

Project on
Enhancing Response Combination Efficiency with Support Vector Regressor and Kernel
Variants and its application in Control Chart

(SQC & OR Unit, ISI, Bangalore)

By

SUSMIT SEN

MS-QMS (2nd year)

Roll no. – mqms2218.

Indian Statistical Institute, Bangalore

UNDER THE GUIDANCE OF

Dr. Moutushi Chatterjee



SQC & OR UNIT

INDIAN STATISTICAL INSTITUTE

BANGALORE

Submitted for partial fulfillment of
Master of Science in Quality Management Science

ACKNOWLEDGEMENT

I express my sincere gratitude to Dr. Moutushi Chatterjee, SQC & OR Unit, Indian Statistical Institute, Bangalore for her invaluable guidance and encouragement to pursue such a challenging problem. I am deeply indebted to my supervisor for helping me at every step of my dissertation.

I desire to convey my gratitude to all other people associated with this work. I would like to thank all faculty members and classmates for their encouragement and help during this dissertation.

Susmit Sen

2nd year, MS-QMS

Roll No. mqms2218.

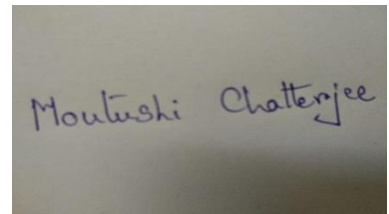
SQC & OR Unit

Indian statistical Institute, Bangalore

CERTIFICATE

TO WHOM IT MAY CONCERN

This is to certify that the dissertation titled “Enhancing Response Combination Efficiency with Support Vector Regressor and Kernel Variants and its application in Control Chart” is a bonafied work done by Susmit Sen, as a partial fulfillment for the award of degree in Master of Science in Quality Management Science under my guidance Dr. Moutushi Chatterjee

A photograph of a handwritten signature in blue ink on a light-colored surface. The signature reads "Moutushi Chatterjee".

Dr. Moutushi Chatterjee

SQC & OR Unit

Indian statistical Institute, Bangalore

1 Introduction

SQC stands for Statistical Quality Control, a methodology used in manufacturing and service industries to monitor and control processes to ensure they meet desired quality standards. It involves the application of statistical methods to analyze data and make informed decisions about process performance and improvement. At its core, SQC focuses on understanding and reducing process variation, which is critical for maintaining consistency and quality in the production of goods or delivery of services. By collecting and analyzing data, SQC helps identify trends, patterns, and potential sources of variation within a process, allowing for proactive adjustments to minimize defects and waste. Key components of SQC include statistical tools such as control charts, process capability analysis, and hypothesis testing. These tools enable organizations to monitor process performance over time, detect deviations from desired targets or specifications, and take corrective actions as needed to maintain quality standards and customer satisfaction.

Overall, SQC plays a vital role in driving continuous improvement efforts, enhancing productivity, and ultimately, ensuring the delivery of high-quality products and services to customers.

Statistical Quality Control (SQC) is a methodology used in various industries to monitor and improve processes, ensuring they meet predefined quality standards. By employing statistical techniques and tools, SQC aims to analyze process data, identify variations, and make informed decisions to enhance quality and efficiency. This abstract provides a concise overview of SQC and its importance in contemporary manufacturing and service sectors. SQC plays a crucial role in reducing defects, minimizing waste, and improving customer satisfaction. Through the implementation of SQC methodologies, organizations can achieve continuous improvement, maintain competitiveness, and drive sustainable growth in today's dynamic business landscape.

Machine learning (ML) in Statistical Quality Control (SQC) involves leveraging algorithms and statistical models to analyze data and improve process control and quality assurance. In SQC, machine learning techniques are employed to handle large volumes of data efficiently, identify patterns, detect anomalies, and predict potential quality issues before they occur. ML algorithms can be trained on historical process data to recognize relationships between process variables and quality outcomes, enabling proactive decision-making to prevent defects or deviations from quality standards.

One common application of ML in SQC is predictive maintenance, where algorithms analyze sensor data from machines to predict equipment failures before they happen, reducing downtime and maintenance costs. ML can also be used for fault detection and classification, identifying anomalies in production processes that could lead to defects or quality issues.

Moreover, ML algorithms can enhance the effectiveness of traditional SQC methods such as control charts by automatically detecting shifts or trends in process data and providing early warnings of potential quality problems. Additionally, ML can optimize process parameters to improve quality and efficiency, leading to better overall process performance.

Overall, integrating machine learning into SQC allows organizations to achieve higher levels of quality control, reduce waste, and enhance productivity by leveraging advanced data analytics techniques to continuously monitor and improve processes.

2 Some Machine Learning concept

2.1 Hyperparameter Tuning

Hyperparameter tuning is a crucial step in the machine learning pipeline, aimed at optimizing model performance and generalization ability by selecting the best hyperparameter values. Numerous works have been conducted to develop efficient hyperparameter tuning methods, explore optimization techniques, and understand the impact of hyperparameters on model behavior.

2.1.1 How it Works

- **Grid Search and Random Search:**

Grid search and random search are two commonly used hyperparameter tuning methods. Works in this area focus on analyzing the computational complexity, convergence properties, and effectiveness of these methods for optimizing different types of machine learning models, including SVMs.

- **Bayesian Optimization:**

Bayesian optimization is a probabilistic optimization technique that has gained popularity for hyperparameter tuning. Works in this area explore the theoretical foundations of Bayesian optimization, develop efficient algorithms, and investigate its applicability to SVMs and other machine learning algorithms.

- **Gradient-Based Optimization:**

Gradient-based optimization techniques have been proposed for hyperparameter tuning, leveraging the gradient information of the performance metric with respect to hyperparameters. Works in this area develop gradient-based optimization algorithms, analyze their convergence properties, and assess their performance in optimizing SVM models.

- **Automated Machine Learning (AutoML):**

Recent advances in AutoML have led to the development of automated hyperparameter tuning frameworks that can efficiently search for optimal hyperparameter configurations. Works in this area focus on developing AutoML systems, evaluating their performance, and integrating them with SVMs and other machine learning algorithms.

2.2 Feature Selection

Feature selection is a critical preprocessing step in machine learning, aimed at identifying the most informative features or variables that contribute to model performance. Numerous works have been conducted to develop feature selection techniques, explore their theoretical properties, and assess their effectiveness in various applications.

2.2.1 How it Works

- **Filter Methods:**

Filter methods assess the relevance of features independently of the chosen machine learning algorithm. Works in this area focus on developing statistical tests, correlation coefficients, and information-theoretic measures for feature ranking and selection. These works explore the mathematical foundations of filter methods and their impact on model performance.

- **Wrapper Methods:**

Wrapper methods evaluate feature subsets by training and testing the machine learning model on different combinations of features. Works in this area develop search strategies, such as forward selection, backward elimination, and recursive feature elimination, to identify the optimal subset based on model performance metrics. These works investigate the computational complexity and effectiveness of wrapper methods for feature selection.

- **Embedded Methods:**

Embedded methods incorporate feature selection directly into the model training process. Works in this area leverage regularization techniques, such as Lasso regression or decision tree pruning, to penalize irrelevant features during model training, effectively selecting the most important features automatically. These works explore the theoretical properties and practical implications of embedded methods for feature selection.

- **Advanced Techniques:**

Recent research has focused on developing advanced feature selection techniques, such as ensemble methods, deep learning-based approaches, and evolutionary algorithms. These works aim to address specific challenges, such as high-dimensional data, multicollinearity, and class imbalance, and improve the robustness and efficiency of feature selection in complex real-world scenarios.

2.3 Multicollinearity

Multicollinearity is a phenomenon in regression analysis where two or more predictor variables are highly correlated with each other, which can lead to issues such as instability in parameter estimates and difficulty in interpreting the coefficients of the variables.

2.3.1 VIF(Variance inflation factor)

Variance Inflation Factor (VIF) is a statistical measure used to assess multicollinearity in regression analysis. It quantifies how much the variance of an estimated regression coefficient is increased due to multicollinearity among predictor variables.

The VIF for each predictor variable is calculated by regressing that variable against all other predictor variables in the model. Mathematically, the VIF for the i th predictor variable is computed as:

$$VIF_i = \frac{1}{1 - R_i^2}$$

where R_i^2 is the coefficient of determination (R-squared) of the regression model with the i th predictor variable as the dependent variable and all other predictor variables as independent variables.

Interpreting VIF values:

- VIF values greater than 1 indicate multicollinearity among predictor variables.
- Typically, VIF values above 5 or 10 are considered problematic, indicating significant multicollinearity that may require remedial action.

Mitigating multicollinearity using VIF:

- High VIF values suggest that the variance of the estimated coefficients is inflated due to multicollinearity.
- To address multicollinearity, variables with high VIF values can be removed from the model, or multicollinearity can be reduced through feature selection, variable transformation, or regularization techniques such as ridge regression.

Overall, VIF provides a useful diagnostic tool for identifying multicollinearity in regression models, allowing analysts to take appropriate measures to improve model reliability and interpretation.

3 SVM in Statistical Process Control

Initially, SVMs were developed for supervised binary classifications between two separable classes (Vapnik 1995, 1998). We also need to identify two classes in the current problem. However, there are two key distinctions that set the current situation apart from typical binary categorization issues. Initially, the process data that has to be tracked isn't marked as a-priori. As a result, SVMs cannot be used to fit data and then make predictions directly. Some sort of semi-supervised approach that retains support vector use is required. Second, there is a significant imbalance in the frequencies of out-of-control states (high) compared to in-control states (low) in statistical process control (SPC). Consequently, the conventional method of dividing the two classes

One-class classification is the domain of machine learning algorithms such as Support Vector Data Description (SVDD). The purpose of outlier or anomaly detection, which is frequently employed, is to find occurrences that deviate noticeably from the bulk of the data. Creating a hypersphere—a sphere in a high-dimensional space—that encloses most of the data points is the primary goal of SVDD. The objective is to classify each occurrence that falls outside of the hypersphere as an outlier and the typical instances as part of the inliers.

Finding the hypersphere's center (\mathbf{O}) and radius (R) that has the smallest volume necessary to hold all of the sample data is therefore the challenge. The limitation is

$$(\mathbf{x}_i - \mathbf{O})^T(\mathbf{x}_i - \mathbf{O}) \leq R^2, i = 1, 2, \dots, l.$$

This optimization can also be solved using Lagrange Multiplier Method. The Lagrange multiplier function is

$$L(R, \mathbf{O}, \alpha_i) = R^2 - \sum_{i=1}^l [R^2 - (\mathbf{x}_i - \mathbf{O})^T(\mathbf{x}_i - \mathbf{O})]$$

with Lagrange multipliers, $\alpha_i \geq 0$. If the partial derivatives of L with respect to R and \mathbf{O} are set equal to zero, then

$$\sum_{i=1}^l \alpha_i = 1,$$

$$\mathbf{O} = \sum_{i=1}^l \alpha_i \mathbf{x}_i.$$

The solution to $\alpha_i, i = 1, 2, \dots, l$ can be determined using the quadratic programming technique to solve the following equivalent optimization problem:

$$Max \sum_{i=1}^l \alpha_i (\mathbf{x}_i \cdot \mathbf{x}_i) - \sum_{i=1, j=1}^l \alpha_i \alpha_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

with constraints $\alpha_i \geq 1$ and $\sum_{i=1}^l \alpha_i = 1$.

Most of the $\alpha_i, i = 1, 2, \dots, l$ are equal to zero. Only a small fraction is larger than zero. The sample points corresponding to the positive α_i , are called the support vectors and they are most informative in constructing the hypersphere.

To obtain the support vector based boundary, we may replace the inner product in equation (14) by the kernel functions are follows:

$$Max \sum_{i=1}^l \alpha_i \mathbf{K}(\mathbf{x}_i, \mathbf{x}_i) - \sum_{i=1, j=1}^l \alpha_i \alpha_j \mathbf{K}(\mathbf{x}_i, \mathbf{x}_j).$$

Here, $\mathbf{K}[\cdot]$ stands for kernel function.

SVM does hard classification on the samples, meaning it tells only whether the sample is from a given class or not. So, we cannot assign any error bars if there is any misclassification. Hence, keeping at focus this point we try to work the same problem using Relevance Vector Machines.

4 Applications of Machine Learning In SQC

4.1 SVM in SQC

Support Vector Machines (SVMs) are applied in SQC for various tasks, such as classification and anomaly detection. SVMs are particularly useful when dealing with complex data patterns and separating data points of different classes with a clear margin. For example, in quality control processes, SVMs can be used to classify products into different quality categories based on multiple input variables, helping to identify defects or anomalies in manufacturing processes.

4.2 Hyperparameter Tuning in SQC

Hyperparameter tuning is essential in SQC to optimize the performance of machine learning models, including SVMs. By fine-tuning hyperparameters such as the kernel type, regularization parameter, and kernel coefficient, practitioners can improve the accuracy and robustness of SVM models in SQC applications. Hyperparameter tuning ensures that SVMs are effectively tailored to the specific characteristics of the data and the requirements of the quality control process.

4.3 Feature Selection in SQC

Feature selection plays a crucial role in SQC by identifying the most relevant variables or features that contribute to the quality of products or processes. By selecting the most informative features, practitioners can simplify model complexity, reduce overfitting, and improve the interpretability of machine learning models, including SVMs. Feature selection techniques help SQC professionals focus on the key factors affecting quality and make more informed decisions to enhance process performance and product quality.

5 About the data

The document provides detailed information about the regression adjustment procedure and its application in monitoring multivariate processes:

Table 1: Dataset

num_of_obs	x1	x2	x3	x4	x5	x6	x7	x8	x9	y1	y2
1	12.78	0.15	91	56	1.54	7.38	1.75	5.89	1.11	951.5	87
2	14.97	0.1	90	49	1.54	7.14	1.71	5.91	1.109	952.2	88
3	15.43	0.07	90	41	1.47	7.33	1.64	5.92	1.104	952.3	86
4	14.95	0.12	89	43	1.54	7.21	1.93	5.71	1.103	951.8	89
5	16.17	0.1	83	42	1.67	7.23	1.86	5.63	1.103	952.3	86
6	17.25	0.07	84	54	1.49	7.15	1.68	5.8	1.099	952.2	91
7	16.57	0.12	89	61	1.64	7.23	1.82	5.88	1.096	950.2	99
8	19.31	0.08	99	60	1.46	7.74	1.69	6.13	1.092	950.5	10
9	18.75	0.04	99	52	1.89	7.57	2.02	6.27	1.084	950.6	10
10	16.99	0.09	98	57	1.66	7.51	1.82	6.38	1.086	949.8	10
11	18.2	0.13	98	49	1.66	7.27	1.92	6.3	1.089	951.2	98
12	16.2	0.16	97	52	2.16	7.21	2.34	6.07	1.089	950.6	96
13	14.72	0.12	82	61	1.49	7.33	1.72	6.01	1.092	948.9	93
14	14.42	0.13	81	63	1.16	7.5	1.5	6.11	1.094	951.7	91
15	11.02	0.1	83	56	1.56	7.14	1.73	6.14	1.102	951.5	91
16	9.82	0.1	86	53	1.26	7.32	1.54	6.15	1.112	951.3	93
17	11.41	0.12	87	49	1.29	7.22	1.57	6.13	1.114	952.9	91
18	14.74	0.1	81	42	1.55	7.17	1.77	6.28	1.114	953.3	94
19	14.5	0.08	84	53	1.57	7.23	1.69	6.28	1.109	953.9	96
20	14.71	0.09	89	46	1.45	7.23	1.67	6.12	1.108	952.6	94
21	15.26	0.13	91	47	1.74	7.28	1.98	6.19	1.105	952.3	99
22	17.3	0.12	95	47	1.57	7.18	1.86	6.06	1.098	952.6	95

Regression Adjustment Procedure The document explains that regression adjustment involves fitting regression models for specific output variables to the process input variables. The residuals obtained from these models are then used to construct univariate control charts for monitoring process stability

Autocorrelation Handling It is highlighted that regression adjustment is particularly useful for handling autocorrelation in data from chemical or process plants. By including the proper set of variables in the regression model, the residuals from the model are typically uncorrelated, even if the original variable exhibited correlation

Limitations of Hotelling T Control Chart The document discusses the limitations of the Hotelling T control chart for detecting mean shifts in multivariate processes. It emphasizes that while the Hotelling T chart is an optimal test statistic for certain hypotheses, it may not be optimal for detecting mean shifts in multivariate processes.

Data Tables and Control Charts The document includes data tables and control charts related to the cascade process data, such as individual values, moving ranges, and autocorrelation functions for residuals from regression on specific variables

6 Applying the methodology on a real multivariate data

After applying different Machine Learning concepts and techniques like Feature Selection, Multi-collinearity, Hyper parameter tuning we get the following outputs (Relevant code is written in the Annexure No. 9.2):

6.1 After dealing with Multi-collinearity and Feature Selection

Here we used multicollinearity and VIF (Variance Inflation Factor) to find the most relevant independent columns in a dataset. The goal was to reduce redundancy among predictor variables and improve the stability of regression models. After selecting the most relevant features, apply different types of kernels and calculate the Mean Squared Error (MSE) for two predictor variables, y1 and y2. This process helped identify the best-performing kernel and assess the predictive accuracy of the regression models for each variable. Here we get the output as follows:

Target Variable	Kernel Type	MSE	Number of Support Vectors	Target Variable	Kernel Type	MSE	Number of Support Vectors
0	y1 linear	0.010848	11	0	y1 linear	0.014472	11
1	y1 poly	0.007625	9	1	y1 poly	0.335098	10
2	y1 rbf	0.009067	8	2	y1 rbf	0.030610	10
3	y1 sigmoid	0.020529	12	3	y1 sigmoid	0.117103	14
4	y2 linear	0.128402	9	4	y2 linear	0.044265	12
5	y2 poly	0.144611	6	5	y2 poly	0.125483	9
6	y2 rbf	0.151488	4	6	y2 rbf	0.066443	10
7	y2 sigmoid	0.161386	4	7	y2 sigmoid	0.210803	14

(a)
(b)

Figure 1: Results with different kernels for individual predictor variables along with count of Support Vectors

6.2 After dealing with Hyper parameter Tuning

Here we conducted hyperparameter tuning to optimize the performance of machine learning models. Hyperparameters are settings that control the learning process, such as the learning rate or the number of trees in a random forest. By systematically adjusting these hyperparameters, I aimed to find the configuration that maximizes the model's predictive accuracy. This process involved testing different combinations of hyperparameters and evaluating their performance using techniques like cross-validation. Ultimately, hyperparameter tuning helped fine-tune the models for better performance on unseen data. Here we get the output as follows:

Target Variable	Kernel Type	MSE	Number of Support Vectors
0	y1 linear	0.010848	11
1	y1 poly	0.042243	15
2	y1 rbf	0.008343	17
3	y1 sigmoid	0.011843	10
4	y2 linear	0.184168	16
5	y2 poly	0.184236	17
6	y2 rbf	0.175966	16
7	y2 sigmoid	0.171625	16

Figure 2

7 Control Chart for Regression Prediction error for different types of kernels

7.1 Control Chart for Regression Prediction Y1

Irrespective of kernels there is no visible pattern in the Control Chart and no points go beyond the control limits UCL (upper control limit) and LCL (lower control limit). So, we can say based on every kernel the process is under control. However since MSE (after Hyper parameter Tuning) for most of the kernels i.e. for the kernels Linear, RBF (Radial Basis Function) , and Sigmoid are lower so we recommend using these kernels for the making of Control Chart for Regression Prediction Y1.(Relevant code is written in the Annexure No. 9.3)

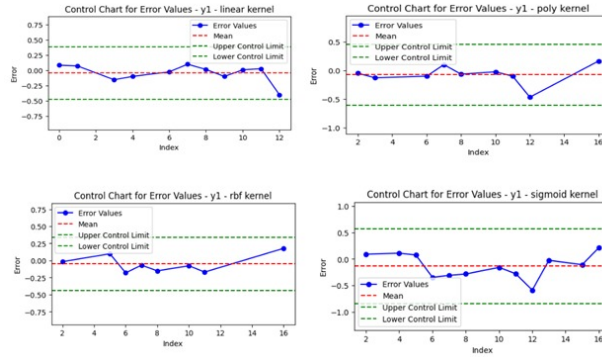


Figure 3

7.2 Control Chart for Regression Prediction Y2

Irrespective of kernels there is no visible pattern in the Control Chart and no points go beyond the control limits UCL (upper control limit) and LCL (lower control limit). So, we can say based on every kernel the process is under control. However, since MSE (after Feature Scaling) for most of the kernels i.e. for the kernels Linear, Polynomial and RBF (Radial Basis Function) are lower, so we recommend using these kernels for the making of Control Chart for Regression Prediction Y2.(Relevant code is written in the Annexure No. 9.3)

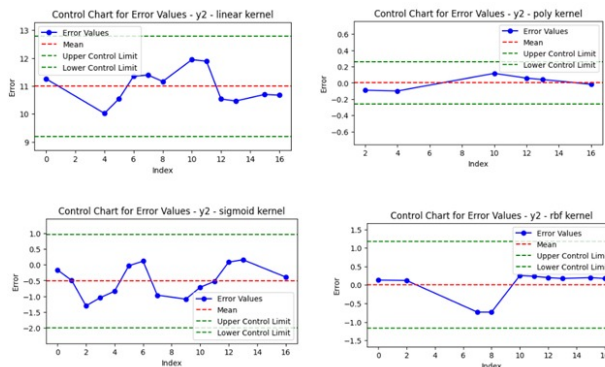


Figure 4

7.3 Control Chart for Combined Regression Prediction

Irrespective of kernels there is no visible pattern in the Control Chart and no points go beyond the control limits UCL (upper control limit) and LCL (lower control limit). So, we can say based on every kernel the process is under control and based on this we can say that this echos similar result as obtained for regression prediction for Y1 and Y2 individually.

Here since python don't provide option for inputting the dependent variables as vectors viz. Y1 and Y2, so we have to use some function (of the dependent variables Y1 and Y2) as dependent variable i.e. $Y = \text{Function}(Y1, Y2)$, where Y is the final dependent variable which take the value of both dependent variables Y1 and Y2 as input and generate final output based on the linear relationship of Y1 and Y2(linear combination).

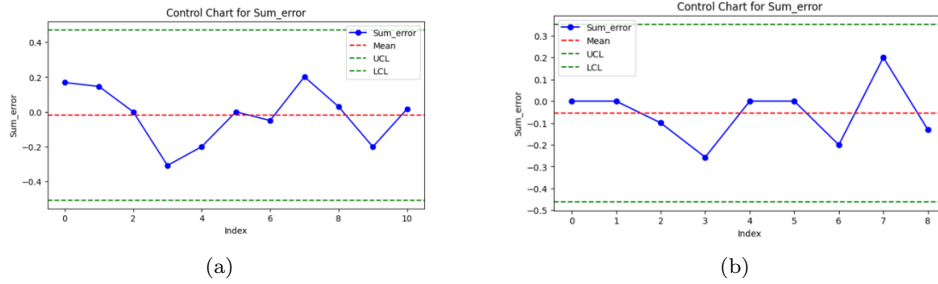


Figure 5: Combined results with different kernels: (a) Linear kernel, (b) Polynomial kernel

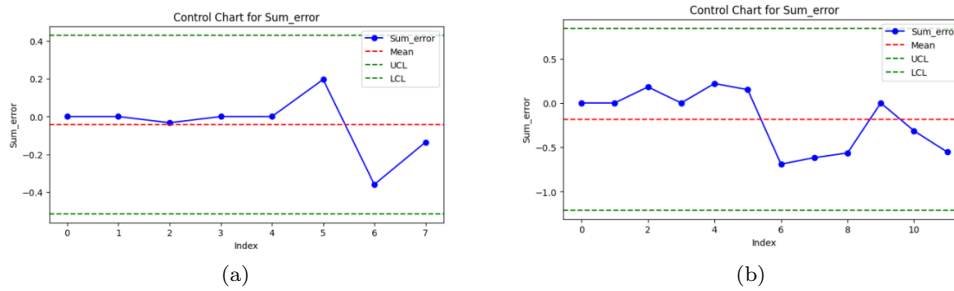


Figure 6: Combined results with different kernels: (c) RBF kernel, (d) Sigmoid kernel

Table 2: Combined MSE value for different types of kernels

Kernel	Without Data Pre-Processing	Feature Selection	Hyperparameter Tuning
Linear	236419.64	126231.54	148468.90
Polynomial	310538.34	185407.34	96568.20
RBF	194418.8845	185703.47	13450.90
Sigmoid	116736.23	107900.19	132723.17

But looking at the above table we see that MSE after Hyperparameter Tuning for the kernels Polynomial and RBF(Radial Basis Function) are lower than the MSE after Feature Scaling for the kernels Linear and Sigmoid, so we recommend using these kernels for the making of Control Chart for the final Regression Prediction Y. So the final Control Chart for the Combined regression Prediction Y will be

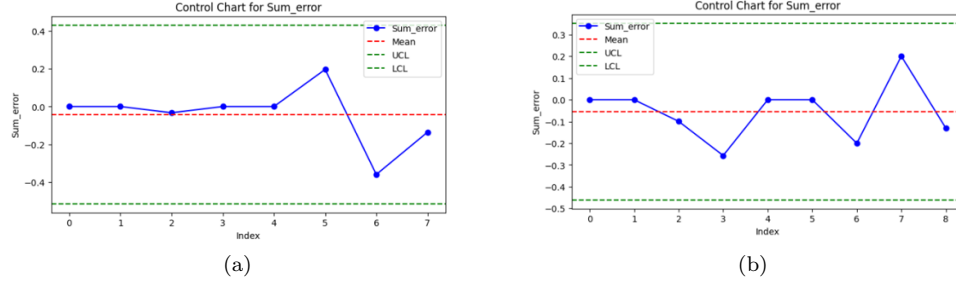


Figure 7: Control Chart for final Combined Regression Prediction with kernels: (m) RBF kernel, (n) Polynomial kernel

Though the MSE values with respect to different kernels are much larger so there is scope to further make another good model and this can be further investigated using various functions of Y_1 and Y_2 as dependent variables to further decrease the MSE values. (Relevant code is written in the Annexure No. 9.4)

8 Conclusion

Here we are dealing with multivariate data where there are two dependent variables (y_1, y_2) and 9 independent variables (x_1, x_2, \dots, x_9). In the book Introduction to statistical quality control, Douglas c. Montgomery uses only y_1 as response variable and accordingly form regression model and from that design the control chart. However since y_1 and y_2 pose as dependent variables for same of explanatory variables, the idea was to utilize both of them simultaneously for one single regression model and on the basis of that set up control chart to access overall health of the corresponding process.

Here instead of using OLS method we have use SVM with various kernels and made comparative performance analysis by individually consider both y_1 and y_2 response variable as well as consider both simultaneously. Since python don't provide option for inputting the dependent variables as vectors viz. y_1 and y_2 , so we have to use some function (of the dependent variables y_1 and y_2) as dependent variable i.e. $y = \text{Function}(y_1, y_2)$, where y is the final dependent variable which take the value of both dependent variables y_1 and y_2 as input and generate final output based on the linear relationship of y_1 and y_2 (linear combination). But we can further improvise and explore i.e. we can also consider the other combinations of y_1 and y_2 (instead of linear combination) to get better result.

We utilize both y_1 and y_2 simultaneously in SVR (Support Vector Regressor), and this utilizes data more than individual prediction models and the use of Support Vectors with different types of kernels made the model less dependent on OLS methodology. In this context, since the data shows linear tendency and hence the model that is in the book Introduction to statistical quality control, Douglas c. Montgomery, for multivariate linear regression (though for y_1 and y_2 individually) also perform good. However since our proposed method doesn't require Linera assumption, it can work for any type of inherent relationship (linear or non-linear).

References

1. Moez Farokhnia & S. T. A. Niaki (2020) Principal component analysis-based control charts using support vector machines for multivariate non-normal distributions, Communications in Statistics - Simulation and Computation, 49:7, 1815-1838, DOI: 10.1080/03610918.2018.1506032.
2. Introduction to Statistical Quality Control, Montgomery, Douglas C., edition=8, year=2020, John Wiley

& Sons, ISBN 1119723094.

3. Sun R & Tsung F (2003) A kernel-distance-based multivariate control chart using support vector methods, *International Journal of Production Research*, 41:13, 2975-2989, DOI: 10.1080/1352816031000075224.

9 Appendices

9.1 import libraries

```
#import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.feature_selection import VarianceThreshold, RFE
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor
import seaborn as sns
import matplotlib.pyplot as plt
from statsmodels.stats.outliers_influence import variance_inflation_factor
from scipy import stats
from sklearn.linear_model import LinearRegression

#import the data
from google.colab import files
# Prompt to upload a file
uploaded = files.upload()
# Specify the file name you uploaded
file_name = "data-desertation.xlsx" # Change to the actual file name
# Read the Excel file into a DataFrame
DF = pd.read_excel(file_name)
# Display the DataFrame
DF.head()
```

9.2 Data preprocessing

```
columns_to_drop = ['obs']
# Drop the specified columns
data_frame_filtered = DF.drop(columns=columns_to_drop)
DF = data_frame_filtered

# Assuming 'data_frame' is your DataFrame with predictor variables
correlation_matrix = DF.corr()

columns_to_drop = ['y1', 'y2']
# Drop the specified columns
data_frame_filtered = DF.drop(columns=columns_to_drop)
X = data_frame_filtered

# Function to calculate VIF for each variable
def calculate_vif(data):
    vif_data = pd.DataFrame()
    vif_data["Variable"] = data.columns
    vif_data["VIF"] =
    [variance_inflation_factor(data.values, i)
```

```

        for i in range(data.shape[1])

    return vif_data

# Calculate VIF for the initial set of predictor variables
vif_scores = calculate_vif(X)

# Print the VIF scores table
print("VIF Scores:")
print(vif_scores)

\subsection{Code for Splitting the data and fitting with RVM algorithm}
\begin{lstlisting}

columns_to_drop = ['x6', 'x9']
# Drop the specified columns
data_frame_filtered = DF.drop(columns=columns_to_drop)
X_DF = data_frame_filtered

# Function to calculate VIF for each variable
def calculate_vif(data):
    vif_data = pd.DataFrame()
    vif_data["Variable"] = data.columns
    vif_data["VIF"] = [variance_inflation_factor(data.values, i)
                        for i in range(data.shape[1])]
    return vif_data

# Calculate VIF for the initial set of predictor variables
vif_scores = calculate_vif(X)

# Print the VIF scores table
print("VIF Scores:")
print(vif_scores)

# Assuming 'data_frame' is your DataFrame with predictor variables
correlation_matrix = X_DF.corr()

columns_to_drop = ['x8', 'x7', 'x6', 'x9']
# Drop the specified columns
data_frame_filtered = DF.drop(columns=columns_to_drop)

# Function to calculate VIF for each variable
def calculate_vif(data):
    vif_data = pd.DataFrame()
    vif_data["Variable"] = data.columns
    vif_data["VIF"] = [variance_inflation_factor(data.values, i) for i in
range(data.shape[1])]
    return vif_data

# Calculate VIF for the initial set of predictor variables
vif_scores = calculate_vif(data_frame_filtered)

# Print the VIF scores table
print("VIF Scores:")
print(vif_scores)

```



```

#plot correlation matrix
# Assuming 'data_frame' is your DataFrame with predictor variables
correlation_matrix = data_frame_filtered.corr()

# Set up a mask to only show values that are >= 0.4 or <= -0.4
# mask = np.abs(correlation_matrix) >= 0.4
mask = correlation_matrix >= 0.4

# Create a heatmap with annotations and highlighting
plt.figure(figsize=(6, 4))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f", linewidths=.5,
            mask=~mask, vmin=-1, vmax=1,
            annot_kws={"size": 10},
            cbar_kws={"shrink": 0.8})

# Highlight cells based on correlation strength
sns.heatmap(correlation_matrix, annot=False, cmap="coolwarm", linewidths=.5,
            mask=mask & (np.abs(correlation_matrix) >= 0.6),
            vmin=-1, vmax=1, cbar=False, square=True, alpha=0.5)

sns.heatmap(correlation_matrix, annot=False, cmap="coolwarm", linewidths=.5,
            mask=mask & (np.abs(correlation_matrix) >= 0.7),
            vmin=-1, vmax=1, cbar=False, square=True, alpha=0.7)

sns.heatmap(correlation_matrix, annot=False, cmap="coolwarm", linewidths=.5,
            mask=mask & (np.abs(correlation_matrix) >= 0.8),
            vmin=-1, vmax=1, cbar=False, square=True, alpha=1)

plt.show()

df = data_frame_filtered

#Normalize the data
# Function to normalize a column using min-max scaling
def normalize_column(column):
    min_value = column.min()
    max_value = column.max()
    normalized_column = (column - min_value) / (max_value - min_value)
    return normalized_column

# Assuming 'data_frame' is your DataFrame with 8 columns
columns_to_drop = ['y1', 'y2']
data_filtered = df.drop(columns=columns_to_drop)

# Split the data into features (X) and target variable (y)
X = data_filtered

# Assuming 'data_frame' is your DataFrame with 8 columns
selected_columns_y1 = normalize_column(df['y1'])
selected_columns_y2 = normalize_column(df['y2'])

y_columns = {'y1': selected_columns_y1, 'y2': selected_columns_y2}

```

```

results_list = []

for target_column, y in y_columns.items():
    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test =
        train_test_split(X, y, test_size=0.2, random_state=42)

    kernels = ['linear', 'poly', 'rbf', 'sigmoid']

    for kernel_type in kernels:
        # Support Vector Regressor with different kernels
        svr = SVR(kernel=kernel_type)
        svr.fit(X_train, y_train)

        # Predict on the test set
        y_pred = svr.predict(X_test)

        # Calculate R-squared
        r2 = r2_score(y_test, y_pred)

        # Calculate adjusted R-squared
        n = len(X_test)
        p = X_test.shape[1]
        adjusted_r2 = 1 - ((1 - r2) * (n - 1) / (n - p - 1))

        # Calculate Mean Squared Error (MSE)
        mse = mean_squared_error(y_test, y_pred)

        # Calculate the difference between R-squared and adjusted R-squared
        difference = abs(r2 - adjusted_r2)

        # Append results to the list
        results_list.append({
            'Target Variable': target_column,
            'Kernel Type': kernel_type,
            # 'R-squared': r2,
            # 'Adjusted R-squared': adjusted_r2,
            'MSE': mse,
            # 'R-Difference': difference,
            'Number of Support Vectors': svr.support_.size
        })

    ## Check for correlations among features
    # correlation_matrix = X_train.corr()

    ## Display a heatmap of the correlation matrix
    # sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
    # plt.title(f"Correlation Matrix - {target_column} - {kernel_type} kernel")
    # plt.show()

# Convert the list of dictionaries to a DataFrame
results_df = pd.DataFrame(results_list)

# Display the results DataFrame

```

```

print(results_df)

# Plot the control chart
def control_chart(errors, target_variable, kernel_type):
    mean_error = errors.mean()
    std_error = errors.std()

    upper_limit = mean_error + 3 * std_error
    lower_limit = mean_error - 3 * std_error

    plt.figure(figsize=(6, 3))
    plt.plot(errors, marker='o', linestyle='-', color='b', label='Error Values')
    plt.axhline(mean_error, color='r', linestyle='--', label='Mean')
    plt.axhline(upper_limit, color='g', linestyle='--', label='Upper Control Limit')
    plt.axhline(lower_limit, color='g', linestyle='--', label='Lower Control Limit')

    # Adjust the y-axis limits to make the lower control limit more visible
    plt.ylim(lower_limit - 0.5, upper_limit + 0.5)

    plt.title('Control Chart for Error Values - {} - {} kernel'.format(target_variable,
        kernel_type))
    plt.xlabel('Index')
    plt.ylabel('Error')
    plt.legend()
    plt.show()

```

9.3 Code for generate tabels of the error values and number of support vectors for the two response variables Y1 and Y2 for different types of kernels

```

# Assuming 'data_filtered' is your DataFrame with features
X = data_filtered

# Assuming 'df' is your DataFrame with the target variables 'y1' and 'y2'
target_variables = ['y1', 'y2']

# Create an empty dictionary to store DataFrames
df_dict = {}

# Iterate over target variables
for target_variable in target_variables:
    y = normalize_column(df[target_variable])

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
        random_state=42)

    # Define different kernel types
    kernel_types = ['linear', 'poly', 'rbf', 'sigmoid']

    # Iterate over kernel types
    for kernel_type in kernel_types:
        # Support Vector Regressor with different kernels

```

```

svr = SVR(kernel=kernel_type)
svr.fit(X_train, y_train)

# Predict on the test set
y_pred = svr.predict(X_test)

# Calculate adjusted R-squared
r2 = r2_score(y_test, y_pred)
n = len(X_test)
p = X_test.shape[1]
adjusted_r2 = 1 - ((1 - r2) * (n - 1) / (n - p - 1))

# Print results for the current model
print("\nModel for {} with {} kernel:".format(target_variable, kernel_type))
print("Adjusted R-squared:", adjusted_r2)
print("Number of Support Vectors:", svr.support_.size)

# Print the respective support vectors from the DataFrame
support_vectors_indices = svr.support_
support_vectors_data = X.iloc[support_vectors_indices]
support_vectors_y_pred = svr.predict(support_vectors_data)

# Calculate the error between target variable and predicted values
error = y.iloc[support_vectors_indices] - support_vectors_y_pred

# Create a Q-Q plot for errors
plt.figure(figsize=(4, 2))
stats.probplot(error, dist="norm", plot=plt)
plt.title('Q-Q Plot for Errors ({} - {})'.format(target_variable, kernel_type))
plt.show()

# Create a new DataFrame to display the results
results_df = pd.DataFrame(data=support_vectors_data.values, columns=X.columns)
results_df[target_variable] = df.iloc[support_vectors_indices][target_variable]
results_df['{}_pred'.format(target_variable)] = support_vectors_y_pred
results_df['error_{}'.format(target_variable)] = error
# results_df.fillna(0, inplace=True)

# Store the DataFrame in the dictionary with a unique name
table_name = f'results_df-{{target_variable}}-{{kernel_type}}'
df_dict[table_name] = results_df

# Plotting predicted vs actual with a regression straight line and the scatter
plot
plt.figure(figsize=(4, 2))
plt.scatter(y_test, y_pred, alpha=0.5, label='Actual vs Predicted')

# Linear Regression for regression line
regression_model = LinearRegression()
regression_model.fit(y_test.values.reshape(-1, 1), y_pred)
regression_line = regression_model.predict(y_test.values.reshape(-1, 1))

# Plot the regression line
plt.plot(y_test, regression_line, color='red', linestyle='--', linewidth=2,

```

```

        label='Regression Line')

plt.title(f'Predicted vs Actual for {target_variable} ({kernel_type} kernel)')
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.legend()
plt.show()

print("\nRespective Support Vectors with {}, {} Predictions, and
      Error:".format(target_variable, target_variable))
results_df.fillna(0, inplace=True)
print(results_df)

# Plot control chart for the error values
control_chart(error, target_variable, kernel_type)

# Access individual DataFrames from the dictionary
# For example, to access the DataFrame for y1 with linear kernel:
selected_df_1= df_dict['results_df_y1_linear']
selected_df_2= df_dict['results_df_y2_linear']
selected_df_3= df_dict['results_df_y1_poly']
selected_df_4= df_dict['results_df_y2_poly']
selected_df_5= df_dict['results_df_y1_rbf']
selected_df_6= df_dict['results_df_y2_rbf']
selected_df_7= df_dict['results_df_y1_sigmoid']
selected_df_8= df_dict['results_df_y2_sigmoid']

col_to_add = ['y2', 'y2_pred', 'error_y2']
selected_linear = selected_df_2[col_to_add]
result_tab = pd.concat([selected_df_1, selected_linear], axis = 1)
result_tab.fillna(0, inplace=True)
# print(result_tab)

result_tab = result_tab[['y1', 'y2', 'y1_pred', 'y2_pred', 'error_y1', 'error_y2']]

# Add two columns and store the result in a new column
result_tab['Sum_actual'] = result_tab['y1'] + result_tab['y2']
result_tab['Sum_pred'] = result_tab['y1_pred'] + result_tab['y2_pred']
result_tab['Sum_error'] = result_tab['error_y1'] + result_tab['error_y2']
print(result_tab)

```

9.4 Code for generate control chart using the combined error value of the two response variables Y1 and Y2 for different types of kernels

```

# Assuming 'df' is your DataFrame
sum_error_column = result_tab['Sum_error']

# Create a Q-Q plot for 'Sum_error'
plt.figure(figsize=(6, 4))
stats.probplot(sum_error_column, dist="norm", plot=plt)
plt.title('Q-Q Plot for Sum_error')
plt.show()

```

```

# Assuming 'df' is your DataFrame
sum_error_column = result_tab['Sum_error']

# Calculate UCL and LCL for the control chart
mean = sum_error_column.mean()
std_dev = sum_error_column.std()
UCL = mean + 3 * std_dev # You can adjust the multiplier as needed
LCL = mean - 3 * std_dev # You can adjust the multiplier as needed

# Create a control chart for 'Sum_error' with UCL and LCL
plt.figure(figsize=(8, 4))
plt.plot(result_tab.index, sum_error_column, marker='o', linestyle='-',
         color='b', label='Sum_error')
plt.axhline(y=mean, color='r', linestyle='--', label='Mean')
plt.axhline(y=UCL, color='g', linestyle='--', label='UCL')
plt.axhline(y=LCL, color='g', linestyle='--', label='LCL')
plt.title('Control Chart for Sum_error')
plt.xlabel('Index')
plt.ylabel('Sum_error')
plt.legend()
plt.show()

from sklearn.linear_model import LinearRegression

# Assuming 'df' is your DataFrame
X_regression = result_tab[['Sum_actual']]
y_regression = result_tab['Sum_pred']

# Create a Linear Regression model
regression_model = LinearRegression()
regression_model.fit(X_regression, y_regression)

# Get the regression line
regression_line = regression_model.predict(X_regression)

# Plot the regression line
plt.figure(figsize=(6, 4))
plt.scatter(result_tab['Sum_actual'], result_tab['Sum_pred'], alpha=0.5)
plt.plot(result_tab['Sum_actual'], regression_line, color='red', linestyle='--',
         linewidth=2)
plt.title('Regression Line for Sum_actual vs Sum_pred')
plt.xlabel('Sum_actual')
plt.ylabel('Sum_pred')
plt.show()

```

9.5 Code for finding the optimal combined prediction

```

# Evaluate the performance of the combined linear predictions
mse_combined_linear = mean_squared_error(result_tab['Sum_actual'],
                                         result_tab['Sum_pred'])

```

```

predictions_y1 = result_tab['y1-pred']
predictions_y2 = result_tab['y2-pred']

from scipy.optimize import minimize

# Define the objective function for optimization
def objective_function(weights, predictions_y1, predictions_y2):
    combined_predictions_optimal = weights[0] * predictions_y1 +
        weights[1] * predictions_y2
    return mean_squared_error(result_tab['Sum-actual'], combined_predictions_optimal)

# Perform optimization to find the optimal weights
result = minimize(objective_function, x0=[0.5, 0.5], args=(predictions_y1,
    predictions_y2))

# Get the optimized weights
optimal_weights = result.x

# Combine the predictions using the optimal weights
combined_predictions_optimal = optimal_weights[0] * predictions_y1 +
    optimal_weights[1] * predictions_y2

#
mse_combined_optimal = mean_squared_error(result_tab['Sum-actual'],
    combined_predictions_optimal)

print("Mean Squared Error (Combined Linear):", mse_combined_linear)
print("Optimal Weights:", optimal_weights)
print("Mean Squared Error (Combined Optimal):", mse_combined_optimal)

# Calculate R-squared
TSS = np.sum((result_tab['Sum-actual'] - np.mean(result_tab['Sum-actual']))**2)
RSS_linear = np.sum((result_tab['Sum-actual'] - result_tab['Sum-pred'])**2)
R_squared_linear = 1 - (RSS_linear / TSS)

# Calculate adjusted R-squared for the linear combination
n = len(result_tab['Sum-actual'])
p_linear = 2 # Number of predictors in the linear combination model
adjusted_R_squared_linear = 1 - ((1 - R_squared_linear) * (n - 1) / (n - p_linear - 1))

# Calculate R-squared for the optimal combination
RSS_optimal = np.sum((result_tab['Sum-actual'] - combined_predictions_optimal)**2)
R_squared_optimal = 1 - (RSS_optimal / TSS)

# Calculate adjusted R-squared for the optimal combination
p_optimal = 2 # Number of predictors in the optimal combination model
adjusted_R_squared_optimal = 1 - ((1 - R_squared_optimal) *
    (n - 1) / (n - p_optimal - 1))

# Print the results
print("R-squared (Linear):", R_squared_linear)
print("Adjusted R-squared (Linear):", adjusted_R_squared_linear)

```

```
print("R-squared (Optimal):", R_squared_optimal)
print("Adjusted R-squared (Optimal):", adjusted_R_squared_optimal)
```