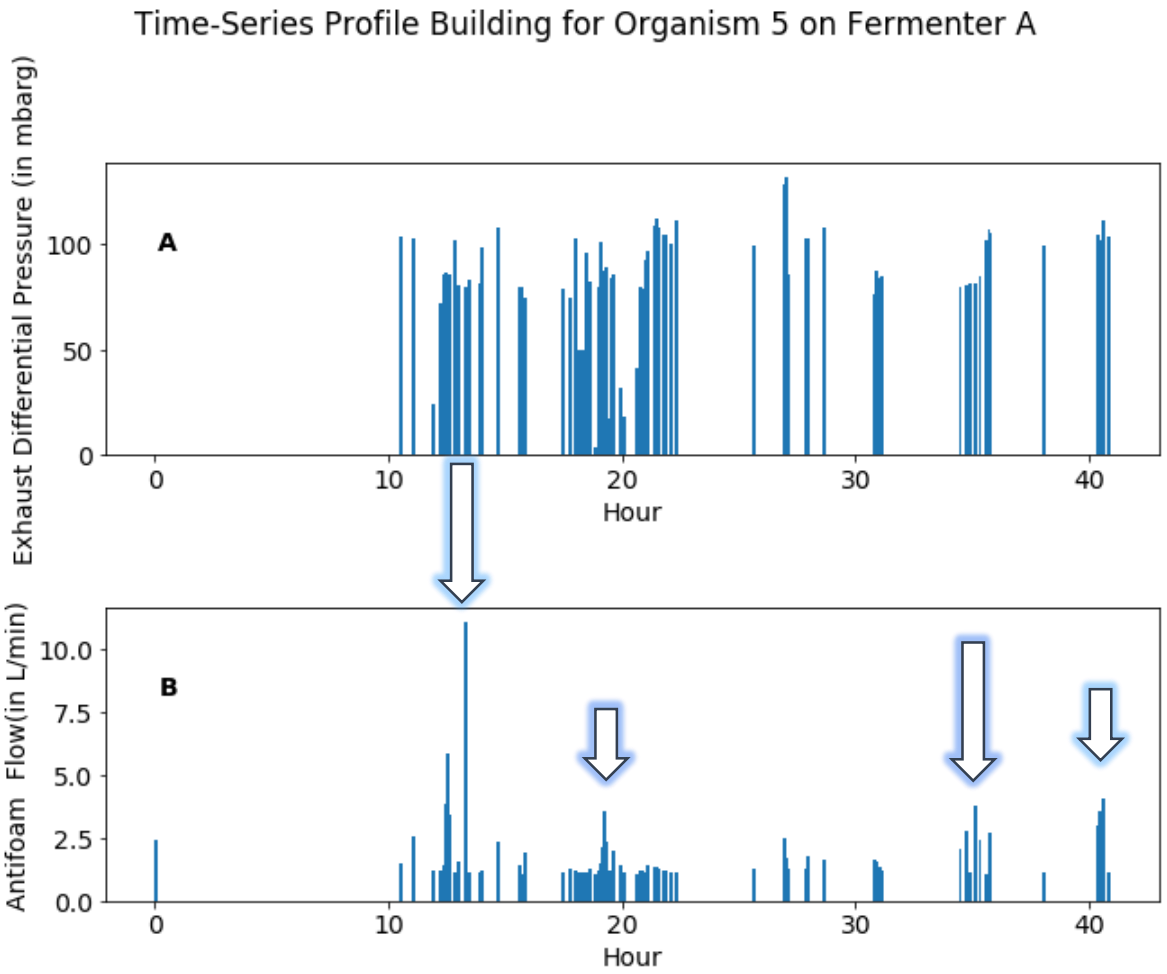


Figure 5.10. Time-Series Antifoam Profiling for Organism 5 on Fermenter A based on Maximum Antifoam Addition (Subplot B shows the top 100 antifoam additions and Subplot A shows the corresponding exhaust differential pressure during the given time).

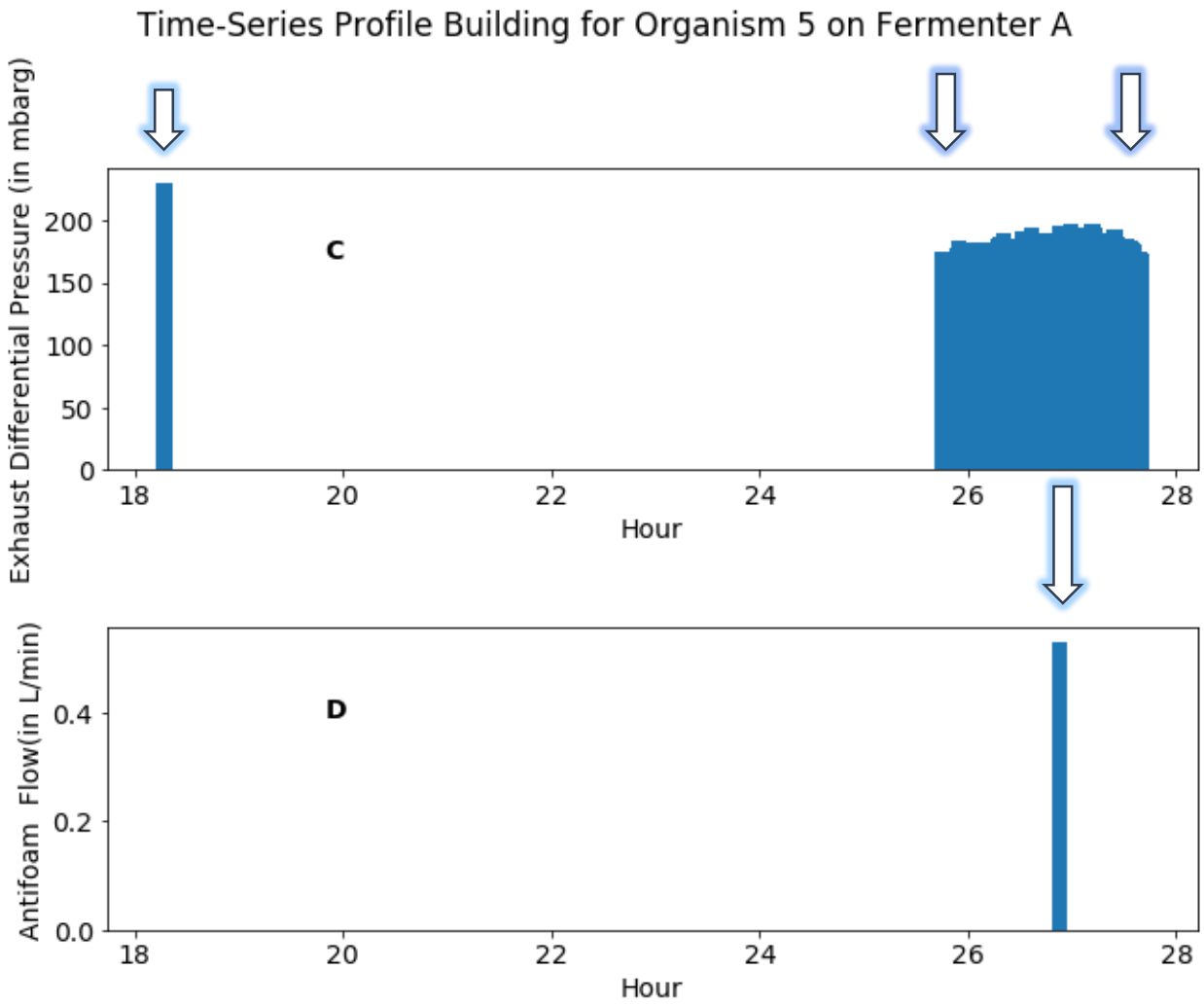


Figure 5.11. Time-Series Antifoam Profiling for Organism 5 on Fermenter A based on Maximum Exhaust Differential Pressure (Subplot A shows the top 100 exhaust differential pressure and Subplot B shows the corresponding antifoam addition during the given time).

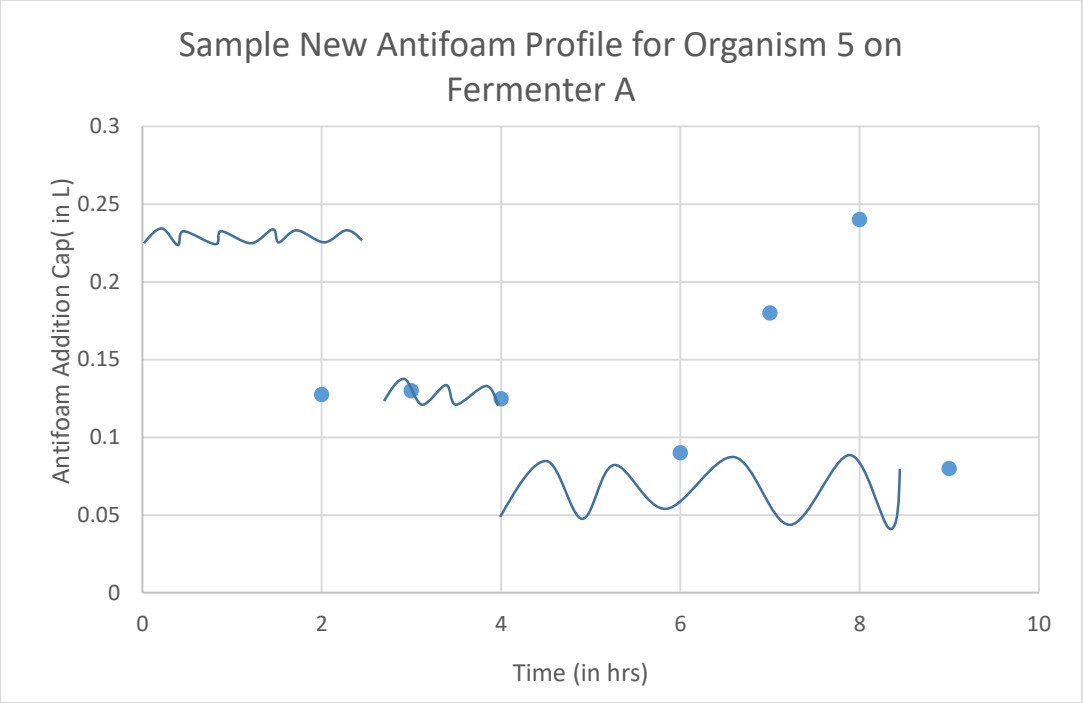


Figure 5.12. Sample deployment-ready capped profile.

5.5.3 Proof-of-Concept (PoC) Antifoam Profiles Deployment Verification Tests

We successfully deploy the antifoam profiles for an organism for three different fermenter types for about a month to gauge the initial results. We perform tests on two different scales: an integrated batch test and a single batch test. For the integrated batch test, we combine all the performance of all the batches from the training set and compare it with all the batches of the test set. For the single batch test, we take one random batch from the training batch and compare it side by side with a random batch from the test set.

We evaluate the performance based on three different criteria:

a) Average Exhaust Differential Pressure (AEDP)

For this criterion, we simply average the exhaust differential pressure (an indicator of foaming) throughout a batch or multiple batches, for a specific organism in a designated fermenter type and compare the averages for the training set and the test set. This measurement gives us some clues towards the extent of improvement in the new profiles. In an ideal setting, the AEDP should decrease with improvement in the model as we continue adding new data to the model.

b) EDP Threshold Cross Count (ETCC)

This criterion is based on the count of exhaust differential pressure measurements that exceed our threshold of 100 mbarg (based on industrial experience). In an ideal setting, the ETCC should decrease with improvement in the model.

c) Hourly Volume Retention (HVR)

This criterion is one of the most important performance indicators as it directly correlates with production yield. For this measurement, we use the following formula:

$$HVR = \frac{Initial\ Maximum\ Volume - Final\ Minimum\ Volume}{Batch\ Run\ Time}$$

The initial maximum volume refers to the maximum volume in the fermenter during the initial stages of fermentation. Similarly, the final minimum volume refers to the minimum volume in the fermenter during the final stages of the fermentation.

Lastly, the batch run time is the runtime of a single batch or the average runtime of multiple batches. We divide the difference by the batch run to normalize the criterion for direct comparison. In an ideal setting, the HVR should decrease with improvement in the model. A decrease in HVR is a firm indicator of an increase in product yield and process improvement.

5.5.4 Initial Performance Results from Deployed Antifoam Profiles

We deploy the profiles designed for a specific organism across 3 out of 4 different fermenter types. We monitor the results for several batches and use two different approaches to compare the initial results: integrated-batch comparison and single-batch comparison as shown in Figures 13 and 14.

For the single-batch approach, we compare an arbitrary batch from the training set with a random batch from the test set. We use the above-mentioned metrics to evaluate the performances of the new antifoam profiles on the fermenter types. From Table 5.4, we see that the profiles show significant improvement with a decrease in hourly volume retention (HVR) across all the fermenters. Fermenter A has a substantial overall reduction in EDP threshold cross count (ETCC) and average exhaust differential pressure (AEDP) as well. Fermenter C has conflicting results and requires further inspection.

For the integrated-batch approach, we combine all the current batch observations and compare them with the merged observations from the past batches. From Table 5.5, we see that the profiles have a large impact on Fermenter A. We reduce the AEDP by 55%, reduce ETCC by 12%, and decrease HVR in the fermenter by 53%. Similarly, we reduce the HVR in Fermenter B by 10%. We see a slight increase in AEDP, but that can be attributed to the lack of data for Fermenter B in both the test and the training dataset.

The results from Fermenter C are conflicting as even though we manage to decrease the HVR, we still see a large increase in the AEDP and significant ETCC overshoot. When we go back to inspect the training dataset, we notice that the training dataset included several negative values which were corrected to zero, resulting in a sparse dataset. Comparatively, the new dataset does not have such sparsity. After further inspection, we verify that the impeller for Fermenter C is inefficient due to mounting errors. This issue led to the pressure overshoot in the line pressure, resulting in many negative values for the exhaust differential pressure. It is possible to remodel the fermenter results by adding a buffer value for the overshoot, to get a more reasonable comparison for the AEDP and ETCC. However, since the HVR is a firmer measure for comparison, we can see that even for Fermenter C, our new antifoam profiles show significant improvement.

Table 5.4. Integrated Batch Test Results

Integrated Batch Test			
		Value	**Indicator**
	HVR	-53%	Very Good
Ferm A	**ETCC**	-12%	Good
	AEDP	-55%	Very Good
	HVR	-10%	Good
Ferm B	**ETCC**	0%	Neutral
	AEDP	5.65%	Bad
	HVR	-14%	Good
Ferm C	**ETCC**	8%	Bad
	AEDP	69%	Very Bad

Table 5.5. Single Batch Test Results

Single Batch Test			
		Value	**Indicator**
	HVR	-41%	Very Good
Ferm A	**ETCC**	-13%	Good
	AEDP	-42%	Very Good
	HVR	-34%	Very Good
Ferm B	**ETCC**	-4%	Good
	AEDP	-1%	Neutral
	HVR	-17%	Good
Ferm C	**ETCC**	12%	Bad
	AEDP	39%	Very Bad

Abbreviations:
HVR: Hourly Volume Retention
AEDP: Average Exhaust Differential Pressure
ETCC: EDP Threshold Cross Count
Note: Negative values are signs of improvement

Figure 5.13. Integrated-Batch test results.

Abbreviations:
HVR: Hourly Volume Retention
AEDP: Average Exhaust Differential Pressure
ETCC: EDP Threshold Cross Count
Note: Negative values are signs of improvement

Figure 5.14. Single-Batch test results.

5.6. Conclusion, Limitations, and Future Research

Herein, we showcase an industrial solution to foaming based on the development of data-driven antifoam profiles. We first predict the antifoam addition for defoaming directly, by using automated machine learning. Then, we use exploratory time-series to build macro-profiles for immediate deployment of antifoam profiles.

We verify the success of our profiles in both single-batch tests and integrated batch tests. Our profiles manage to reduce the average exhaust pressure, reduce the exhaust differential pressure overshoots, and most importantly reduce the hourly volume retention of the fermenter. By decreasing the hourly volume loss by over 53% and lowering the average pressure by over 55% in Fermenter A (integrated), we demonstrate our ability to control foaming with machine learning and data analytics. Decreasing the hourly volume loss leads to a significant increase in production yield and results in a corresponding revenue increase.

We have tested the results for a single strain across three different fermenter types. It would be interesting to gauge the performance of the profiles for the other five strain types and the fourth fermenter type. In Section 5.5.2, we mention the possibility of establishing micro-profiles for further improvement. We believe that ensemble methods, automated machine learning, and deep learning tools, can help with building those profiles. We show how one can predict the antifoam directly with these models; with the right amount of data, we may be able to constraint the antifoam addition to a limit and have accurate micro-profiles to further improve the process.

References

1. Vardar-Sukan, F. Foaming: Consequences, Prevention and Destruction. *Biotechnology Advances* **1998**, *16* (5-6), 913–948.
2. Richtering, W. In *Smart colloidal materials*; Springer: Berlin, 2006; p 102.
3. Langevin, D. Aqueous Foams and Foam Films Stabilised by Surfactants. Gravity-Free Studies. *Comptes Rendus Mécanique* **2017**, *345* (1), 47–55.
4. Junker, B. Foam and Its Mitigation in Fermentation Systems. *Biotechnology Progress* **2007**, *23* (4), 767–784.
5. St-Pierre Lemieux, G.; Groleau, D.; Proulx, P. Introduction on Foam and Its Impact in Bioreactors. *Canadian Journal of Biotechnology* **2019**, *3* (2), 143–157.
6. Varley, J.; Brown, A. K.; Boyd, J. W. R.; Dodd, P. W.; Gallagher, S. Dynamic Multi-Point Measurement of Foam Behaviour for a Continuous Fermentation over a Range of Key Process Variables. *Biochemical Engineering Journal* **2004**, *20* (1), 61–72.
7. Prud'homme, R. K. *Foams: Theory: Measurements: Applications*; Taylor and Francis: London, 2017

8. Velugula-Yellela, S. R.; Williams, A.; Trunfio, N.; Hsu, C.-J.; Chavez, B.; Yoon, S.; Agarabi, C. Impact of Media and Antifoam Selection on Monoclonal Antibody Production and Quality Using a High Throughput Micro-Bioreactor System. *Biotechnology Progress* **2017**, *34* (1), 262–270.
9. Nielsen, J. C.; Senne de Oliveira Lino, F.; Rasmussen, T. G.; Thykær, J.; Workman, C. T.; Basso, T. O. Industrial Antifoam Agents Impair Ethanol Fermentation and Induce Stress Responses in Yeast Cells. *Applied Microbiology and Biotechnology* **2017**, *101* (22), 8237–8248.
10. Chandran Suja, V.; Kar, A.; Cates, W.; Remmert, S. M.; Fuller, G. G. Foam Stability in Filtered Lubricants Containing Antifoams. *Journal of Colloid and Interface Science* **2020**, *567*, 1–9.
11. McClure, D. D.; Lamy, M.; Black, L.; Kavanagh, J. M.; Barton, G. W. An Experimental Investigation into the Behaviour of Antifoaming Agents. *Chemical Engineering Science* **2017**, *160*, 269–274.
12. Wu, H.; Dong, R.; Wu, S. Exploring Low-Cost Practical Antifoaming Strategies in the Ammonia Stripping Process of Anaerobic Digested Slurry. *Chemical Engineering Journal* **2018**, *344*, 228–235.
13. Jiang, J.; Zu, Y.; Li, X.; Meng, Q.; Long, X. Recent Progress towards Industrial Rhamnolipids Fermentation: Process Optimization and Foam Control. *Bioresource Technology* **2020**, *298*, 122394.
14. Agarwal, A.; Liu, Y. A.; McDowell, C. 110th Anniversary: Ensemble-Based Machine Learning for Industrial Fermenter Classification and Foaming Control. *Industrial & Engineering Chemistry Research* **2019**, *58* (36), 16719–16729.
15. Olson, R. S.; Urbanowicz, R. J.; Andrews, P. C.; Lavender, N. A.; Kidd, L. C.; Moore, J. H. Automating Biomedical Data Science through Tree-Based Pipeline Optimization. *Applications of Evolutionary Computation* **2016**, 123–137.
16. Schweidtmann, A. M.; Clayton, A. D.; Holmes, N.; Bradford, E.; Bourne, R. A.; Lapkin, A. A. Machine Learning Meets Continuous Flow Chemistry: Automated Optimization towards the Pareto Front of Multiple Objectives. *Chemical Engineering Journal* **2018**, *352*, 277–282.
17. Orlenko, A.; Moore, J. H.; Orzechowski, P.; Olson, R. S.; Cairns, J.; Caraballo, P. J.; Weinshilboum, R. M.; Wang, L.; Breitenstein, M. K. Considerations for Automated Machine Learning in Clinical Metabolic Profiling: Altered Homocysteine Plasma Concentration Associated with Metformin Exposure. *Biocomputing* **2018**.
18. Venkatasubramanian, V. The Promise of Artificial Intelligence in Chemical Engineering: Is It Here, Finally? *AIChE Journal* **2018**, *65* (2), 466–478.
19. Ge, Z.; Song, Z.; Ding, S. X.; Huang, B. Data Mining and Analytics in the Process Industry: The Role of Machine Learning. *IEEE Access* **2017**, *5*, 20590–20616.
20. Kim, G. B.; Kim, W. J.; Kim, H. U.; Lee, S. Y. Machine Learning Applications in Systems Metabolic Engineering. *Current Opinion in Biotechnology* **2020**, *64*, 1–9.
21. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* **2011**, *12*, 2825.
22. McKinney, W. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference,* Austin, TX, June, **2010.**
23. Hunter, J. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering* **2007**, *9*, 90.

24. Lundberg, S. M.; Lee, S.-I. A Unified Approach to Interpreting Model Predictions. *31st Conference on Neural Information Processing Systems* **2017**, 4768–4777.
25. Nielsen, P. H.; Oxenbøll, K. M.; Wenzel, H. Cradle-to-Gate Environmental Assessment of Enzyme Products Produced Industrially in Denmark by Novozymes A/s. *The International Journal of Life Cycle Assessment* **2006**, *12* (6), 432–438.
26. Meyer, H.-P.; Minas, W.; Schmidhalter, D. Industrial-Scale Fermentation. *Industrial Biotechnology* **2016**, 1–53.
27. Demirci, A.; Pometto, A. L. Repeated-Batch Fermentation in Biofilm Reactors with Plastic-Composite Supports for Lactic Acid Production. *Applied Microbiology and Biotechnology* **1995**, *43* (4), 585–589.

Chapter 6. Deep Learning Optimization on Industrial and Simulated Polyolefin Datasets

6.1 Introduction

This chapter presents the workflow to build a deep neural network (DNN) and three different types of recurrent neural network (RNN) using Python. We showcase the use of these two methodologies for two different cases. For the first case, we demonstrate how to build a steady-state (time-independent) model for a high-density polyethylene (HDPE) industrial plant to predict the melt index (MI) using Keras libraries to build deep neural networks. And for the second case, we cover how to build three different types of dynamic recurrent neural network from scratch to predict the melt index from a simulated time-dependent polymer dataset.

Section 6.1 explains the resurgence of deep neural networks and describes the basics of a deep neural network. Section 6.2 provides the details about the different types of recurrent neural networks. Section 6.3 describes an industrial HDPE plant setup and the variables associated with the industrial dataset. Section 6.4 introduces a step-by-step workflow to build a deep neural network using Keras libraries. Section 6.5 describes the dynamic dataset simulated for an HDPE plant. Section 6.6 illustrates how to build the three different types of recurrent neural network architecture to deal with time-sensitive data in a multistep workflow.

There are several published studies which use different deep neural network architectures and recurrent neural networks for plant control and optimization. [1,2,10,11,14] Deep learning is used for soft sensor development [1], fault diagnosis [2], predicting chemical properties [3,4,13], designing polymers [8,11,12], predicting protein interactions [5], tracking particle dynamics [7], fouling in heat exchangers [19], etc. However, the implementation of these architectures can be quite challenging for new readers.

Through this chapter, we want to enable readers to create their own deep neural network and recurrent neural network for optimization with little to no neural network exposure, minimal computing resources, and easy implementation.

6.1.1 Why use deep neural networks?

In the past couple of years, we have seen a surge in big data collection, cutting-edge hardware improvements, and release of several user-friendly software platforms to enable engineers to use deep neural networks for their research. In addition to accessibility to resources for building deep neural networks, many industries are promoting process engineering solutions based on digital transformation. One of the examples of such a transformation is application of a ‘digital twin’. A digital twin is a virtual representation of a process/model which becomes more powerful as it keeps receiving real world data from its real-world counterpart.

Before we talk about deep neural networks, we need to briefly talk about neural networks. Neural networks are a cluster of nodes, known as neurons, which are arranged into an ordered sequence of at least three groups known as layers. The first layer is called the input layer and has the same number of neurons as the number of input variables for the system, the last layer is known as the output layer; the layers in between are called hidden layers. The choice of number of layers and number of nodes is task-dependent and is mostly based on experience. [18] Deep neural networks are just neural networks with two or more hidden layers. Appendix B gives a brief introduction to different neural network terminologies and mathematics behind neural networks.

Most steady-state (time-independent) data can be modelled using a simple feed-forward neural network, where the connections between the nodes do not form a cycle. However, for dynamic (time-dependent) data, a more complex neural network architecture is more useful. Recurrent Neural Networks (RNN) are very useful in handling such time-dependent sequence data. In a recurrent neural network, outputs from previous time steps are taken as inputs for the current time step.

6.2 Different Types of Recurrent Neural Network

Recurrent Neural Networks (RNN) are mostly used to deal with sequential data types like time-series data. RNNs are trained by backpropagation which is a method of fine-tuning the weights of a neural network.

In this chapter, we will be looking into three different types of RNN:

6.2.1 Long short-term memory (LSTM)

In a RNN architecture, continuously updating the weights while training sometimes leads to decreasing gradient, making it harder for a network to converge with proper training. Inversely, you can also have increased gradient, which also makes it harder for a network to converge. To deal with these vanishing gradient problems and exploding gradient problems, Hochreiter et al. proposed LSTM which is a special kind of RNN with memory units to overcome long-term dependencies among the data. [20]

An LSTM layer consists of a set of connected memory cells which pass the sequential information. Each of these memory cells has three information gates: input gate, forget gate, and the output gate as shown in Figure 6.1 and two states: cell state and hidden state. These help the memory cell control what information flows in, what to forget, and what to memorize across the time series to learn long-term dependencies. The memory cell has two activation functions: Sigmoid (maintains range between 0 and 1) and Hyperbolic Tangent (maintains range between -1 and 1). The sigmoid activation function helps the gates to update or forget the values, while hyperbolic tangent helps regulate the network. The attached Appendix B gives more details about these activation functions.

In an LSTM memory cell, the first step involves combining the previous hidden state and current input at the forget gate, where the combination is passed through the sigmoid function. Then, the combination is passed through the input gate, where it gets transformed by both sigmoid and hyperbolic tangent functions. The hyperbolic tangent output is multiplied with the sigmoid output to get a primary cell state. The primary cell state is then multiplied by the forget vector from the forget gate and added pointwise to get a new cell state. For the final output gate, we first send the combination from the input gate through a sigmoid function and then send the new cell state through a hyperbolic tangent function. The output (or the hidden state) is finally calculated by multiplying the outputs from the output gate operations.

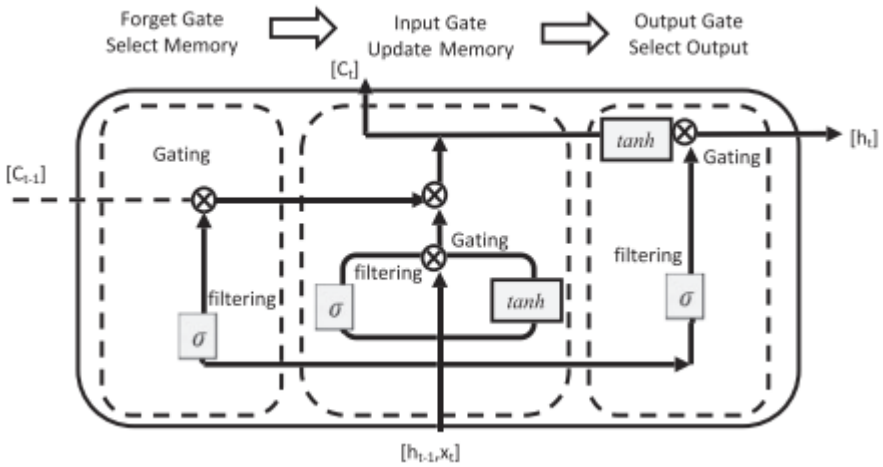


Figure 6.1 LSTM memory cell. Reproduced from Reference 19.

6.2.2 Bidirectional LSTM

Bidirectional LSTM (BiLSTM) is a modification to LSTM, that consists of two LSTMs: one taking the input in a forward direction, and the other in a backward direction. BiLSTM increases the amount of information available to the network, by combining the outputs from both forward and backward directions to make its final prediction.

6.2.3 Gated recurrent unit (GRU)

A variant of LSTM architecture, GRU proposed by Cho et al. is a simplified version of LSTM. [21] Each memory cell for a GRU consists of only two gates: reset gate and update gate and has only one state: hidden state. The update gate acts like the forget and input gate of LSTM as it decides which information to forget and what new information to add or update. The reset gate acts as a type of forget gate as it decides how much past information to forget. Like LSTM, the current input (x_t) and previous hidden state (h_{t-1}) are combined as they go through the reset gate and the update gate to give an output (h_t).

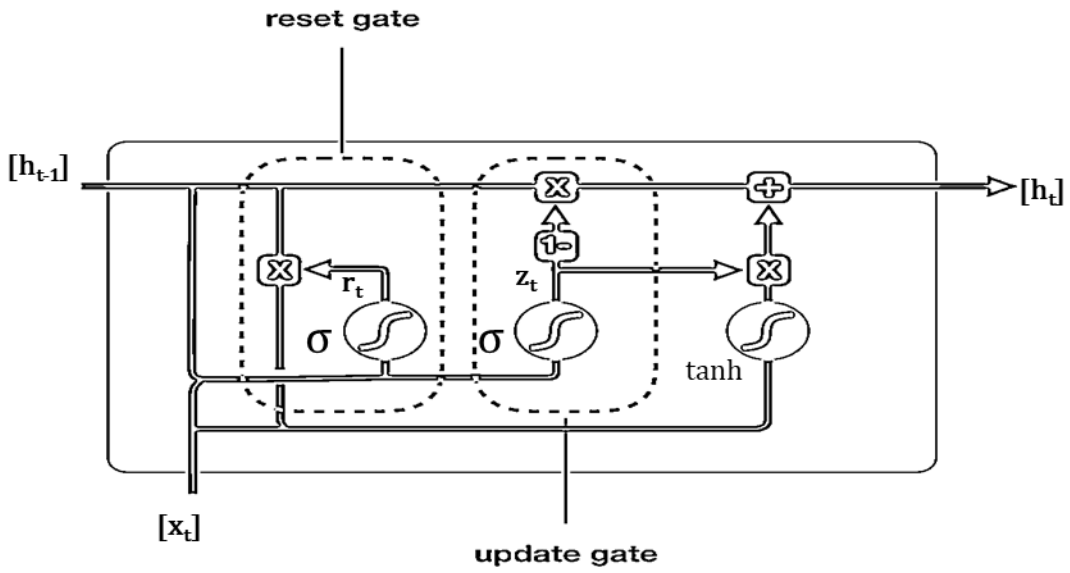


Figure 6.2 GRU memory cell. Adapted from reference 22.

6.3 Industrial HDPE Reactor Setup

Figure 6.3 illustrates a slurry process to produce a high-density polyethylene (HDPE). The polymerization process involves two different reactors, and the entire process is highly exothermic. Almost the entire reactor volume is occupied by the polyethylene slurry, which undergoes separation, removing any unreacted monomer, solvent, catalyst, and other reactants from the polymer. Then, the slurry is cooled using internal coils and external cooling systems and put through a series of molds and packaged.

The 2005-2006 real-industrial HDPE plant dataset is provided by Park et. al. [25] From the raw process data, we choose the 14 input variables to predict the output variable, the melt index (MI) as shown in Table 6.1.

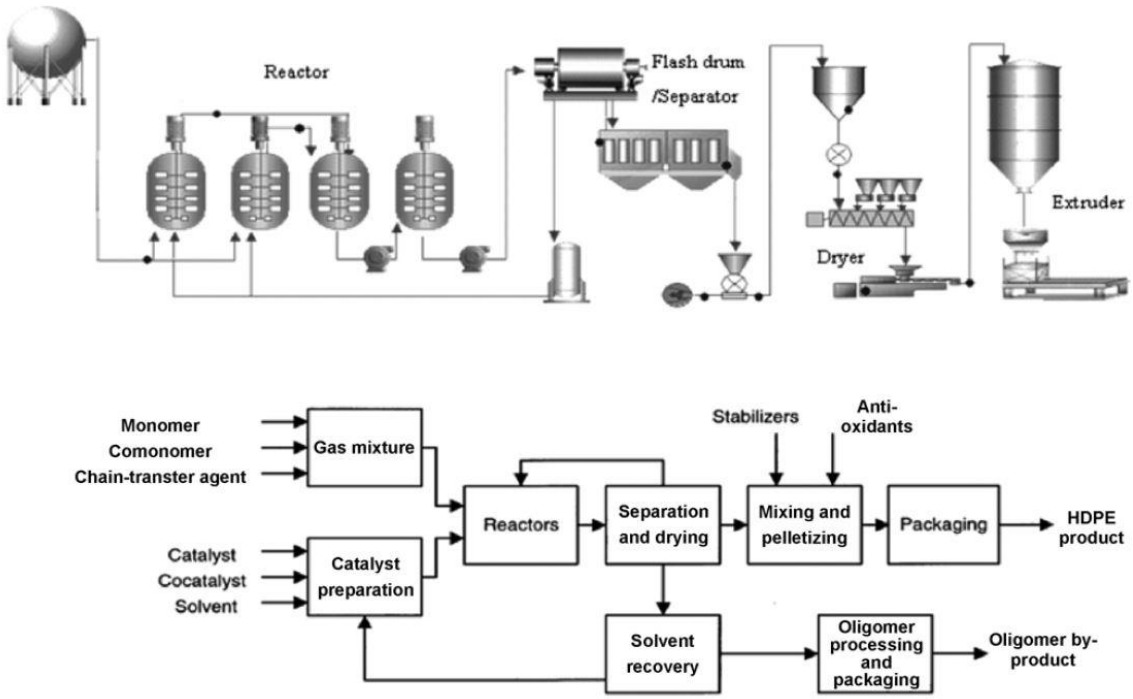


Figure 6.3 Schematic of the HDPE reactor used to build a soft sensor. Reproduced from reference (23).

Table 6.1. List of all the input variables for HDPE reactor.

Input Variables	**Symbol**	**Input Variables**	**Symbol**
Ethylene Feed Rate	ETH	Reactor temperature (Top)	RT_TOP
Hexane Feed Rate	HDH	Reactor temperature (Middle)	RT_MIDDLE
Recycled Hexane Feed Rate	HMH	Reactor level	LEVEL
Reactant (BUE ligand) Feed Rate	PRL/BUE	Reactor Pressure_1	RP1
Hydrogen Feed Rate	HYD	Reactor Pressure_2	RP2
Reactor temperature (Bottom)	RT_BOTTOM	Jacket Temperature_1	JT1
Agitator Speed	AGITATOR	Jacket Temperaure_2	JT2

6.4 Deep Neural Network Model Using Keras Libraries

Keras is a deep learning application programming interface (API) written in Python, running on top of the machine learning libraries like TensorFlow and Theano. For our first case model, we use Keras to build a deep neural network in a google colab notebook.

We follow these steps to build a deep neural network for melt index prediction:

1. Remove any observation with empty or missing values. Then, divide the dataset into training and test datasets. We will use 80-20 split (80% training data and 20% test data). It is important to note that the training data can be further divided into validation set (70% training data and 10% validation data) for hyperparameter tuning, if needed.
2. Select the dependent and independent variables of the dataset.
3. Build the deep-neural network using Keras library.
4. Extract the value of the model performance metric.
5. Plot a scatterplot to visualize the model predictions.

The first step for every model building is data preprocessing. In this step, we make sure that we divide our dataset into two randomly sampled subsets using an 8:2 ratio. We also make sure that all null values are removed from the dataset. Figure 6.4 shows the code for splitting our dataset and removing all observations with missing values. Here 'Axis=0' represents the row values and by using 'how=any' we remove all rows with any missing column values. For this example, we have two different datasets, HDPE1 and HDPE2 representing two different timelines for the HDPE plant.

```
import pandas as pd
df1 = pd.read_excel(current_path+'/HDPE1.xlsx')
df2 = pd.read_excel(current_path+'/HDPE2.xlsx')
df1=df1.dropna(axis=0, how='any')
df2=df2.dropna(axis=0, how='any')
train1=df1.sample(frac=0.8,random_state=200)
test1=df1.drop(train1.index)
train2=df2.sample(frac=0.8,random_state=200)
test2=df2.drop(train2.index)
train1.to_excel('HDPEtrain1.xlsx',index=False)
test1.to_excel('HDPEtest1.xlsx',index=False)
train2.to_excel('HDPEtrain2.xlsx',index=False)
test2.to_excel('HDPEtest2.xlsx',index=False)
```

Figure 6.4 Splitting the HDPE dataset into training and test sets and removing null values.

The second step involves selecting our X-values (independent variables) and Y-value (dependent variable). Figure 6.5 illustrates how we do that. We use the HDPE1 dataset for our model. We select the first 14 columns as our X-values and the last column as our Y-value.

```
traindataset=pd.read_excel('HDPEtrain1.xlsx')
testdataset=pd.read_excel('HDPEtest1.xlsx')
X_train1=traindataset.iloc[:,0:14].values
y_train=traindataset.iloc[:,14:15].values
X_test1=testdataset.iloc[:,0:14].values
y_test=testdataset.iloc[:,14:15].values
```

Figure 6.5 Selecting X-values and Y-value

The third step involves building the deep neural network model using Keras. Figure 6.6 shows the network architecture for the model. As we can see from the structure, the deep neural network has 1 input layer with 14 neurons, 2 hidden layers with 10 neurons, and an output layer with 1 neuron. We use rectified linear unit (ReLU) activation function, as the activation function for the hidden layers, which is one of the most used activation functions in deep learning models. We use simple linear activation function for the output layer as we want a numerical regression output. We use 'Adam optimization' for training the neural net. Appendix B describes activation functions and different optimization techniques.

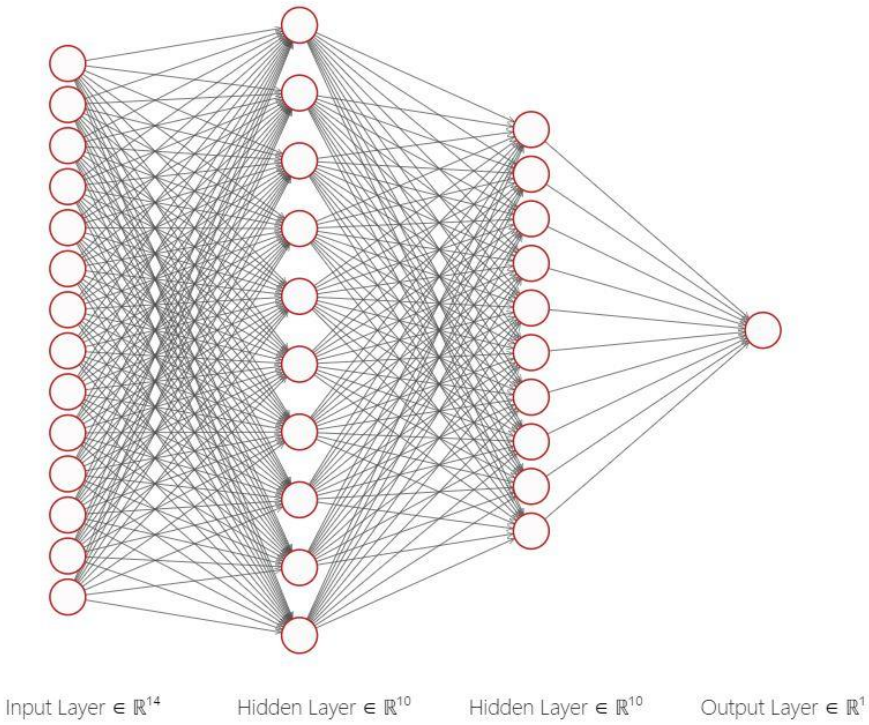


Figure 6.6 Deep neural network with two hidden layers.

We use sequential Keras model to build the layers one at a time. For more complex architectures we can use functional Keras models. For layer type, we use dense layer type such that each neuron in the dense layer receives input from all the neurons of its previous layer. Using the model, we minimize the mean squared error and train 481 parameters as shown in Figure 6.7.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
# define base model
def dnn_model():
  # create model
  model = Sequential()
  model.add(Dense(14, input_dim=14, kernel_initializer='normal', activation='relu'))
  model.add(Dense(10, kernel_initializer='normal', activation='relu'))
  model.add(Dense(10, kernel_initializer='normal', activation='relu'))
  model.add(Dense(1, kernel_initializer='normal',activation='linear'))
  # Compile model
  model.compile(loss='mean_squared_error', optimizer='adam')
  return model
# evaluate model
dnn_model().summary()
estimator = KerasRegressor(build_fn=dnn_model, epochs=100, batch_size=5, verbose=0)
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 14)                210
_________________________________________________________________
dense_17 (Dense)             (None, 10)                150
_________________________________________________________________
dense_18 (Dense)             (None, 10)                110
_________________________________________________________________
dense_19 (Dense)             (None, 1)                 11
=================================================================
Total params: 481
Trainable params: 481
Non-trainable params: 0
_________________________________________________________________
```

Figure 6.7 Deep neural network using Keras.

In the fourth step, we extract the evaluation metric, root mean squared error (RMSE) value, for the model. As the name suggests, RMSE measures the standard deviation of the residuals. It is used to see how spread the predicted values are. Figure 6.8 shows the code for extracting the RMSE value from the model. We can see that the RMSE value for melt index is 1.2606.

```
estimator.fit(X_train1,y_train)
ypred= estimator.predict(X_test1)
rmse_nn = np.sqrt(mean_squared_error(y_test,ypred))
print('Root Mean Squared Error:', round(rmse_nn,4))
```

```
Root Mean Squared Error: 1.2606
```

Figure 6.8 Evaluation metric calculation (RMSE calculation).

The fifth and final step involves visualizing the actual and predicted values. Figure 6.9 explains how we can visualize the residuals with a scatterplot. From the figure, we can see that our simple deep neural network model does a good job of predicting the melt index.

```
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
fig, ax = plt.subplots()
ax.scatter(x = range(0, y_test.size), y=y_test, c = 'blue', label = 'Actual', alpha = 0.5,s=4)
ax.scatter(x = range(0, ypred.size), y=ypred, c = 'red', label = 'Predicted', alpha = 0.5,s=4)
plt.legend()
plt.title('Prediction Error for Neural Networks')
ax.set_xlabel('Observations')
ax.set_ylabel('Melt-Index Value')
plt.show()
```

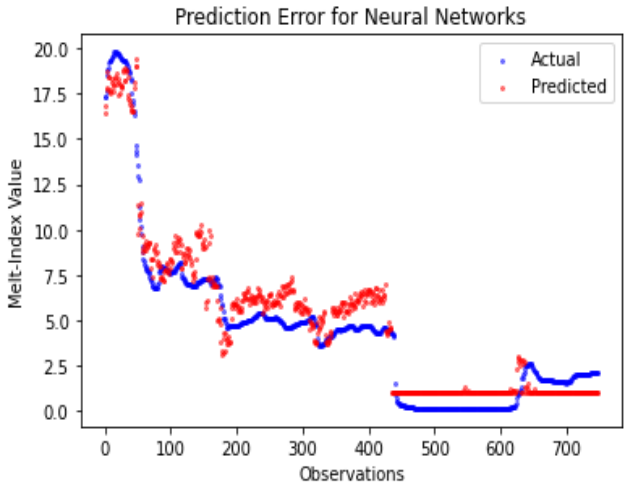


Figure 6.9 Visualization of actual vs predicted plot for melt-index prediction.

6.5 Time-Dependent Polymer Dataset Obtained by Simulation for an HDPE Plant

For the second case, we use data from an HDPE plant simulation and predict the melt index using dynamic deep neural network modeling techniques.

We simulate a two-reactor slurry HDPE process with ethylene as the monomer and butylene as the comonomer. Hexane is used as the solvent for the process. We build a dynamic process model like the procedure mentioned by Sharma and Liu. [26] The dynamic model is used to make changes in the process variables at set time intervals to simulate a real-time dynamic HDPE plant. We run the simulations twice with different changes in the process variables at the same time periods to simulate two different datasets for training and testing.

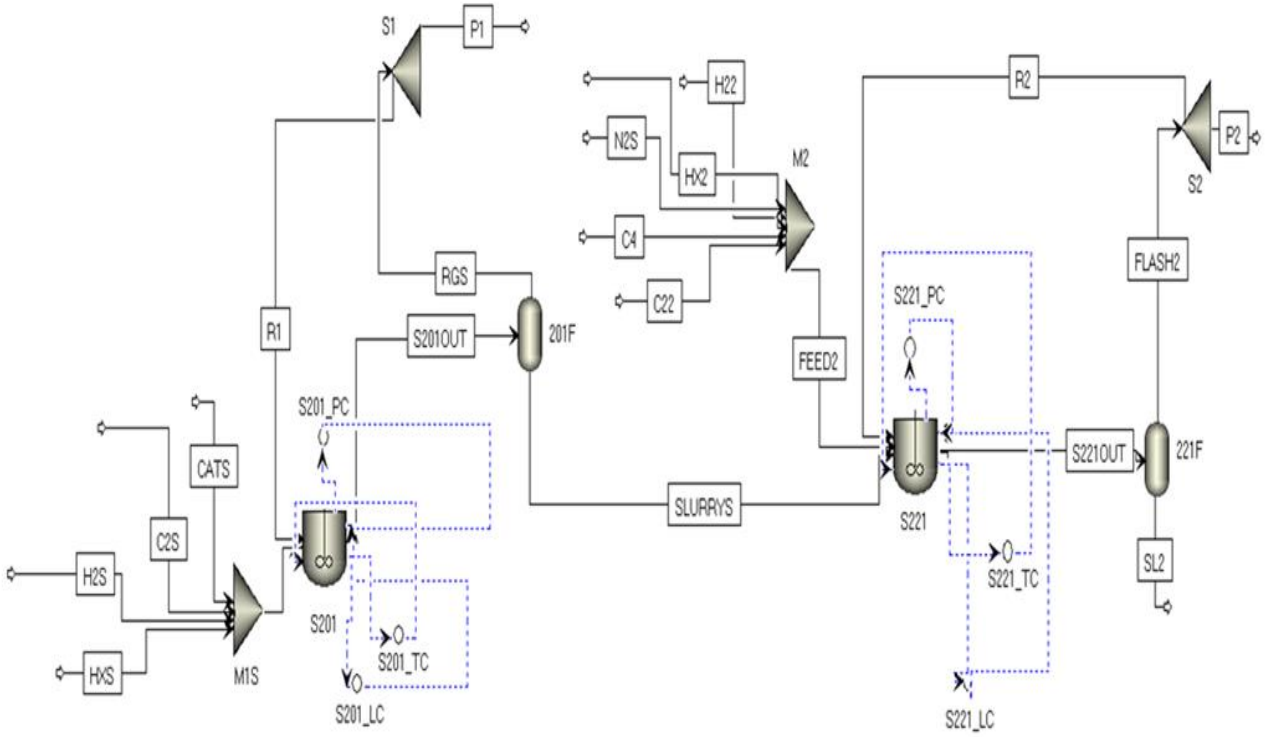


Figure 6.10 Dynamic HDPE production process with 11 process variables.

From the raw process data, we use 8 input variables to predict the melt index (MI) as the output variable as shown in Table 6.2.

Table 6.2. List of all the input variables for simulated HDPE reactor.

Input Variables	Symbol	Input Variables	Symbol
Hexane Solvent Feed to 1st reactor	HX	Ethylene Feed to 2nd reactor	C22
Hydrogen Feed to 1st reactor	H2	Butylene feed to 2nd reactor	C4
Ethylene Feed to 1st reactor	C2	Hexane Solvent feed to 2nd reactor	HX2
Catalyst Feed to 1st reactor	CAT	Hydrogen Feed to 2nd reactor	H22

6.6 Dynamic Deep Learning Using 3 Types of Recurrent Neural Networks

In this section, we look at building three different types of deep recurrent neural network structures to extract the melt index for a dynamic process.

We use the following steps as a step-by-step workflow to build the network architectures. We add some additional steps compared to Section 6.4 like data transformation to demonstrate a more complex workflow.

1) Remove any observation with empty or missing values. Then, divide the dataset into training and test datasets. We will use 80-20 split (80% training data and 20% test data).

2) Select the dependent and independent variables of the dataset.

3) Transform the variables by standardizing or normalizing the variables.

4) Build and train the three different types of recurrent neural network.

5) Make predictions and visualize the predictions for all the architectures.

6) Extract the value of the prediction metric for model evaluation.

Our first step like before is data preprocessing, where we check for any missing values and then split the dataset into training and test datasets. Figure 6.11 shows the code for checking the missing values and interpolating the values if missing. In our case there are no missing values. There are other methods

of dealing with missing values like listwise deletion, pairwise deletion, mean substitution, maximum likelihood, multiple imputation, etc. [27]

```
#Import the required data from excel files from their respective file locations.
import pandas as pd
df1= pd.read_excel(current_path+'/HDPE_trainingdata1.xlsx')
df2= pd.read_excel(current_path+'/HDPE_testdata1.xlsx')

#Check for missing values.
df1.isnull().sum()
df2.isnull().sum()

#Replace missing values with interpolation (if needed).
def replace_missing (attribute):
    return attribute.interpolate(inplace=True)
replace_missing(df1['CAT1'])
replace_missing(df1['C21'])
replace_missing(df1['H21'])
replace_missing(df1['HX1'])
replace_missing(df1['C22'])
replace_missing(df1['C42'])
replace_missing(df1['HX2'])
replace_missing(df1['H22'])
replace_missing(df1['Melt_index'])
```

Figure 6.11 Checking for missing values and interpolation of missing values.

In order to split the dataset into training and test datasets, we use a different approach. Instead of just directly splitting the dataset using an 8:2 train-test split, we use a time-split validation method. [16] Our dataset consists of two separate batch runs and we use one of the runs to train the model and test the model on some instances (where we see the maximum variation in the melt index) from the second run. By doing this, we mimic a real industrial setting where we must predict how the next batch will look like based on the previous batch. Figure 6.12 shows the code for the splitting method along with their dimensions, code for the split plot, and Figure 6.13 helps us visualize the training and test data used for model building.

```
# Split train data and test data

import matplotlib.pyplot as plt

train_dataset1=df1
test_size= int(len(df2)*0.50)
test_trial=df2.iloc[test_size:]
test_size1=int(len(test_trial)*0.9)
test_dataset1=test_trial.iloc[:test_size1]
# Plot train and test data
plt.figure(figsize = (10, 6))
plt.plot(train_dataset1['Melt_index'])
plt.plot(test_dataset1['Melt_index'])
plt.xlabel('Observation')
plt.ylabel('Melt_index')
plt.legend(['Train set', 'Test set'], loc='upper right')
print('Dimension of train data: ',train_dataset1.shape)
print('Dimension of test data: ', test_dataset1.shape)
```

```
Dimension of train data:  (2112, 10)
Dimension of test data:  (545, 10)
```

Figure 6.12 Time-split validation method

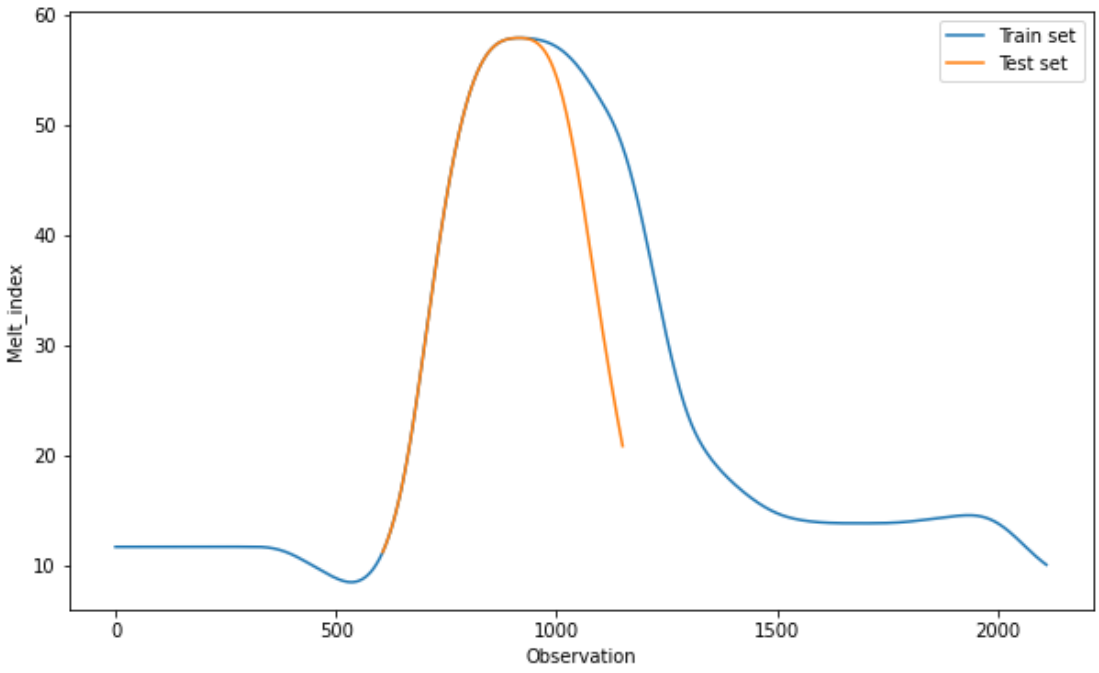


Figure 6.13 Train-test split plot using time-split.

The second step is selecting the independent and dependent variables like we did for the simple deep neural network. From Figure 6.14, we see that we choose the 8 independent variables other than melt index and time as our X variables and melt index as our Y variable (dependent).

```
# Split train data to X and y
X_train = train_dataset1.drop(['Melt_index','Time'], axis = 1)

y_train = train_dataset1.loc[:,['Melt_index']]
# Split test data to X and y
X_test = test_dataset1.drop(['Melt_index','Time'], axis = 1)
y_test = test_dataset1.loc[:,['Melt_index']]
```

Figure 6.14 Selecting independent and dependent variables.

The third step involves using data transformation techniques like feature scaling. The two most common methods for feature scaling include standardization (ensures zero mean and unit variance) and min-max normalization (re-scales features with a distribution value between 0 to 1). Feature scaling helps gradient-based algorithms like deep neural networks to converge faster. Standardization is preferred over min-max normalization in some cases as it is robust to outliers. We will use standardization for our transformation as shown in Figure 6.15. If we wanted to use min-max normalization, we would import and use MinMaxScaler instead of StandardScaler.

```
# Different scaler for input and output
from sklearn.preprocessing import StandardScaler
scaler_x = StandardScaler()
scaler_y = StandardScaler()
# Fit the scaler using available training data
input_scaler = scaler_x.fit(X_train)
output_scaler = scaler_y.fit(y_train)
# Apply the scaler to training data
train_y_norm = output_scaler.transform(y_train)
train_x_norm = input_scaler.transform(X_train)
# Apply the scaler to test data
test_y_norm = output_scaler.transform(y_test)
test_x_norm = input_scaler.transform(X_test)
```

Figure 6.15 Feature scaling using standardization or z-score method.

For the fourth step, we will be building and training three different RNN architectures: Long Short-Term Memory (LSTM), Bidirectional LSTM (BiLSTM), and Gated Recurrent Unit (GRU).

Before we start building the models, it is important to reshape the input data into a 3-D format. Figure 6.16 shows how we can do that:

```
import numpy as np
# Create a 3D input
def create_dataset (X, y, time_steps = 1):
    Xs, ys = [], []
    for i in range(len(X)-time_steps):
        v = X[i:i+time_steps, :]
        Xs.append(v)
        ys.append(y[i+time_steps])
    return np.array(Xs), np.array(ys)
TIME_STEPS = 1
X_test, y_test = create_dataset(test_x_norm, test_y_norm,
                                TIME_STEPS)
X_train, y_train = create_dataset(train_x_norm, train_y_norm,
                                  TIME_STEPS)
print('X_train.shape: ', X_train.shape)
print('y_train.shape: ', y_train.shape)
print('X_test.shape: ', X_test.shape)
print('y_test.shape: ', y_test.shape)
```

```
X_train.shape:  (2111, 1, 8)
y_train.shape:  (2111, 1)
X_test.shape:  (544, 1, 8)
y_test.shape:  (544, 1)
```

Figure 6.16 Reshaping input data.

Time-steps in Figure 6.16 refer to the number of instances to consider for input per iteration before giving an output. For example, if our dataset does not show any significant variation in output for every 30 instances, we can set the timestep to be 30. In our case, we set the timestep as 1.

Now we are ready to build the model as we build three different RNN architectures as shown in Figure 6.17.

```
import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import optimizers as optimizers

from tensorflow.keras import Sequential, layers, callbacks

from tensorflow.keras.layers import Dense, LSTM, Dropout, GRU, Bidirectional

# Create BiLSTM model
def create_model_bilstm(units):
    model = Sequential()
    model.add(Bidirectional(LSTM(8,
              return_sequences=True,recurrent_dropout=0.1),
              input_shape=(X_train.shape[1], X_train.shape[2])))
    model.add(Dropout(0.2))
    #first hidden layer
    model.add(Bidirectional(LSTM(units = units,return_sequences=True)))
    model.add(Dropout(0.2))
    #second hidden layer
    model.add(Bidirectional(LSTM(units = units)))
    model.add(Dropout(0.2))
    model.add(Dense(1))
    #Compile model
    model.compile(loss='mse', optimizer=optimizers.Adam(1e-4))
    return model
# Create LSTM or GRU model
def create_model(units, m):
    model = Sequential()
    model.add(m (8, return_sequences = True,
                input_shape = [X_train.shape[1], X_train.shape[2]]))
    model.add(Dropout(0.2))
     #first hidden layer
    model.add(m (units = units, return_sequences=True))
    model.add(Dropout(0.2))
    #second hidden layer
    model.add(m (units = units))
    model.add(Dropout(0.2))
    model.add(Dense(units = 1))
    optimizer = keras.optimizers.Adam(learning_rate=1e-4)
    #Compile model
    model.compile(loss='mse', optimizer=optimizer)
    return model
# BiLSTM
model_bilstm = create_model_bilstm(5)
# GRU and LSTM
model_gru = create_model(5, GRU)
model_lstm = create_model(5, LSTM)
```

Figure 6.17 Creating different RNN architectures

The first step towards creating the RNN architectures using Python is to import all the essential libraries. Here we use Keras API of the Tensorflow library to build the structures. We first create the BiLSTM model with two hidden layers. We use the dropout regularization method to ensure that our models are not overfitting. Since we have 8 input variables, we use 8 as the number of input neurons. We select Adam optimizer and a learning rate of 0.0001. The attached Appendix B gives information about the optimizer. To select the number of hidden neurons in each hidden layers (5 in our case as shown in Figure 6.17), we use a method developed by Sheela et al.[17], using the following equation:

$$N_h = (4n^2 + 3)/(n^2 - 8) \quad (6.1)$$

Here, n is the number of input neurons. There are several rules which can be followed to determine the number of hidden neurons depending on the nature of the dataset, complexity of activation functions, and the size of the dataset. [23-24] It is important to investigate structured trial and error method for every specific case.

We can also check the number of parameters involved in each of the architectures as shown in Figure 6.18. We can see that the BiLSTM model is trained on 2619 parameters. Similarly, LSTM is trained on 1050 and GRU is trained on 843 parameters respectively. It intuitively makes sense, as BiLSTM accounts for input in both forward and backward direction and GRU is a simplified version of LSTM.

```
model_bilstm.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bidirectional (Bidirectional (None, 1, 16)             1088
_________________________________________________________________
dropout (Dropout)            (None, 1, 16)             0
_________________________________________________________________
bidirectional_1 (Bidirection (None, 1, 10)             880
_________________________________________________________________
dropout_1 (Dropout)          (None, 1, 10)             0
_________________________________________________________________
bidirectional_2 (Bidirection (None, 10)                640
_________________________________________________________________
dropout_2 (Dropout)          (None, 10)                0
_________________________________________________________________
dense (Dense)                (None, 1)                 11
=================================================================
Total params: 2,619
Trainable params: 2,619
Non-trainable params: 0
_________________________________________________________________
```

Figure 6.18 Parameters involved in training BiLSTM.

Next, we train the network architectures as shown in Figure 6.19. We use 20% of the training data for validation. We set the number of epochs to 100. The number of epochs define the number of times the learning algorithm will work through the entire dataset. In order to avoid overfitting, we also set an early stop to stop training when validation loss has not improved after 30 epochs.

We set the batch size to 5. The batch size determines how many times the network weights are updated per epoch. Batch sizes and number of epochs are also based on the nature and size of the dataset; structured trial and error should be used to determine the best batch size. We can use grid search to determine the best number of epochs and batch size.

```
# Fit BiLSTM, LSTM and GRU
def fit_model(model):
    early_stop = keras.callbacks.EarlyStopping(monitor = 'val_loss',
                                               patience = 30)
    history = model.fit(X_train, y_train, epochs = 100,
                        validation_split = 0.2, batch_size = 5,
                        shuffle = False, callbacks = [early_stop])
    return history
history_bilstm = fit_model(model_bilstm)
history_lstm = fit_model(model_lstm)
history_gru = fit_model(model_gru)
```

Figure 6.19 Fitting the RNN models.

We can also plot the training and validation loss to visualize our losses through each epoch. Figure 6.20 shows the required code and the plot for training and validation loss in BiLSTM.

```
# Plot train loss and validation loss
def plot_loss (history):
    plt.figure(figsize = (10, 6))
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.ylabel('Loss')
    plt.xlabel('epoch')
    plt.legend(['Train loss', 'Validation loss'], loc='upper right')
plot_loss (history_bilstm)
plot_loss (history_lstm)
plot_loss (history_gru)
```

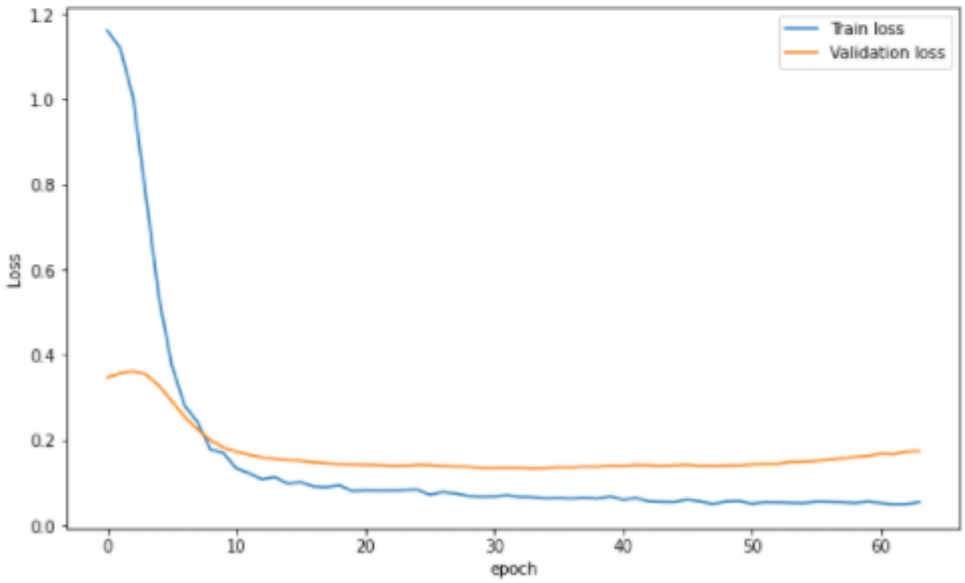


Figure 6.20 Training and validation loss in BiLSTM.

Now we are done with our fourth step; before we move on to the fifth step of comparing the actual values with the prediction, we must convert the target dependent variable (Y-variable/melt index) back to the original data space. We show the transformation in Figure 6.21.

```
y_test = scaler_y.inverse_transform(y_test)
y_train = scaler_y.inverse_transform(y_train)
```

Figure 6.21 Inverse transform of target variable.

Now for the fifth step, we want to make predictions and visualize the real target variable and the predictions for our three RNN architectures. Figure 6.22 (a-d) shows the steps to make the predictions and plot the prediction vs real values for the melt index.

```
# Make prediction
def prediction(model):
    prediction = model.predict(X_test)
    prediction = scaler_y.inverse_transform(prediction)
    return prediction
prediction_bilstm = prediction(model_bilstm)
prediction_lstm = prediction(model_lstm)
prediction_gru = prediction(model_gru)
# Plot true future vs prediction
def plot_future(prediction, y_test):
    plt.figure(figsize=(10, 6))
    range_future = len(prediction)
    plt.plot(np.arange(range_future), np.array(y_test),
             label='True Future')
    plt.plot(np.arange(range_future),np.array(prediction),
            label='Prediction')
    plt.legend(loc='upper left')
    plt.xlabel('Observation')
    plt.ylabel('Melt_index)')
plot_future(prediction_bilstm, y_test)
plot_future(prediction_lstm, y_test)
plot_future(prediction_gru, y_test)
```

Figure 6.22 (a) Making prediction for the three RNN architectures and plotting the true data vs prediction for the three models.

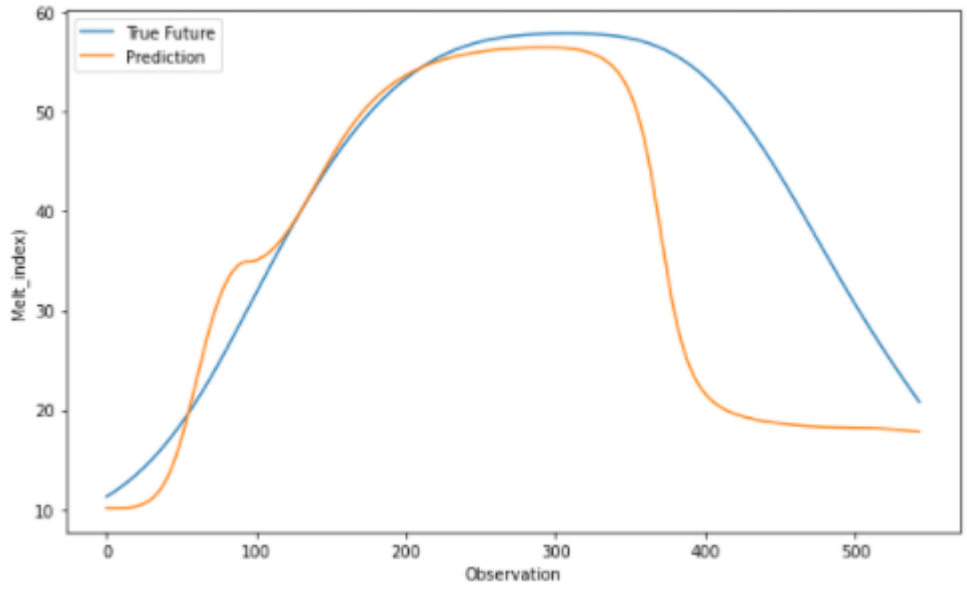


Figure 6.22(b) Prediction vs true data for BiLSTM

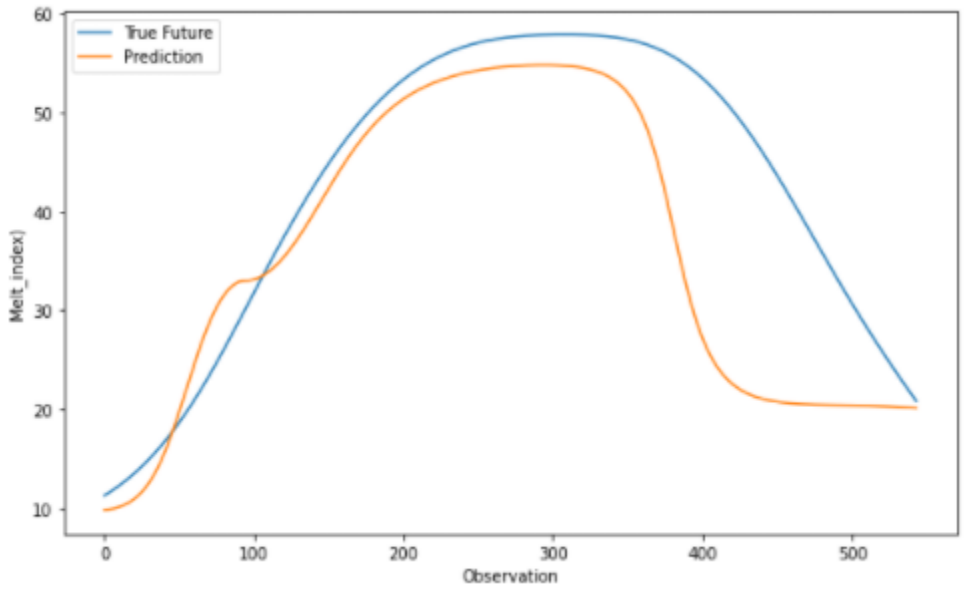


Figure 6.22(c) Prediction vs true data for LSTM

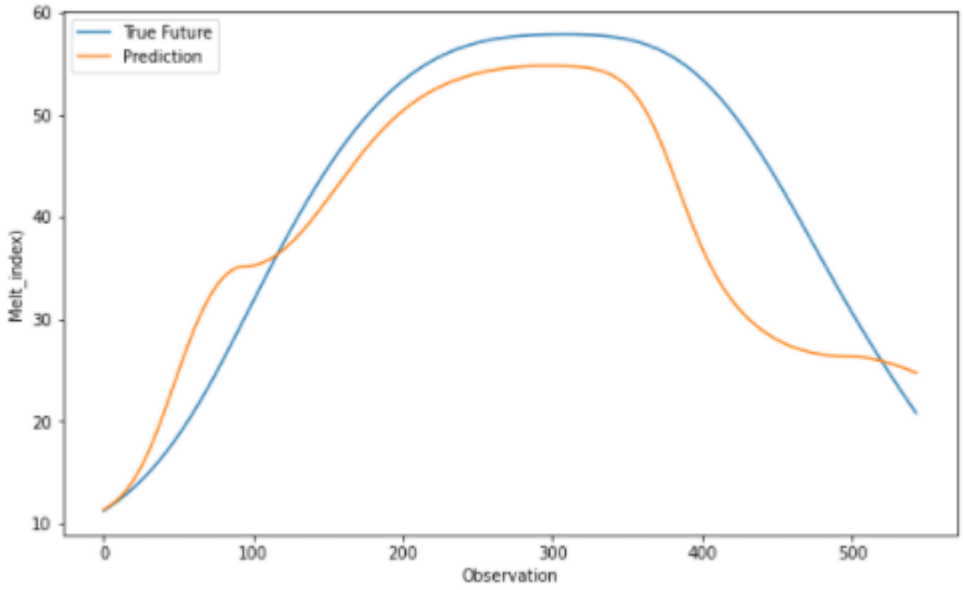


Figure 6.22(d) Prediction vs true data for GRU

As a final step, we can evaluate the performance based on different metrics like root-mean squared error(RMSE) and mean absolute error (MAE). Figure 6.23 shows the steps for model evaluation.

```
# Define a function to calculate MAE and RMSE
def evaluate_prediction(predictions, actual, model_name):
    errors = predictions - actual
    mse = np.square(errors).mean()
    rmse = np.sqrt(mse)
    mae = np.abs(errors).mean()
    print(model_name + ':')
    print('Mean Absolute Error: {:.4f}'.format(mae))
    print('Root Mean Square Error: {:.4f}'.format(rmse))
    print('')


evaluate_prediction(prediction_bilstm, y_test, 'Bidirectional LSTM')
evaluate_prediction(prediction_lstm, y_test, 'LSTM')
evaluate_prediction(prediction_gru, y_test, 'GRU')
```

Figure 6.23 Prediction of the target variable.

6.7 Results and Limitations

Table 6.3 summarizes the results from the prediction. We can see that GRU gives the best performance with the lowest MAE and RMSE. For a large dataset, we would expect BiLSTM to give the best performance, but for a smaller dataset like ours, LSTM and GRU give better performances. GRU is preferred when we want our models to converge faster and are not too deeply concerned with high accuracy.

Table 6.3. Model performance evaluation results for RNN architectures.

RNN model	MAE	RMSE
LSTM	6.33	9.18
BiLSTM	7.59	12.38
GRU	5.61	6.94

We can further improve the models by tuning the tunable hyperparameters like number of hidden layers, number of epochs, batch size, number of hidden neurons, alternative loss functions, alternate optimization algorithms, etc. It is important to use proper regularization techniques like dropout and cross-validation to ensure we account for issues like overfitting and underfitting.

References:

1. Ke, W.; Huang, D.; Yang, F.; Jiang, Y. Soft Sensor Development and Applications Based on LSTM in Deep Neural Networks. *2017 IEEE Symposium Series on Computational Intelligence (SSCI)* **2017**.

2. Xie, D.; Bai, L. A Hierarchical Deep Neural Network for Fault Diagnosis on Tennessee-Eastman Process. *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)* **2015**.
3. Goh, G. B.; Hodas, N. O.; Siegel, C.; Vishnu, A. SMILES2Vec: An Interpretable General-Purpose Deep Neural Network for Predicting Chemical Properties. *arXiv:1712.02034 [stat.ML]*, arXiv preprint, 2017. https://arxiv.org/abs/1712.02034.
4. Hou, F.; Wu, Z.; Hu, Z.; Xiao, Z.; Wang, L.; Zhang, X.; Li, G. Comparison Study on the Prediction of Multiple Molecular Properties by Various Neural Networks. *The Journal of Physical Chemistry A* **2018**, *122*, 9128–9134.
5. Du, X.; Sun, S.; Hu, C.; Yao, Y.; Yan, Y.; Zhang, Y. Deepppi: Boosting Prediction of Protein–Protein Interactions with Deep Neural Networks. *Journal of Chemical Information and Modeling* **2017**, *57*, 1499–1510.
6. Liu, D.; Tan, Y.; Khoram, E.; Yu, Z. Training Deep Neural Networks for the Inverse Design of Nanophotonic Structures. *ACS Photonics* **2018**, *5*, 1365– 1369, DOI: 10.1021/acsphotonics.7b01377
7. Zhong, Y.; Li, C.; Zhou, H.; Wang, G. Developing Noise-Resistant Three-Dimensional Single Particle Tracking Using Deep Neural Networks. *Analytical Chemistry* **2018**, *90*, 10748–10757.
8. Kuenneth, C.; Schertzer, W.; Ramprasad, R. Copolymer Informatics with Multitask Deep Neural Networks. *Macromolecules* **2021**, *54*, 7321–7321.
9. Xu, Y.; Ma, J.; Liaw, A.; Sheridan, R. P.; Svetnik, V. Demystifying Multitask Deep Neural Networks for Quantitative Structure–Activity Relationships. *Journal of Chemical Information and Modeling* **2017**, *57*, 2490–2504.
10. Zhou, Y.; Cahya, S.; Combs, S. A.; Nicolaou, C. A.; Wang, J.; Desai, P. V.; Shen, J. Exploring Tunable Hyperparameters for Deep Neural Networks with Industrial Adme Data Sets. *Journal of Chemical Information and Modeling* **2018**, *59*, 1005–1016.
11. Goli, E.; Vyas, S.; Koric, S.; Sobh, N.; Geubelle, P. H. ChemNet: A Deep Neural Network for Advanced Composites Manufacturing. *The Journal of Physical Chemistry B* **2020**, *124*, 9428–9437.
12. Munshi, J.; Chen, W.; Chien, T. Y.; Balasubramanian, G. Transfer Learned Designer Polymers for Organic Solar Cells. *Journal of Chemical Information and Modeling* **2021**, *61*, 134–142.
13. Wu, J.; Wang, S.; Zhou, L.; Ji, X.; Dai, Y.; Dang, Y.; Kraft, M. Deep-Learning Architecture in Qspr Modeling for the Prediction of Energy Conversion Efficiency of Solar Cells. *Industrial & Engineering Chemistry Research* **2020**, *59*, 18991–19000.
14. Agarwal, P.; Tamer, M.; Sahraei, M. H.; Budman, H. Deep Learning for Classification of Profit-Based Operating Regions in Industrial Processes. *Industrial & Engineering Chemistry Research* **2019**, *59*, 2378–2395.
15. Kantz, E. D.; Tiwari, S.; Watrous, J. D.; Cheng, S.; Jain, M. Deep Neural Networks for Classification OF Lc-Ms SPECTRAL PEAKS. *Analytical Chemistry* **2019**, *91*, 12407–12413.
16. Sheridan, R. P. Time-Split Cross-Validation as a Method for Estimating the Goodness of Prospective Prediction. *Journal of Chemical Information and Modeling* **2013**, *53*, 783–790.
17. Sheela, K. G.; Deepa, S. N. Review on Methods to Fix Number of Hidden Neurons in Neural Networks. *Mathematical Problems in Engineering* **2013**, *2013*, 1–11.
18. Yuan, H. C.; Xiong, F. L.; Huai, X. Y. A Method for Estimating the Number of Hidden Neurons in Feed-Forward Neural Networks Based on Information Entropy. *Computers and Electronics in Agriculture* **2003**, *40*, 57–64.
19. Madhu P.K., R.; Subbaiah, J.; Krithivasan, K. Rf-Lstm-Based Method for Prediction and Diagnosis of Fouling in Heat Exchanger. *Asia-Pacific Journal of Chemical Engineering* **2021**.
20. Hochreiter, S.; Schmidhuber, J. Long Short-Term Memory. *Neural Computation* **1997**, *9*, 1735–1780.
21. Chung J, Gulcehre C, Cho K, Bengio Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv:1412.3555, arXiv preprint, 2014. https://arxiv.org/abs/1412.3555.

22. Phi, M. Illustrated guide TO LSTM's And GRU's: A step by step explanation. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21 (accessed Sep 20, 2021).
23. Shin-ike K. A two phase method for determining the number of neurons in the hidden layer of a 3-layer neural network. *Proc SICE Ann Conf.* **2010**;2010:238–42.
24. Karsoliya S. Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture. *International Journal of Engineering Trends and Technology* **2012**, 3, 713–717.
25. Park, T. C.; Kim, T. Y.; Yeo, Y. K. Prediction of the Melt Flow Index Using Partial Least Squares and Support Vector Regression in High-Density POLYETHYLENE (HDPE) Process. *Korean Journal of Chemical Engineering* **2010**, *27*, 1662–1668.
26. Sharma, N.; Liu, Y. A. 110th Anniversary: An Effective Methodology for Kinetic Parameter Estimation for Modeling Commercial Polyolefin Processes from Plant Data Using Efficient Simulation Software Tools. Industrial & Engineering Chemistry Research 2019, 58 (31), 14209–14226.
27. Kang, H. The Prevention and Handling of the Missing Data. *Korean Journal of Anesthesiology* **2013**, *64*, 402

Chapter 7. Dissertation Summary

7.1 Concluding Remarks

In this work, we successfully demonstrate five different cases where we use machine learning algorithms backed by statistical theories for multivariate data analysis. We demonstrate applications of several novel techniques like ensemble methods, automated machine learning, exploratory time series, and deep learning for solving industrial problems.

For the LDPE dataset, we demonstrate manufacturing quality control using multi-output regression. We are able to detect anomalies and observe inter-variable and IO (input-output) relationships. We use the HDPE dataset to build a soft sensor for melt flow index. Deep neural networks give us the best results as a soft sensor among the different non-linear machine learning models. We also demonstrate a step-by-step methodology of building different types of deep neural networks for both continuous process (industrial dataset) and batch process (simulated dataset). Through our work, we also show how multivariate data analysis tools like Aspen ProMV can be used for monitoring and optimizing polyolefin industrial datasets.

The second half of this dissertation is focused on foaming control and implementation of a robust machine learning-based defoaming strategy. In the industrial sector, foaming remains an inevitable side effect of mixing, shearing, powder incorporation, and the metabolic activities of microorganisms in a bioprocess. In our work, we show how we can integrate a dataset with different fermenters using ensemble-based classification models. We use ensemble-methods to predict the exhaust differential pressure (an indicator of foaming). In order to mitigate the adverse effects of excessive foaming, we present a method of predicting the antifoam addition using automated machine learning. Finally, as a proof-of-concept, we build antifoam profiles using exploratory time-series and deploy the profiles in the industry. The initial deployment results show an overall decrease in the volume loss (up to 53%) and a decrease in the average exhaust differential pressure (up to 55%), among other indicators.

7.2 Future Outlook

In this section, we look at different approaches in which the research can be used to advance the objectives of the presented dissertation.

Fermentation Data Integration

Our classification models were able to predict the type of fermenter with operating conditions alone. This allows for integrating datasets consisting of operating conditions to build a more robust generalized model. We believe that such a fermentation integration model can have other novel applications in pattern/trend detection outside foaming.

Foam Prediction

Using ensemble-based methods we are able to predict foaming with high accuracy. It would be interesting to see if we are able to use the model for designing protocols for fermentation operation with the goal of maximizing production yield and minimizing the impacts of foaming.

Defoaming Strategy

We build antifoam profiles using exploratory time-series and deployed the profiles for one organism across three different fermenters. It would be interesting to gauge the performance of all of the six organisms across all four different fermenters. This research can be used not only for bioprocessing but for any chemical process which faces the problems of foaming. Herein, we present a novel strategy for defoaming, which is the first to use a data-based approach to reduce the impact of foaming.

Defoaming Antifoam Profile Enhancement

We present automated machine learning for predicting antifoam addition directly; currently antifoam addition constraint restricts us from using the model. We believe such a method can instead be used to build antifoam micro-profiles which can further enhance the antifoam macro-profiles built by using exploratory time-series as noted in Chapter 5.

Anomaly Detection and Quality Control

We demonstrate how Aspen ProMV and different machine learning algorithms can help with anomaly detection and quality control with multi-output regression. It would be interesting to see if we can use automated machine learning for the same.

Deep Learning

Through our work, we look at both static and dynamic polyolefin datasets and build models around industrial and simulated datasets. It would be interesting to use the dynamic regression models for other time-series datasets found in chemical processes.

Appendix

APPENDIX A

Introduction to Python for Chemical Engineers

Python is a high-level general purpose programming language designed by Guido van Rossum in 1980s. Some of the characteristics of Python which makes it such a popular programming language are: simplicity, versatility, cross-platform, open-source, free software, large unique library, and exception handling capacity.

Simplicity: Python was ranked as the second-most popular programming language by Github in 2020. [1] Due to its user-friendliness, English language parallels, and adaptability, Python aids new programmers in mastering the concepts of programming in a simple manner. Engineers new to programming can easily learn the syntaxes of the language without having to learn complex syntaxes like the ones used for programs such as C++, Java, and PHP.

Versatility: Over the past decades, Python has emerged to be one of the most diverse programming language, which can be used for software development, operations, visualization, data analytics, finance, design, machine learning, artificial intelligence, etc.

In chemical engineering, Python has a broad array of applications like soft-sensor development, data analytics and visualization, operations, process optimization, simulation, process design, automatic calculation, anomaly detection, etc.

Cross-platform: Python programs can run across different operating systems like Windows, Mac, and Linux. Some operating systems like Mac and Linux, come with preinstalled versions of Python. While, for operating system like Windows, one can easily install Python and a graphical interface if needed.

Open-source: Python is publicly accessible with open-source license. Anyone can see, modify, and distribute the code, even for commercial use.

Free software: All the versions of Python, including the latest version, can be installed for free on multiple devices for all operating systems (Windows, Mac, Linux, Other).

Large unique library: One of the reason for the popularity of Python is its huge collection of libraries, which is increasing exponentially as its simplicity attracts thousands of developers to develop new libraries. Some of the popular libraries are: Numpy, Scikit-Learn, TensorFlow, Pandas, Keras, etc.

Exception handling capacity: Errors in Python are of two types: Syntax errors and exceptions. Errors are the problems in a program due to which the program will stop the execution. Syntax errors are errors caused by a character or string incorrectly placed in a command or instruction that causes a failure in execution.

On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program. Python offers several exception handling mechanisms to elegantly handle errors

without disturbing the workflow of majority of the code and solve problems, which can sometimes speed up the script.

A.1.1 Installing Python:

For installing and using Python, we recommend using Spyder, an open-source cross-platform integrated development environment (IDE) for scientific programming in the Python language.

The best way to install Spyder as well as get other useful programming toolkits is to install a group package called Anaconda. Anaconda is a free and open-source distribution of Python and R programming languages for data science and machine learning. Anaconda comes with over a 1500 packages (including the package management system Conda) and a graphic user interface (GUI) named Anaconda Navigator. The Anaconda Navigator also allows users to install some applications by default such as Jupyter Notebook, Spyder IDE and Rstudio (for R).

The step-by-step guide for installing Anaconda Navigator is:

1) Go to anaconda.com

2) From the products category select Individual Edition.

3) Click on Download.

4) Select the right operating system and choose 64-bit processor for most advanced computer systems (ideal for RAM greater than 4GB) or 32-bit for older systems.

5) Click on executable file, then click next, read the licensing agreement and click on agree to the terms.

6) Select an install for “Just Me” unless you’re installing for all users (which requires Windows Administrator privileges) and click Next.

7) Select the installation location.

8) Choose whether to add Anaconda to your PATH environment variable. We recommend not adding Anaconda to the PATH environment variable, since this can interfere with other software. Instead, use Anaconda software by opening Anaconda Navigator or the Anaconda Prompt from the Start Menu.

9) Choose whether to register Anaconda as your default Python. We recommend selecting this option.

10) Click on Install, then Next, and finally Finish to complete installation.

For more information and operating system specific guide, users can visit:

https://docs.anaconda.com/anaconda/install/

A.1.2 Basics with Python

1. Opening Python

To use Python, we will be using Spyder IDE as a graphical interface. We open Spyder by searching for Spyder or using Anaconda Navigator.

2. Creating a new file

We create a new file by clicking on new file or pressing Ctrl+N; every new file is created as a Python script and its directory location can be selected and is visible above the variable explorer window.

It is important to save the script in the right location before executing the script.

3. Writing a script

We write the script in the command window and the executed script output can be seen in the IPython console window in Spyder. Stored functions, variables, and basic mathematical operations can directly be called in the console window.

4. Using Python as a calculator

Python can be used as a calculator to perform basic mathematical operations. For instance, we can directly use the console window for the calculation:

```
In [1]: 12/1.9926
Out[1]: 6.022282445046673
```

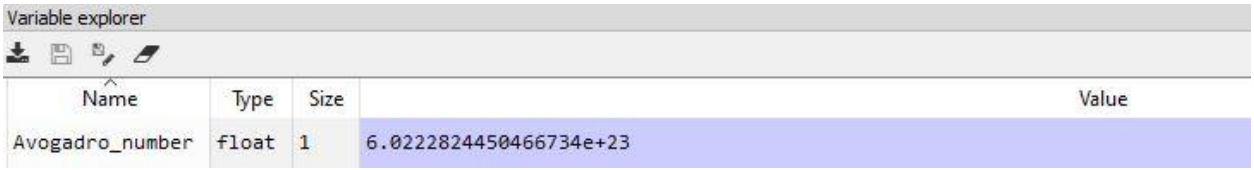
Such calculations are very basic, in order to do some complicated calculations, we need to learn about storing values in variables and some mathematical libraries, which we will look into in the sections ahead.

5. Storing values in variables

In Python, we can assign a value to a variable, using the equals sign. For instance, we can store Avogadro's number:

```
In [2]: Avogadro_number=12/(1.9926*pow(10,-23))
```

We use the built-in power function (pow) to handle exponentials. The resulting variable is stored as Avogadro_number and can be seen in the variable explorer window.



A.1.3 Different Data Types in Python

Variables can store data of different types; Python has the following data types built-in by default:

1. Text Type ('str')

To create a string, we use single or double quotes around some text, for instance:

```
In [3]: closing_salutation='CHEers'
```

2. Numeric Types ('int, float, complex')

Here we can see x, y, and z stored as integer, float, and complex numbers respectively.

Name	Type	Size	Value
x	int	1	5
y	float	1	4.63
z	complex	1	(2+5j)

3. Sequence Types ('list, tuple, range')

List: Lists are used to store multiple items in a single variable. Lists are created using square brackets. They are ordered, changeable, and allow duplicate values. List items are indexed, the first item has index [0], the second item has index [1] etc. For instance:

```
In [10]: George_Davis=['UK','Father_of_chemical_engineering',1850, 1906]
```

Tuple: Tuples are also used to store multiple items in a single variable like lists. They are created using round brackets. They are ordered, unchangeable, and allow duplicate values. Tuple items are indexed as well, the first item has index [0], the second item has index [1] etc. For instance:

```
In [11]: George_Davis=('UK','Father_of_chemical_engineering',1850, 1906)
```

Before we look into range, let us look at the differences between lists and tuples:

a. Syntax Difference: As shown above, a list is created using square brackets, while a tuple is created using round brackets.

b. Mutability: We can easily change or modify list values based on index, while a tuple cannot be changed. Since lists are mutable, we can't use a list as a key in a dictionary. This is because only an immutable object can be used as a key in a dictionary. Thus, we can use tuples as dictionary keys if needed. Below we show how the list we created before can be mutated:

```
In [13]: George_Davis[0]='Birmingham, UK'
```

Now, we can see the change in the list in the variable explorer window:

Variable explorer

Name	Type	Size	Value
George_Davis	list	4	['Birmingham, UK', 'Father_of_chemical_engineering', 1850, 1906]

c. Copying and reusability: Since, tuples are immutable, they can simply be reused with no necessity to copy. However, lists can be copied as shown below:

```
In [14]: copy_George_Davis=list(George_Davis)
```

copy_George_Davis	list	4	['Birmingham, UK', 'Father_of_chemical_engineering', 1850, 1906]

The elements in the copied list are identical to the original list, however the list itself is different, as shown below:

```
In [15]: print(copy_George_Davis is George_Davis)
False
```

d. Memory Difference: Python allocates memory to tuples in terms of larger blocks with a low overhead because they are immutable. On the other hand, for lists, Pythons allocates small memory blocks. Thus, tuples use smaller memory space compared to lists. This makes tuples a bit faster than lists when you have a large number of elements.

Range: The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number. Here we store a range of multiples of 4 starting from 4 and ending at 21:

Script:

```
Four_multiples=range(4,21,4)
for n in Four_multiples:
  print(n)
```

Output:

```
4
8
12
16
20
```

4. Mapping Type ('dict')

Dictionaries are used to store data values in 'key:value' pairs. They are created using curly brackets. They are ordered (for Python 3.7 and above, unordered for other versions), changeable, and do not allow duplicate values. Here we store the information we used before as a dictionary:

```
In [26]: Father_of_chemical_engineering = {"Name": "George Davis","Country": "United Kingdom","Birth year": 1850}
```

With dictionaries we can search for specific values for different keys easily. For instance, if we wanted to see the birth year for George Davis in our dictionary, we can simply use:

```
In [27]: print(Father_of_chemical_engineering["Birth year"])
1850
```

5. Set Types ('set', 'frozenset')

Set: Sets are used to store multiple items in a single variable. They are also created using curly brackets. They are unordered, unchangeable, unindexed, and do not allow duplicate values. Sets are mutable, allowing us to add or remove values from it.

```
In [34]: Father_of_chemical_engineering = {"George Davis","United Kingdom","1850"}
```

We can see the unordered characteristic of the set when we call out the set:

```
In [36]: Father_of_chemical_engineering
Out[36]: {'1850', 'George Davis', 'United Kingdom'}
```

We can add new values to a set using add:

```
In [37]: Father_of_chemical_engineering.add(1906)

In [38]: Father_of_chemical_engineering
Out[38]: {'1850', 1906, 'George Davis', 'United Kingdom'}
```

Frozenset: They are nothing but immutable sets. We cannot add or remove values from a frozenset, once it is created. They are sometimes used as dictionary keys, since they are immutable.

6. Boolean Type ('bool')

The bool() function allows you to evaluate any value, and give you True or False in return. The following values are considered false for bool: None, False, Zero of any numeric type (0, 0.0, 0j), empty sequence, empty mapping, etc.

```
In [41]: bool(0)
Out[41]: False

In [42]: bool(Father_of_chemical_engineering)
Out[42]: True
```

Another way to use bool is by using the built-in Boolean function:

```
In [43]: print(10 > 9)
True

In [44]: print(10 == 9)
False
```

7. Binary Type ('bytes', 'bytearray', 'memoryview')

Bytes command can convert objects into bytes objects, or create empty bytes object of the specified size. They resulting bytes objects are immutable.

Bytearray are the same as bytes, but are mutable.

Memoryview returns a memoryview object from bytes and bytearray. The resulting object can be obtained via slicing without copying the entire set of data.

A.1.4 Functions and loops in Python

Functions are blocks of code that perform specific tasks in Python. Generally, there are two types of functions: in-built functions and user-defined functions. As the name suggests, in-built functions are pre-built functions which can directly be used or called in a Python script. For instance, we used 'pow' function before in order to use exponentials to define Avogadro's number. The pow function is a built-in function.

In this section, we are going to learn how to build user-defined functions. A function is defined by the def command. We build a function my_first_function below:

```
def my_first_function(x):
    return x**2-4*x+2*x**3
```

After we save the function in a script, we can call it in the console window for different values of x as shown below:

```
In [3]: print(my_first_function(3))
51


In [4]: print(my_first_function(1))
-1
```

Functions are very useful in solving different linear and non-linear equations using Python.

Loops are used to iterate over a sequence type, allowing us to execute a command over and over.

a. For loop example

```
Father_of_chemical_engineering = {"George Davis","United Kingdom","1850"}
for x in Father_of_chemical_engineering:
  print(x)
```

```
1850
George Davis
United Kingdom
```

For loops are generally used to iterate over a sequence of numbers using range. We can use for loop for the function we built above as follows:

```
for x in range(1,6,1):
    print (my_first_function(x))
```

Here we calculate 'my_first_function' for the number 1 to 5, and get the following results:

```
-1
12
51
128
255
```

b. While loop example

```
x=0
while (x<6):
    print (my_first_function(x))
    x=x+1
```

Using this loop, we calculate 'my_first_function' till the conditional statement x<6 is met (0 to 5), we get:

```
0
-1
12
51
128
255
```

With while loop we can execute a set of statements if the condition for the loop is true.

c. Break and continue statements example

```
for val in "Engineering":
    if val == "e":
        break
    print(val)
```

```
E
n
g
i
n
```

Break statements terminate the loop containing it and are used to control the flow of the program.

Similarly, continue statements are used to instruct a loop to continue to the next iteration. For example:

```
for val in "Engineering":
    if val == "e":
        continue
    print(val)
```

```
E
n
g
i
n
r
i
n
g
```

We can see that the output is everything other than the letter 'e'.

A.2 Libraries in Python

A library is a collection of modules or a set of pre-combines codes that can be used iteratively to reduce the time required to implement a function or code. They are reusable resources which help improve effectiveness and efficiency within Python. Python by default has a standard library, which is a collection of exact syntax, token, and semantics of Python. With over 200 core modules, the python standard library provides users with several data type, text processing, mathematical, and generic operational modules.

Due to its popularity, Python has an ocean of open-source libraries under its umbrella. We look at some of the libraries popular with chemical engineers:

1. Chemics

Chemics is a Python library created by Wiggins et al. used for basic operational tools for chemical reactor engineering. [1] This library allows users to perform several operations like calculating dimensionless numbers, gas heat capacities, gas thermal conductivities, mass transfer correlations, transport velocity, pressure drop, molecular weights etc.

The library is a handy tool for chemical engineers who rely on multiple tables for such calculations. The library allows for a fast and efficient implementation of several useful chemical engineering formulas.

Here is an example, where we calculate Archimedes number for fluid transport:

We know Archimedes number is a dimensionless number used to determine the motion of fluids due to density differences. It is the ratio of gravitational forces to viscous forces.

It is given by the formula:

$$Ar = \frac{d_p^3 \rho_g (\rho_s - \rho_g) g}{\mu^2}$$

Where, d_p is the particle diameter, $\rho_g$is the gas density, ρ_s is the solid density, μ is the dynamic viscosity, and g is the local external field like gravitational acceleration.

For d_p= 1 [mm], ρ_g= 910 [kg/m^3], ρ_s= 2500 [kg/m^3], and μ= 0.001307 [kg/(m.s)]:

```
In [15]: import chemics as ch
    ...: ch.archimedes(0.001,910,2500,0.001307)
Out[15]: 8309.14521243683
```

2. Fluids

Fluids is another open-source library for chemical engineers, created by Bell et al.[2] This vast library covers many essential tools for chemical engineers ranging from piping, fittings, pumps, tanks, two phase flows, control valve sizing, pressure drop calculations, etc.

As an example, we solve for the mass flow rate (in kg/s) of an isothermal compressible gas flowing through a pipe. The formula used for the calculation is:

$$\dot{m}^2 = \frac{\left(\frac{\pi D^2}{4}\right)^2 \rho_{avg} \left(P_1^2 - P_2^2\right)}{P_1 \left(f_d \frac{L}{D} + 2 \ln \frac{P_1}{P_2}\right)}$$

Where, ρ_{avg} is the average density of gas in pipe, f_d is Darcy friction factor, P_1 and P_2 are the inlet and outlet pressures from pipe, L is the length of the pipe, D is the inner diameter of the pipe, and $\dot{m}$ is the mass flow rate of gas through the pipe.

For a gas with average density of 11.3 kg/m^3 flowing through a 1km long pipe with inner diameter of 0.5m, initially at 10 bar pressure going downstream to a pressure of 9 bar, we calculate the mass flow rate as follows:

```
In [16]: import fluids
    ...: fluids.isothermal_gas(rho=11.3,fd=0.00185,P1=1E6,P2=9E5,L=1000,D=0.5)
Out[16]: 145.4847572636031
```

Here, we input the specifications in SI units and use 0.00185 as the Darcy factor. The same methodology can be used to find different variables of the formula. For example, to find the downstream pressure for the same pipe with a flow rate of 250 kg/s:

```
In [18]: fluids.isothermal_gas(rho=11.3,fd=0.00185,P1=1E6,L=1000,D=0.5,m=250)
Out[18]: 541423.4532578246
```

We get a downstream pressure of 5.4 bars or 541423.45 pascals.

3. TensorFlow

TensorFlow is a Python library created by Google Brain Team used to create deep learning models directly or by using wrapper libraries like Keras. [3] TensorFlow allows for a series of operations on tensors; tensors are mathematical objects that can be used to describe physical properties, like scalars and vectors. Since neural networks are easily expressed as computational graphs, they can be implemented using a series of operations on Tensors using TensorFlow.

Some of the features which makes TensorFlow an ideal deep learning library are flexibility, large community, open-source, visual construct, parallel neural network training, etc. Such features as well as optimizing strategies like XLA (accelerated linear algebra) for compiling makes TensorFlow a useful library for building and optimizing deep neural networks as shown in Chapter 10.

4. Scikit-learn

Scikit-learn is a machine learning library created by Cournpeau et al. [4] It provides the users with a plethora of supervised and unsupervised machine learning algorithms for different classification, regression, and clustering tasks. It covers algorithms like K-nearest neighbors, Support Vector Machine (SVM), random forests, etc.

Some of the features of Scikit-learn which make it a standard for implementing some machine learning algorithms are availability of model evaluation techniques like cross-validation, unsupervised learning algorithms like factor analysis, unsupervised neural networks, principal component analysis, etc.

5. Numpy

Numpy is one of the most fundamental libraries in Python which offers support for multi-dimensional arrays and matrices with a large collection of mathematical functions to operate on these arrays/matrices. It was created by Oliphant et al. and is extensively used for array creation and manipulation. [5]

Numpy interface can be utilized for expressing images, sound waves, and other binary raw systems as an array of real numbers. Arrays are a collection of values that can have one or more dimensions. A Numpy array of one dimension is called a vector while one with two dimensions is called a matrix. With Numpy arrays we can perform element-wise operations which are not possible using Python lists.

6. Pandas

Similar to Numpy, Pandas is another popular library used to handle data as dataframes created by McKinney et al. [6] Pandas is widely used for most data analysis due to its flexible and extremely thorough toolkit for data manipulation. It allows users to reshape and pivot data sets with flexibility and works well with dynamic data. In addition, it also allows for label-based data slicing, indexing, and subsetting.

Pandas library can be used to import or export data from/to Microsoft Excel, making it a very handy tool for data-related operations. Below we show a basic step to read data from an excel file using Pandas:

```
In [3]: import pandas as pd
   ...: excel_file='HDPE_trainingdata.xlsx'
   ...: dataset= pd.read_excel(excel_file)
   ...: dataset.head()
Out[3]:
   Time    CAT1     C21  H21      HX1     C22     C42     HX2  H22  Melt_index
0  0.00  255.31  7000.0  8.0  15000.0  7000.0  1000.0  2000.0  1.0     11.6419
1  0.02  255.31  7000.0  8.0  15000.0  7000.0  1000.0  2000.0  1.0     11.6420
2  0.04  255.31  7000.0  8.0  15000.0  7000.0  1000.0  2000.0  1.0     11.6420
3  0.06  255.31  7000.0  8.0  15000.0  7000.0  1000.0  2000.0  1.0     11.6421
4  0.08  255.31  7000.0  8.0  15000.0  7000.0  1000.0  2000.0  1.0     11.6421
```

We use head() to look at the first few rows of our dataset.

7. Matplotlib

Matplotlib is a widely used 2-D plotting library created by Hunter et al. for creating static, animated, and interactive visualizations in Python. [7] It is used to produce publication quality figures by using Python scripts. Matplotlib can be used to generate a variety of data visualization tools like plots, histograms, bar charts, error charts, scatterplots, etc. Matplotlib is also used as a state-based interface by using 'matplotlib.pyplot', which provides users with a MATLAB-like way of plotting.

Below we show a simple example to plot a line plot using Matplotlib.pyplot:

```
import matplotlib.pyplot as plt
dataset.plot(x='Time', y='CAT1',xlabel='Time',ylabel='Catalyst flow rate',
color='red',figsize=(10,6),title='Catalyst flow rate over time')
plt.show()
```

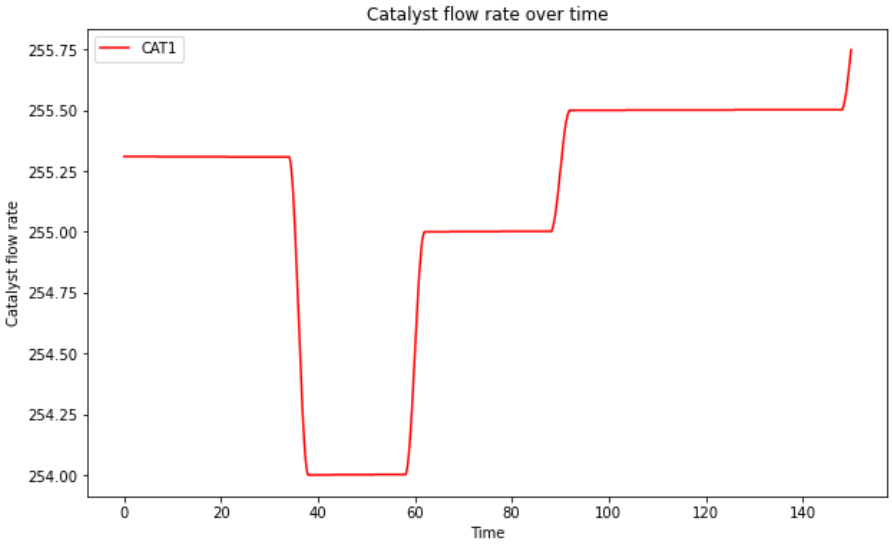


Figure A.1 Line plot for catalyst flow over time using Matplotlib

By using these libraries and many other open-source libraries, readers can handle several basic and complex engineering problems in Python with ease. This appendix introduces the readers with the fundamentals of Python, which is the most versatile and well-rounded programming language for chemical engineering applications.

A.3 Regression with Python: Hyper-parameter Optimization and sample codes

Algorithms	Standard Parameters	Commonly used values for parameters	Method for selecting paramters	Sample code
Simple Linear Regression	N/A	N/A	N/A	Appendix 2.1
K-nearest neighbor	n_neigbors, weights, algorithm, leaf_size, p, metric	n_neigbors=[3,5,10,15,20], weights=uniform, algorithm=auto, leaf_size=30, p=2, metric='minkowski'	GridSearch	Appendix 2.2
Decision Trees	criterion, splitter, max_depth, min_samples_split, min_samples_leaf,min_weight_fraction_leaf,max_features	criterion=mse, splitter=best, max_depth=[2,3,4,5,6,7,8,9,20,50,100], min_samples_split=[2,5,10,15,20,40], min_samples_leaf=[1,3,5,10,15,20], min_weight_fraction_leaf=0, max_features=auto	GridSearch	Appendix 2.3
Random Forest	n_estimators,criterion, max_depth, min_samples_split, min_samples_leaf,min_weight_fraction_leaf,max_features,bootstrap,oob_score,	n_estimators=[100,200,500,1000,2000], criterion=mse, max_depth=[2,3,4,5,6,7,8,9,20,50,100], min_samples_split=[2,5,10,15,20,40], min_samples_leaf=[1,3,5,10,15,20], min_weight_fraction_leaf=0, max_features=auto	GridSearch	Appendix 2.4
Partial Least Squares	n_components	n_components=[Integer value]	Obtained by calculating Q2_score and R2_score. As the scores stop increasing, we pick the number of components associated with the score.	Appendix 2.5
Neural Network/ Deep Neural Networks	Model, activation, loss, optimizer, batch_size,epochs, number_of_hidden_layers, input_layer_neurons, output_layer_neurons, hidden_layer_neurons,kernel_initializer	Model=Sequential(), activation=relu, loss=mean_squared_error, optimizer=adam,batch_size=[1, size of training set, 32,64, 128],epochs=[10,100,500,1000],number of_hidden_layers=[1,2,integer value],input_layer_neurons=number of input variables, output_layer_neurons=number of output variables, hidden_layer_neurons=2/3 of the input layer plus output layer neuron,kernel_initializer=normal	Obtained by a thorough understanding of the data structure, gridsearch, understanding the computational limitations, and trial/error.	Appendix 2.6
Support Vector Machine	kernel, gamma, C,	kernel=rbf, gamma=[0.001,0.01,0.1, 0.2, 0.5, 0.6, 0.9], C=[10, 100, 1000, 10000]	GridSearch and selecting suitable kernel for dataset	Appendix 2.7
Principal Component Analysis	n_components	n_components=[Integer value]	Obtained by using scree plot, the number of components retained have eigenvalues>1	Appendix 2.8

Sample code for the table above:

1) Linear Regression Sample Code:

```
from sklearn.linear_model import LinearRegression
import pandas as pd
import warnings
import numpy as np
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score
warnings.filterwarnings('ignore')
warnings.filterwarnings('ignore', category=DeprecationWarning)
dataset=pd.read_excel("LDPE dataset_KevinMcgregor.xlsx",encoding = 'unicode_escape')
dataset=dataset.drop(36,axis=0)
X=dataset.iloc[0:39,1:23].values
y=dataset.iloc[0:39,23:29].values
# define model
model = LinearRegression()
model.fit(X, y)
X_test=dataset.iloc[39:49,1:23].values
y_test=dataset.iloc[39:49,23:29].values
y1_test=y_test[:,0:1]
y2_test=y_test[:,1:2]
y3_test=y_test[:,2:3]
y4_test=y_test[:,3:4]
y5_test=y_test[:,4:5]
y6_test=y_test[:,5:6]
y_cv = cross_val_predict(model, X_test, y_test, cv=10)
y_cv1=y_cv[:,0:1]
y_cv2=y_cv[:,1:2]
y_cv3=y_cv[:,2:3]
y_cv4=y_cv[:,3:4]
y_cv5=y_cv[:,4:5]
y_cv6=y_cv[:,5:6]
score= r2_score(y_test, y_cv)
score1= r2_score(y1_test,y_cv1)
score2= r2_score(y2_test,y_cv2)
score3= r2_score(y3_test,y_cv3)
score4= r2_score(y4_test,y_cv4)
score5= r2_score(y5_test,y_cv5)
score6= r2_score(y6_test,y_cv6)
rmse = np.sqrt(mean_squared_error(y_test, y_cv))
rmse1= np.sqrt(mean_squared_error(y1_test,y_cv1))
rmse2= np.sqrt(mean_squared_error(y2_test,y_cv2))
rmse3= np.sqrt(mean_squared_error(y3_test,y_cv3))
rmse4= np.sqrt(mean_squared_error(y4_test,y_cv4))
rmse5= np.sqrt(mean_squared_error(y5_test,y_cv5))
rmse6= np.sqrt(mean_squared_error(y6_test,y_cv6))
```

2) K-NN Sample Code:

```
from sklearn.neighbors import KNeighborsRegressor
import pandas as pd
import warnings
import numpy as np
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score
warnings.filterwarnings('ignore')
warnings.filterwarnings('ignore', category=DeprecationWarning)
dataset=pd.read_excel("LDPE dataset_KevinMcgregor.xlsx",encoding = 'unicode_escape')
dataset=dataset.drop(36,axis=0)
X=dataset.iloc[0:39,1:23].values
y=dataset.iloc[0:39,23:29].values
```

```
# define model
model = KNeighborsRegressor(n_neighbors=9)
model.fit(X, y)
X_test=dataset.iloc[39:49,1:23].values
y_test=dataset.iloc[39:49,23:29].values
y1_test=y_test[:,0:1]
y2_test=y_test[:,1:2]
y3_test=y_test[:,2:3]
y4_test=y_test[:,3:4]
y5_test=y_test[:,4:5]
y6_test=y_test[:,5:6]
y_cv = cross_val_predict(model, X_test, y_test, cv=10)
y_cv1=y_cv[:,0:1]
y_cv2=y_cv[:,1:2]
y_cv3=y_cv[:,2:3]
y_cv4=y_cv[:,3:4]
y_cv5=y_cv[:,4:5]
y_cv6=y_cv[:,5:6]
score= r2_score(y_test, y_cv)
score1= r2_score(y1_test,y_cv1)
score2= r2_score(y2_test,y_cv2)
score3= r2_score(y3_test,y_cv3)
score4= r2_score(y4_test,y_cv4)
score5= r2_score(y5_test,y_cv5)
score6= r2_score(y6_test,y_cv6)
rmse = np.sqrt(mean_squared_error(y_test, y_cv))
rmse1= np.sqrt(mean_squared_error(y1_test,y_cv1))
rmse2= np.sqrt(mean_squared_error(y2_test,y_cv2))
rmse3= np.sqrt(mean_squared_error(y3_test,y_cv3))
rmse4= np.sqrt(mean_squared_error(y4_test,y_cv4))
rmse5= np.sqrt(mean_squared_error(y5_test,y_cv5))
rmse6= np.sqrt(mean_squared_error(y6_test,y_cv6))
```

3) Decision Trees Sample Code:

```
from sklearn.tree import DecisionTreeRegressor
import pandas as pd
import warnings
import numpy as np
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score
warnings.filterwarnings('ignore')
warnings.filterwarnings('ignore', category=DeprecationWarning)
dataset=pd.read_excel("LDPE dataset_KevinMcgregor.xlsx",encoding = 'unicode_escape')
dataset=dataset.drop(36,axis=0)
X=dataset.iloc[0:39,1:23].values
y=dataset.iloc[0:39,23:29].values
# define model
model = DecisionTreeRegressor()
model.fit(X, y)
X_test=dataset.iloc[39:49,1:23].values
y_test=dataset.iloc[39:49,23:29].values
y1_test=y_test[:,0:1]
y2_test=y_test[:,1:2]
y3_test=y_test[:,2:3]
y4_test=y_test[:,3:4]
y5_test=y_test[:,4:5]
y6_test=y_test[:,5:6]
y_cv = cross_val_predict(model, X_test, y_test, cv=10)
y_cv1=y_cv[:,0:1]
y_cv2=y_cv[:,1:2]
y_cv3=y_cv[:,2:3]
y_cv4=y_cv[:,3:4]
y_cv5=y_cv[:,4:5]
y_cv6=y_cv[:,5:6]
score= r2_score(y_test, y_cv)
score1= r2_score(y1_test,y_cv1)
score2= r2_score(y2_test,y_cv2)
score3= r2_score(y3_test,y_cv3)
```

```
score4= r2_score(y4_test,y_cv4)
score5= r2_score(y5_test,y_cv5)
score6= r2_score(y6_test,y_cv6)
rmse = np.sqrt(mean_squared_error(y_test, y_cv))
rmse1= np.sqrt(mean_squared_error(y1_test,y_cv1))
rmse2= np.sqrt(mean_squared_error(y2_test,y_cv2))
rmse3= np.sqrt(mean_squared_error(y3_test,y_cv3))
rmse4= np.sqrt(mean_squared_error(y4_test,y_cv4))
rmse5= np.sqrt(mean_squared_error(y5_test,y_cv5))
rmse6= np.sqrt(mean_squared_error(y6_test,y_cv6))
```

4) Random Forest Sample Code:

```
from sklearn.ensemble import RandomForestRegressor as rf
from sklearn.metrics import mean_squared_error
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings('ignore', category=DeprecationWarning)
dataset=pd.read_excel ("Mastermerged.xlsx",encoding = 'unicode_escape')
traindataset=pd.read_excel ("TrainAll.xlsx",encoding = 'unicode_escape')
testdataset=pd.read_excel ("TestAll.xlsx",encoding = 'unicode_escape')
testA=pd.read_excel ("TestA.xlsx",encoding = 'unicode_escape')
testB=pd.read_excel ("TestB.xlsx",encoding = 'unicode_escape')
testC=pd.read_excel ("TestC.xlsx",encoding = 'unicode_escape')
testH=pd.read_excel ("TestH.xlsx",encoding = 'unicode_escape')
X1=dataset.iloc[:,0:12].values
y=dataset.iloc[:,13:14].values
from sklearn import preprocessing
X=preprocessing.scale(X1)
X_train1=traindataset.iloc[:,0:12].values
X_train=preprocessing.scale(X_train1)
y_train=traindataset.iloc[:,13:14].values
X_test1=testdataset.iloc[:,0:12].values
X_test=preprocessing.scale(X_test1)
y_test=testdataset.iloc[:,13:14].values
X_testA1=testA.iloc[:,0:12].values
X_testA=preprocessing.scale(X_testA1)
y_testA=testA.iloc[:,13:14].values
X_testB1=testB.iloc[:,0:12].values
X_testB=preprocessing.scale(X_testB1)
y_testB=testB.iloc[:,13:14].values
X_testC1=testC.iloc[:,0:12].values
X_testC=preprocessing.scale(X_testC1)
y_testC=testC.iloc[:,13:14].values
X_testH1=testH.iloc[:,0:12].values
X_testH=preprocessing.scale(X_testH1)
y_testH=testH.iloc[:,13:14].values

rfr= rf(n_estimators = 1000, random_state = 423,min_samples_split = 2,min_samples_leaf= 4,
max_features='sqrt',max_depth= 50,bootstrap='True')
# Train the model on training data
rfr.fit(X_train,y_train);
predictions = rfr.predict(X_test)
# Calculate the absolute errors
errors = np.sqrt(mean_squared_error(predictions,y_test))

# Print out the mean absolute error (mae)
print('Root Mean Squared Error:', round(errors,2), 'L.')
from pprint import pprint
# Look at parameters used by our current forest
print('Parameters currently in use:\n')
pprint(rf().get_params())
from sklearn.model_selection import RandomizedSearchCV
# Number of trees in random forest
n_estimators = [200]
# Number of features to consider at every split
```

```
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [5,10,20,50]
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}
pprint(random_grid)
#Use the random grid to search for best hyperparameters
# First create the base model to tune
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
rf_random = RandomizedSearchCV(estimator = rf(), param_distributions = random_grid, n_iter = 100,
cv = 3, verbose=2, random_state=42, n_jobs = -1)
# Fit the random search model
rf_random.fit(X_train, y_train)
rf_random.best_params_
print (rf_random.best_params_)
```

5) Partial Least Squares Sample Code:

```
from sklearn.cross_decomposition import PLSRegression
import pandas as pd
import warnings
import numpy as np
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score
warnings.filterwarnings('ignore')
warnings.filterwarnings('ignore', category=DeprecationWarning)
dataset=pd.read_excel("LDPE dataset_KevinMcgregor.xlsx",encoding = 'unicode_escape')
dataset=dataset.drop(36,axis=0)
X=dataset.iloc[0:39,1:23].values
y=dataset.iloc[0:39,23:29].values
# define model
model = PLSRegression(n_components=8)
model.fit(X, y)
X_test=dataset.iloc[39:49,1:23].values
y_test=dataset.iloc[39:49,23:29].values
y1_test=y_test[:,0:1]
y2_test=y_test[:,1:2]
y3_test=y_test[:,2:3]
y4_test=y_test[:,3:4]
y5_test=y_test[:,4:5]
y6_test=y_test[:,5:6]
y_cv = cross_val_predict(model, X_test, y_test, cv=10)
y_cv1=y_cv[:,0:1]
y_cv2=y_cv[:,1:2]
y_cv3=y_cv[:,2:3]
y_cv4=y_cv[:,3:4]
y_cv5=y_cv[:,4:5]
y_cv6=y_cv[:,5:6]
score= r2_score(y_test, y_cv)
score1= r2_score(y1_test,y_cv1)
score2= r2_score(y2_test,y_cv2)
score3= r2_score(y3_test,y_cv3)
score4= r2_score(y4_test,y_cv4)
score5= r2_score(y5_test,y_cv5)
score6= r2_score(y6_test,y_cv6)
```

6) Neural Network Sample Code:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import pandas as pd
dataset=pd.read_csv ('Master batches.csv',encoding = 'unicode_escape')
X=dataset.iloc[:,0:11].values
Y=dataset.iloc[:,11:12].values
## define base model
#def baseline_model():
#   # create model
#   model = Sequential()
#   model.add(Dense(11, input_dim=11, kernel_initializer='normal', activation='relu'))
#   model.add(Dense(1, kernel_initializer='normal'))
#   # Compile model
#   model.compile(loss='mean_squared_error', optimizer='adam')
#   return model
## fix random seed for reproducibility
seed = 7
#np.random.seed(seed)
## evaluate model with standardized dataset
#estimator = KerasRegressor(build_fn=baseline_model, epochs=100, batch_size=5, verbose=0)
#kfold = KFold(n_splits=10, random_state=seed)
#results = cross_val_score(estimator, X, Y, cv=kfold)
#print("Results: %.2f (%.2f) MSE" % (results.mean(), results.std()))
#
#np.random.seed(seed)
#estimators = []
#estimators.append(('standardize', StandardScaler()))
#estimators.append(('mlp', KerasRegressor(build_fn=baseline_model, epochs=50, batch_size=5,
verbose=0)))
#pipeline = Pipeline(estimators)
#kfold = KFold(n_splits=10, random_state=seed)
#results = cross_val_score(pipeline, X, Y, cv=kfold)
#print("Standardized: %.2f (%.2f) MSE" % (results.mean(), results.std()))

# define the model
def larger_model():
#    create model
    model = Sequential()
    model.add(Dense(11, input_dim=11, kernel_initializer='normal', activation='relu'))
    model.add(Dense(5, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

np.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=larger_model, epochs=100, batch_size=5,
verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Larger: %.2f (%.2f) MSE" % (results.mean(), results.std()))

#def wider_model():
#   # create model
#   model = Sequential()
#   model.add(Dense(20, input_dim=11, kernel_initializer='normal', activation='relu'))
#   model.add(Dense(1, kernel_initializer='normal'))
#   # Compile model
#   model.compile(loss='mean_squared_error', optimizer='adam')
#   return model
#
#np.random.seed(seed)
```

```
#estimators = []
#estimators.append(('standardize', StandardScaler()))
#estimators.append(('mlp', KerasRegressor(build_fn=wider_model, epochs=100, batch_size=5,
verbose=0)))
#pipeline = Pipeline(estimators)
#kfold = KFold(n_splits=10, random_state=seed)
#results = cross_val_score(pipeline, X, Y, cv=kfold)
#print("Wider: %.2f (%.2f) MSE" % (results.mean(), results.std()))

kfold = KFold(n_splits=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Larger: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

7) SVM Sample Code:

```
from sklearn.model_selection import train_test_split
import pandas as pd
dataset=pd.read_csv ('Master batches.csv',encoding = 'unicode_escape')
X=dataset.iloc[:,0:11].values
Y=dataset.iloc[:,11:12].values
Y=Y.ravel()
X_train, X_test, y_train, y_test = train_test_split(X,Y,test_size=.2, random_state=422)
from sklearn.model_selection import GridSearchCV
import math
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
model = SVR(kernel='rbf', C=1e3, gamma = 0.5, epsilon = 0.01)
print(model)
model.fit(X_train,y_train)
pred_y = model.predict(X_test)
mse =mean_squared_error(pred_y,y_test)
print("Mean Squared Error:",mse)
rmse = math.sqrt(mse)
print("Root Mean Squared Error:", rmse)
# Tuning of parameters for regression by cross-validation
K = 10                # Number of cross valiations
# Parameters for tuning
parameters = [{'kernel': ['rbf'], 'gamma': [0.1, 0.2, 0.5, 0.6, 0.9],'C': [10, 100, 1000,
10000]}]
print("Tuning hyper-parameters")
from sklearn.metrics import make_scorer
scorer = make_scorer(mean_squared_error, greater_is_better=False)
svr= GridSearchCV(SVR(epsilon = 0.01), parameters, cv = K, scoring=scorer)
svr.fit(X, Y)
# Checking the score for all parameters
print("Grid scores on training set:")
means = svr.cv_results_['mean_test_score']
stds = svr.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, svr.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r"% (mean, std * 2, params))
```

8) PCA Sample Code:

```
from sklearn.decomposition import PCA
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings('ignore', category=DeprecationWarning)
dataset=pd.read_excel("LDPE dataset_KevinMcgregor.xlsx",encoding = 'unicode_escape')
dataset=dataset.drop(36,axis=0)
X=dataset.iloc[0:39,1:23].values
model =PCA(n_components=2)
model.fit_transform(X)
print(pca.explained_variance_ratio_)
print(pca.singular_values_
```

References:

1. Wiggins, G. https://chemics.github.io/ (accessed Aug 26, 2021).
2. Caleb Bell (2016-2021). fluids: Fluid dynamics component of Chemical Engineering Design Library (ChEDL) https://github.com/CalebBell/fluids (accessed Aug 26, 2021).
3. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; Kudlur, M.; Levenberg, J.; Monga, R.; Moore, S.; Murray, D. G.; Steiner, B.; Tucker, P.; Vasudevan, V.; Warden, P.; Wicke, M.; Yu, Y.; Zheng, X. TensorFlow: A system for large-scale machine learning. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, 2016.
4. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **2011**, 12, 2825– 2830.
5. Harris, C. R.; Millman, K. J.; van der Walt, S. J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N. J.; Kern, R.; Picus, M.; Hoyer, S.; van Kerkwijk, M. H.; Brett, M.; Haldane, A.; del Río, J. F.; Wiebe, M.; Peterson, P.; Gérard-Marchant, P.; Sheppard, K.; Reddy, T.; Weckesser, W.; Abbasi, H.; Gohlke, C.; Oliphant, T. E. Array Programming with Numpy. Nature **2020**, 585 (7825), 357–362.
6. McKinney, W. Data Structures for Statistical Computing in Python. Proceedings of the 9th Python in Science Conference, Austin, TX, June 28 to July 3, 2010.
7. Hunter, J. Matplotlib: A 2D Graphics Environment. Comput. Sci. Eng. 2007, 9, 90, DOI: 10.1109/MCSE.2007.55

APPENDIX B

Introduction to Neural Networks

B.1 What is a Neural Network?

Artificial neural networks, commonly known as 'neural networks' mimic the mechanism of learning in biological organisms. Neurons, which are the basic working unit of a brain, transmit information through specialized projections called axons and dendrites. Such a transmission is replicated in an artificial neural network, where each neuron is a mathematical operation connected to other neurons by weights. Each input to the input neurons is scaled with weights and the sum is passed through different activation functions towards the output neurons. Learning occurs by simultaneously changing the weights connecting the neurons.

B.2 Developing Neural Network Architecture and Some Common Topology for Neural Networks

B.2.1 Modeling a neuron

A neuron is the fundamental information processing unit of the network. In Figure C.1, we identify the four basic components of a neuron:

i. Synapses: Connecting links which are characterized by a weight of its own (w_{ij}). The weight includes both positive and negative values.

ii. Adder: Operator for summing up input signals modified by their respective synaptic weights.

iii. Activation function: A mathematical function which limits the amplitude of the output of a neuron to a finite value. We compare different activation functions and their uses ahead in section 1.5.

iv. Bias: A constant used to provide affine transformation to the adder signal by increasing or lowering the net input of an activation function.

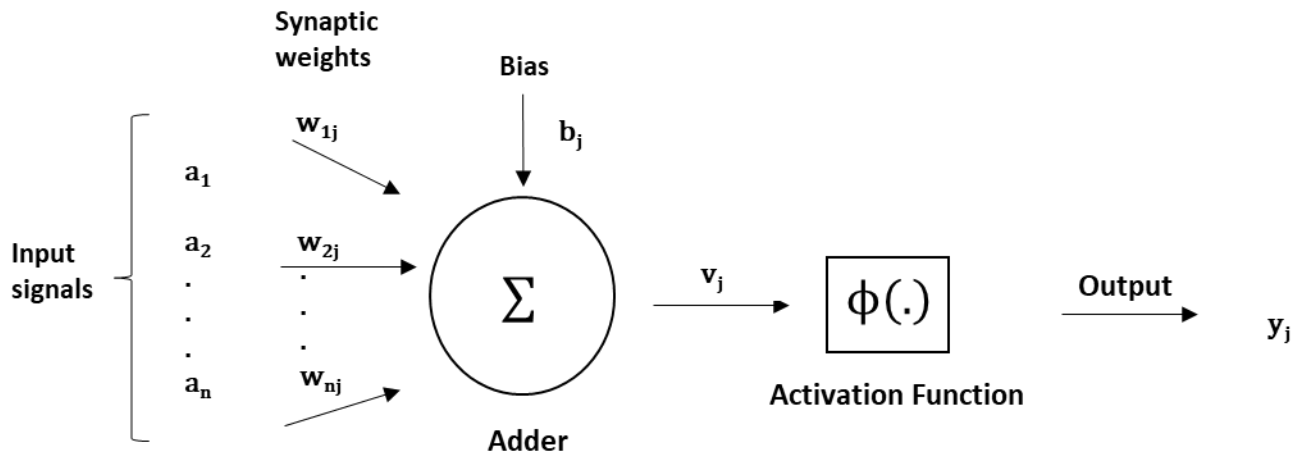


Figure C.1 Anatomy of a simple neuron.

We can mathematically describe a neuron by using the following equations:

$$u_j = \sum_{i=1}^{n} w_{nj} a_n \quad \text{(B.1)}$$

$$v_j = u_j + b_j \quad \text{(B.2)}$$

$$y_j = \phi(v_j) \quad \text{(B.3)}$$

B.2.2 Topology of a neural network

Neural nets are arranged into an ordered sequence of grouped partitions known as layers. The *topology* of a neural network refers to the interconnection of the neurons. The first layer is called the input layer and usually has the same number of neurons as the number of input variables for the system, the last layer is known as the output layer; the layers in between are called hidden layers. The architecture of the neural network depends on the connection patterns, number of neurons, number of layers, nature of the activation functions, and the learning algorithm.

The connection patterns for a neuron can be partitioned into four categories: intralayer connections, interlayer connections, and self-connections, and supralayer connections. An intralayer connection is a connection between neurons of the same layer of the neural network. An interlayer connection is a

connection between neurons in adjacent layers of the neural network. Self-connections are connections which originate and terminate at the same neuron. Supralayer connections are connections between neurons that are neither in the same layer nor in the adjacent layer of the neural network.

Within the interlayer connections, we have two directional connections:

1) Feed-forward neural network: Unidirectional networks in which the signal flows from input to output without forming a loop. They are the simplest form of interlayer connection, which can be single-layered or multi-layered.

2) Feedback or recurrent neural network: Multidirectional networks in which the signal flows in both directions forming a loop. They are dynamic in nature and are inherently more complex than feedforward networks.

B.2.3 Common Architectures for Neural Networks

Over time researchers have come up with several architectures for building neural networks. We describe some of these popular architectures used in chemical engineering:

1) Perceptron

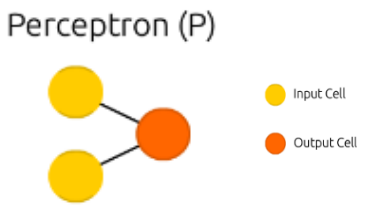


Figure B.2 Perceptron model architecture.

Perceptron model, proposed by Minsky-Papert is the simplest model architecture which can learn linearly separable problems. It accepts weighted inputs and applies the activation function to obtain the output. A single-layer perceptron with no hidden layers is no longer used for solving problems; multi-layer perceptron models are more common as they provide more complexity and non-linear capabilities.

2) Feed Forward Neural Network

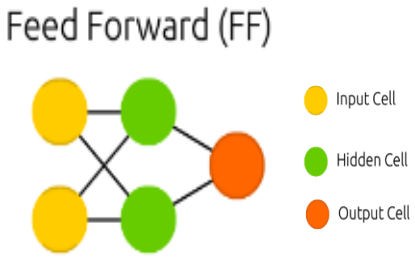


Figure B.3 Feed-forward model architecture.

As mentioned above in Section C.2.2, feed forward neural networks are unidirectional networks which have forward propagation leading to static weights. They are less complex, fast, and responsive to noisy datasets. Feedforward networks are primarily used for supervised learning in cases where the dataset is neither sequential nor time dependent. Supervised learning refers to learning techniques which include training with both input and correct output data to build a network.

3) Recurrent Neural Network

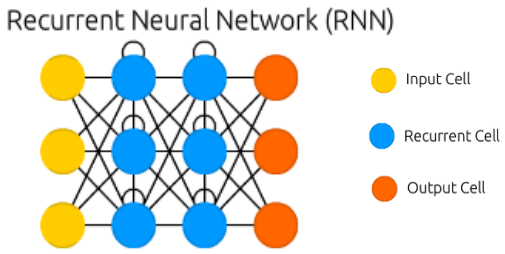


Figure B.4 Recurrent neural network model architecture.

Recurrent neural networks are multidirectional networks in which the output from previous step is fed as input for the current step for prediction. The first layer is typically a feed forward neural network followed by recurrent neural network layer where a memory function is used to retrieve information from the previous time-step. They are useful in modeling sequential data and can process inputs of any length.

4) Deep Convolutional Neural Network

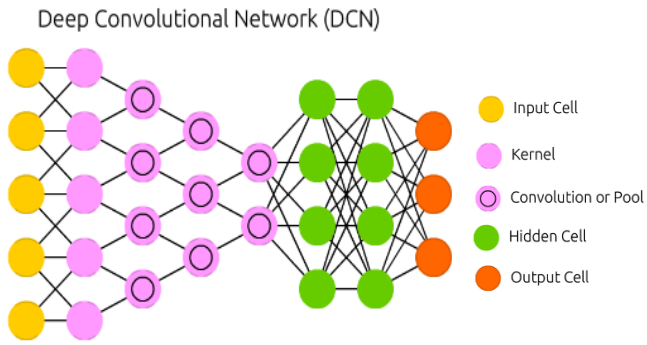


Figure B.5 Deep convolutional network model architecture.

Deep convolutional neural networks are deep neural networks which were primarily designed for computer vision tasks. These networks contain a three-dimensional arrangement of neurons instead of the standard two-dimensional setup. These networks are based on two basic operations: convolution and pooling. The convolution step is used to extract features from the dataset. The input features are taken in batch-wise like a filter. The pooling step, also called subsampling, involves reducing the dimensionality of the features extracted from the convolution step.

5) Radial-Basis Function Neural Network

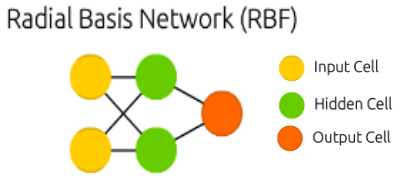


Figure B.6 Radial basis network model architecture.

Radial-basis function neural networks are networks that use Gaussian functions as activation functions. It is usually a three-layered feed forward network with a single hidden layer. The network is known as a local approximation network with a linear output to map non-linear input data. These networks are faster and have better scaling properties when compared to a simple feed-forward network.

B.3 Comparison of different activation functions

Activation functions act as mathematical channels which limit the amplitude of the output of a neural network. These functions can be divided into three categories: i. Binary step function, ii. Linear activation function, and iii. Non-linear activation functions.

Binary activation function is a threshold-based activation function. If the input value is above or below a certain threshold, the neuron is activated, and a signal is sent to the next layer. It does not allow multi-value outputs. Linear activation function creates an output signal proportional to the input and is not confined in a range as it can take values from (-∞,∞). Linear activation functions have limited power and do not allow for the usage of backpropagation or multiple hidden layers as with linear activation the final layer will always be a linear function of the first layer.

Non-linear activation functions are the basis for most common neural network topologies. Some of the common non-linear activation functions are:

1. **Sigmoid/Logistic**

Equation: $sig(x) = \frac{1}{(1+e^{-x})}$ (B.4)

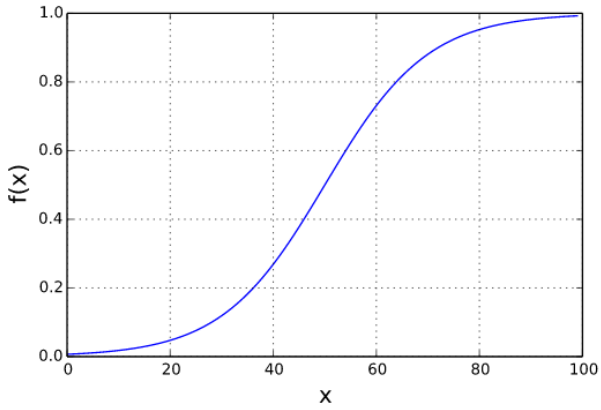


Figure B.7 Sigmoid activation function

The sigmoid function curve looks like a S-shape curve and the function provides output values ranging from 0 to 1. Due to the specific range, it is commonly used for models which predict probability as an output. It provides smooth gradient preventing drastic changes in outputs; however, very large or very small input values are not represented as well in the output leading to vanishing gradient problem.

2. **Hyperbolic Tangent (TanH)**

Equation: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (B.5)

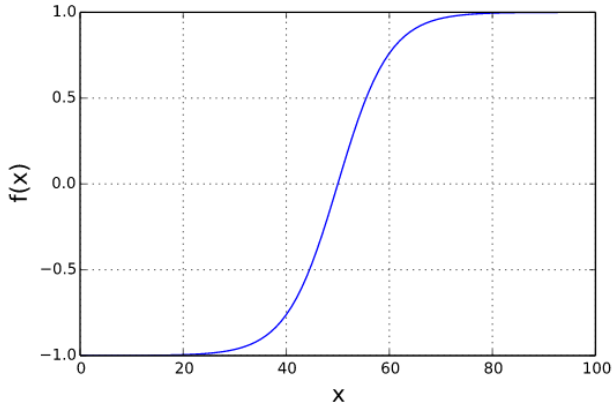


Figure B.8 Hyperbolic tangent activation function

TanH function is a mathematically shifted version of the sigmoid function with output values ranging from -1 to 1. Due to its range, its mean is zero-centered making it easier to model inputs with negative, neutral, and positive values.

3. **Rectified Linear Unit (ReLU)**

Equation: $ReLU(x) = \max(0, x)$ (B.6)

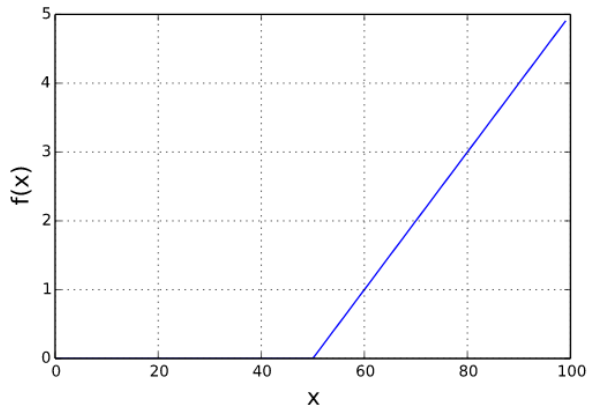


Figure B.9 Rectified linear unit activation function

ReLU function is the most widely used activation function. Although it seems like a linear function, ReLU function is non-linear and has a derivative function. It is computationally efficient and allows for backpropagation. It does not handle inputs closer to zero and negative inputs very well during backpropagation leading to 'dying ReLU' problem, in which the gradient of the function becomes zero.

4. **Leaky ReLU**

   Equation: $LReLU(x) = \max(0.01 * (\mathrm{x}, \mathrm{x}))$ (B.7)

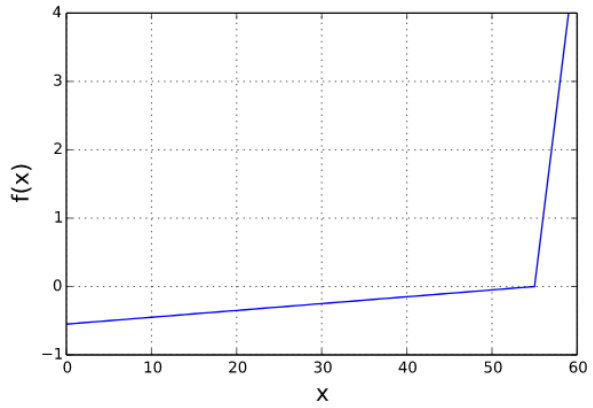


Figure B.10 Leaky rectified linear unit activation function

A variation of the ReLU function which allows for backpropagation for negative input values. It does not provide consistent predictions for negative input values.

5. **Parametric ReLU**

   Equation: $PReLU(x) = \max(\alpha * (\mathrm{x}, \mathrm{x}))$ (B.8)

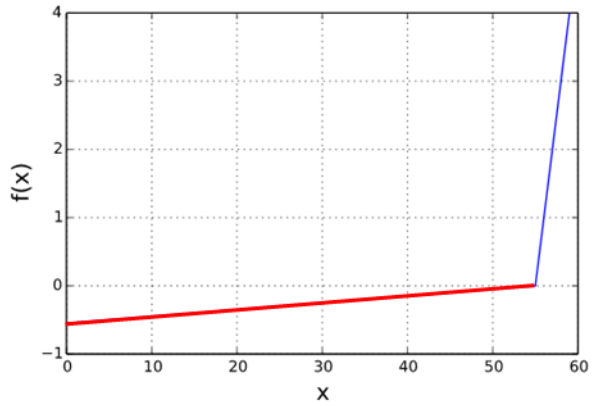


Figure B.11 Parametric rectified linear unit activation function (red line changes based on the value of α

Another variation of ReLU function which allows for negative slopes to be learned as well. Performance is dependent on the chosen parametric value. For α=0.01 PReLU becomes LReLU.

6. **Softmax**

Equation: $S(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$ (B.9)

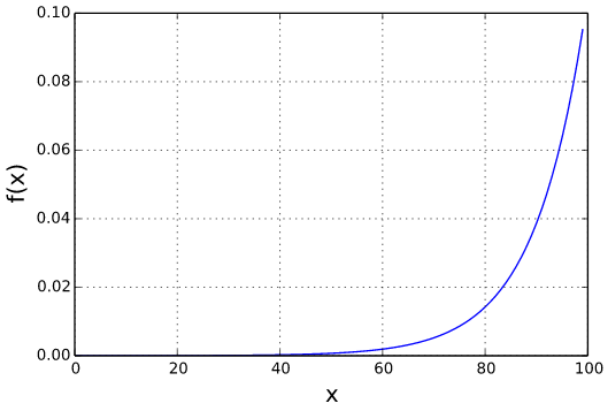


Figure B.12 Softmax activation function

Softmax activation function is a normalized exponential function which generalizes logistic function to multiple dimensions. It is generally used for multinomial logistic regression and multiclass classification

problems. The range of output values for the function fall from 0 to 1 and the sum of the output values is equal to 1.

7. **Swish**

Equation: $swish(x) = \frac{x}{(1+e^{-x})}$ (B.10)

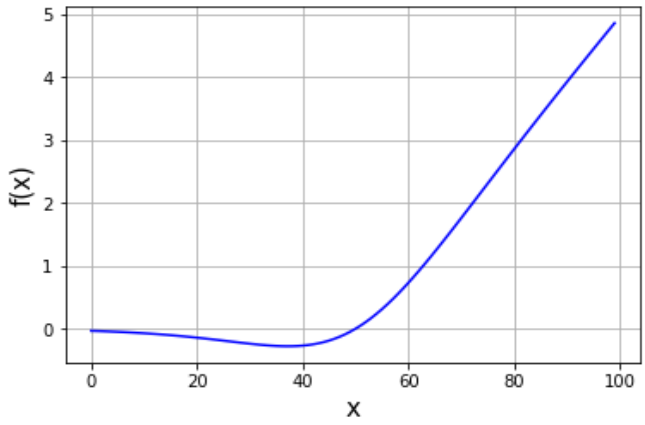


Figure B.13 Swish activation function

A simple modification to the ReLU function, designed by the Google Brain Team, swish activation function is a smooth non-monotonic function which outperforms or matches the performance of ReLU activation function in most cases. The lower bound for the network is bounded such that large negative weights are zeroed out, while its upper bound is unbounded such that large positive weights are not saturated to 1.

B.4 Optimization Strategies

Optimization strategies are often used in a neural network to reduce losses and increase the network accuracy by adjusting the attributes of the neural network like synaptic weights and learning rate. Some of the commonly used optimization strategies are:

1. **Gradient Descent**

Gradient descent is one of the most popular and common ways of optimizing a neural network. In gradient descent, we minimize an objective function J(θ) parameterized by the model's parameters $\theta \in R$, by updating them in the opposite direction of the gradient of the objective function, explained by the following equation:

$$\theta^1 = \theta^0 - \eta \nabla J(\theta) \quad \text{(B.11)}$$

where,

$\theta^1 = Next\ position,$

$\theta^0 = Current\ position,$

$\eta = Learning\ rate$

$\nabla J(\theta) = Gradient$ of the objective/cost function

Gradient descent is a first-order derivative optimization algorithm in which the learning rate(α) is altered to decide on the number of steps to be taken to reach a (local) minimum. Based on the amount of data used, gradient descent can be further divided into three variants:

a) Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) updates the parameters of the function after computation of loss on each training example. The parameters have high variance resulting in heavy fluctuations in the objective function. By using a single sample data per iteration, SGD converges relatively quicker and requires less memory compared to other optimization strategies. Due to the inherent randomness, SGD is noisier than simple gradient descent and can overshoot even after achieving a global minimum.

b) Batch Gradient Descent

Batch gradient descent or vanilla gradient computes the gradient of the loss function for the entire dataset. Since, it requires gradient computation for the entire dataset, it can be very slow and intractable for larger datasets that do not fit in memory. It is also very hard to apply on-line or in real time. Batch gradient descent is the traditional form of optimization which is simple to implement and interpret for smaller datasets.

c) Mini-batch Gradient Descent

Mini-batch gradient descent takes the best from batch gradient descent and stochastic gradient descent to compute gradient for a mini-batch of n training examples. In addition to frequently updating the parameters, mini-batch gradient descent has less variance. The computation requires medium amount

of memory storage capacity. Even though it addresses some drawbacks of the gradient descent variants, mini-batch gradient has some drawbacks as well.

The selection of an optimal learning rate for mini-batch gradient descent can be difficult as a smaller learning rate can take a long time for the algorithm to converge, and a larger learning rate can lead to the algorithm getting trapped at some suboptimal local minima.

2. Momentum

Momentum is an optimization strategy introduced to accelerate convergence of SGD towards the relevant direction and dampen oscillations to reduce the fluctuations in irrelevant direction. Another hyperparameter, known as momentum(γ), is added leading to the following equation:

$$V(t) = \gamma V(t-1) + \eta \, \nabla J(\theta) \quad \text{(B.12)}$$

where,

$V(t) = velocity\ at\ time\ t$

$\gamma = momentum\ or\ friction$ (usually given a value of 0.9)

The weights in the SGD equation (B.11) are updated accordingly, leading to the following equation:

$$\theta^1 = \theta^0 - V(t) \quad \text{(B.13)}$$

The momentum optimization algorithm helps minimize the oscillations and reduces the high variance in SGD. Momentum-based SGD converges faster than regular SGD.

3. Nesterov Accelerated Gradient

Nesterov Accelerated Gradient (NAG) is a variation of Momentum optimization algorithm, which is slightly better than the standard momentum optimization. With NAG, instead of calculating gradient at current position, we calculate the gradient for a future position. By applying this small change, it prevents the momentum term from overshooting and missing the local minima. The momentum equation (B.12) is changed to the following equation:

$$V(t) = \gamma V(t-1) + \eta \, \nabla J(\theta - \gamma V(t-1)) \quad \text{(B.14)}$$

4. Adagrad

Adagrad is a second-order derivative optimization algorithm in which the learning rate (η) is changed for each parameter for each step 't'. The algorithm makes smaller updates (low learning rates) for frequently occurring features and bigger updates (high learning rates) for infrequent features. In the algorithm we take the partial derivative of the objective function at a time t, given by:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}) \quad \text{(B.15)}$$

The updated parameters are then given by the following equation:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}+\epsilon}} \cdot g_{t,i} \quad \text{(B.16)}$$

where,

$G_{t,ii} = Diagonal\ matrix\ with\ diagonal\ element\ (i,i)\ as\ the\ sum\ of\ squares\ of\ the\ gradients$

$\epsilon = Smoothing\ term\ to\ avoid\ division\ by\ zero$

One of the main advantages of using Adagrad optimization strategy is that it eliminates the need to manually tune the learning rate. Most implementations use an initial rate of 0.01 and allow for it to be updated with the parameters for each iteration at a specific time interval (t). One of the drawbacks of Adagrad is the accumulation of square of gradients in the denominator leading to the shrinkage of the learning rate, as it eventually becomes infinitesimally small. At that point, the algorithm is unable to gain any further new information.

5. AdaDelta and RMSProp

AdaDelta, is an extension of Adagrad, which successfully eliminates the problem of diminishing learning rate due to accumulation of square of gradients. AdaDelta limits the window of past gradients to some fixed size w.

We use exponentially running average of all the past squared gradients instead of inefficiently storing them. The running average at time t, depends only on the previous average and the current gradient and is given by:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \quad \text{(B.17)}$$

Like equation B.16, the new equation for AdaDelta becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[g^2]_{t,i}+\epsilon}} \cdot g_{t,i} \quad \text{(B.18)}$$

where,

$E[g^2]_{t,i} = Exponential\ moving\ average\ at\ time\ step\ t$

Since, the denominator is just the root mean squared error criterion we replace it with the short-hand criterion:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{RMS[g]_{t,i}} \cdot g_{t,i} \quad \text{(B.19)}$$

Both AdaDelta and RMSProp have been developed independently around the same time to deal with Adagrad's diminishing learning rate. The equation for RMSProp is identical to the first parameter update for AdaDelta, represented by equation B.19. In order to ensure the same units as the parameter for the gradient, for AdaDelta, we define another exponentially decaying average based on the square parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \quad \text{(B.20)}$$

The root mean squared of the parameter update then becomes:

$$RMS[\Delta\theta]_{t,i} = \sqrt{E[\Delta\theta^2]_{t,i} + \epsilon} \quad \text{(B.21)}$$

Since, $RMS[\Delta\theta]_{t,i}$ is unknown, we approximate it with the RMS of the parameter update until the previous time step. Then by replacing $RMS[\Delta\theta]_{t-1,i}$ with the learning rate η, in equation B.19, we get:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{RMS[\Delta\theta]_{t-1,i}}{RMS[g]_{t,i}} \cdot g_{t,i} \quad \text{(B.22)}$$

With equation B.22, we remove the learning rate from the parameter update to give the final AdaDelta update equation. AdaDelta, while being more computationally expensive, prevents learning rate decay and eliminates the necessity of setting a default learning rate. RMSProp on the other hand deals with the decaying learning rate but does not eliminate the necessity to set a default learning rate.

6. Adam

Adaptive Momentum Estimation (Adam) combines the momentum model with adaptive models like RMSProp and AdaDelta. It is one of the most used optimization algorithms, which converges very fast and includes a bias correction mechanism. Adam stores both the exponential running average of the past gradients (m^t) and the exponential running average of the past-squared gradients (v^t). The decaying averages are calculated as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad \text{(B.23)}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) {g_t}^2 \quad \text{(B.24)}$$

where,

$m_t = First\ moment\ (mean)\ of\ the\ gradient$

$v_t = Second\ moment\ (uncentered\ variance)\ of\ the\ gradient$

$\beta_1, \beta_2 = Exponential\ decay\ rates\ for\ the\ moment\ estimates$

As the moments are initiated as vectors of zero, during the initial time steps when the decay rates are small (i.e., when $\beta_1$and β_2 are close to 1), the moments are biased towards zero. In order to correct the raw moment estimates, we use the following equations:

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t} \quad \text{(B.25)}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad \text{(B.26)}$$

where,

$\widehat{m}_t$ and $\hat{v}_t$ are the bias-corrected first and second moment estimates

Like AdaDelta (Equation B.19) and RMSProp (Equation B.22), we get the final parameter update using the above moment estimates as:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\hat{v}_{t,i}+\epsilon}} . \widehat{m}_{t,i} \quad \text{(B.27)}$$

Adam is one of the most popular optimization techniques, which converges rapidly and solves the problem of vanishing learning rate and high variance. The common values for β_1 is 0.9, β_2 is 0.999, and ϵ is 10^{-8}.

7. Nadam

Nadam (Nesterov-accelerated Adaptive Moment Estimation) is a variation to Adam optimization technique, which combines Adam and NAG. The algorithm applies acceleration to the parameters before computing the gradients, then updates the gradients computed with the interim parameters. This change allows the algorithm to mitigate problems like accumulation of large error gradients, also

known as exploding gradients.

In order to modify Adam to Nadam, we update m_t, the first moment term for the gradient. First, let us recall Nesterov-accelerated Momentum (NAG), which can be shown from equation B.14:

$$V(t) = \gamma V(t-1) + \eta \nabla J(\theta - \gamma V(t-1))$$

Since, V(t) is also known as the first moment of a gradient (m_t) and based on the terminologies used for Adagrad and Adam, we can rewrite the terms for equation B.14 as :

$$m_t = \gamma m_{t-1} + \eta g_t \quad \text{(B.28)}$$

where,

$$g_t = \nabla_\theta J(\theta_t - \gamma V(t-1)) \quad \text{(B.29)}$$

Then, the parameter update for the NAG equation becomes:

$$\theta_{t+1,i} = \theta_{t,i} - m_{t,i} \quad \text{(B.30)}$$

$$\theta_{t+1,i} = \theta_{t,i} - (\gamma m_{t-1} + \eta g_t) \quad \text{(B.31)}$$

Dozat et al. proposed to modify NAG in the following manner. Instead of applying the momentum step twice for both the gradient and then the parameters, we now use the current momentum vector directly to update the parameters, which can be shown by the following equations:

$$g_t = \nabla_\theta J(\theta_t) \quad \text{(B.32)}$$

$$m_t = \gamma m_{t-1} + \eta g_t \quad \text{(B.33)}$$

$$\theta_{t+1,i} = \theta_{t,i} - (\gamma m_t + \eta g_t) \quad \text{(B.34)}$$

By comparing equation B.31 and equation B.34, we can see that the update no longer utilizes the previous momentum vector, m_{t-1} for parameter update. And, also from equation B.32, we see that the gradient is no longer based on momentum. Now, in order to combine NAG with Adam, we recall the equations B.23, B.25, and B.27:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\hat{v}_{t,i}} + \epsilon} \cdot \hat{m}_{t,i}$$

When we use equations B.23 and B.25 to expand equation B.27, we get:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\hat{v}_{t,i}}+\epsilon} \cdot \left[\frac{\beta_1 m_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right] \quad \text{(B.35)}$$

We can see that, $\frac{\beta_1 m_{t-1}}{1-\beta_1^t}$ is just the bias-corrected estimate of the momentum vector from the previous time step. For simplicity, we ignore that the denominator is $1-\beta_1^t$ instead of $1-\beta_1^{t-1}$. We can therefore replace $\frac{\beta_1 m_{t-1}}{1-\beta_1^t}$ with $\hat{m}_{t-1}$, giving us the following equation:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\hat{v}_{t,i}}+\epsilon} \cdot \left[\beta_1 \hat{m}_{t-1} + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right] \quad \text{(B.36)}$$

Just like we updated NAG shown by equation B.31 into equation B.34, by eliminating the previous momentum vector for parameter update. Similarly, we eliminate the bias-corrected estimate of the momentum vector from the previous time step and replace it with the current bias-corrected estimate of the momentum vector:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\hat{v}_{t,i}}+\epsilon} \cdot \left[\beta_1 \hat{m}_{t} + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right] \quad \text{(B.37)}$$

This equation represents the final parameter update using Nadam. Nadam is slightly more computationally expensive than Adam but uses Nesterov Accelerated Gradient (NAG) instead of vanilla momentum update.

References:

1. Veen, F. van. Fjodor van Veen, author at the Asimov Institute. https://www.asimovinstitute.org/author/fjodorvanveen/ (accessed Dec 6, 2021).