The University Of Sheffield.

Multisensor Decision Systems

ACS6124

Assignment Code ACS6124-001

By

Ankur Tiwari

Registration Number 210158775

Contents

Introduction

The objective of this report is to showcase how the performance of a multi-sensor system can be improved by reducing uncertainty, improving robustness and reliability by deploying statistical decision-making tools like Bayesian estimation. The report has been split into two parts, The first part deals with the problem of condition monitoring and fault diagnosis of a rotary machine by extracting features from a set of vibration signals, and then classify them into different clusters using KNN algorithm. The second part deals with multisensor signal estimation using different Bayesian estimators.

Problem-I-a: The steps involved are:

Spectral Analysis of the signals

Spectral analysis refers to the techniques of estimating power content of the frequency components of a signal. understanding the frequency dependence can yield meaningful information. **In a time, domain signal, the amplitude of this signal varies with respect to time but by just by looking at the time domain representation of this signal there is no easy way to figure out the frequency components present in the signal** so this signal can be converted into frequency domain by using the FFT (Fast Fourier Transform) but we use Welch's method for estimating power spectral Density.

Using FFT, we obtain Magnitude response and the Phase response. Magnitude response tells us about strength of frequency component and Phase response tells how these frequency components align in time. The output of the FFT is complex valued whereas the input is real valued. To deal with complex numbers present in the output of FFT we must compute the magnitude and phase using different functions. We also must compute frequency vector and deal with new positive and negative frequencies, even after going through all these steps we still do not have the signal power information. Moreover, **FFT is inefficient for describing non-stationary signals introduced by faults in vibration signals.**

Welch function **pwelch** plots the signal power density as a function of frequency, it uses the Welch method to compute the power spectral density estimate. **It is an ideal**

function to use when you have little information about the spectral content of the signal. pwelch plots one-sided power spectral density of the signal by default what this means is we don't have to worry about positive and negative frequencies.

<u>Note: Matlab code for spectral Analysis is given in Appendix 1</u>.

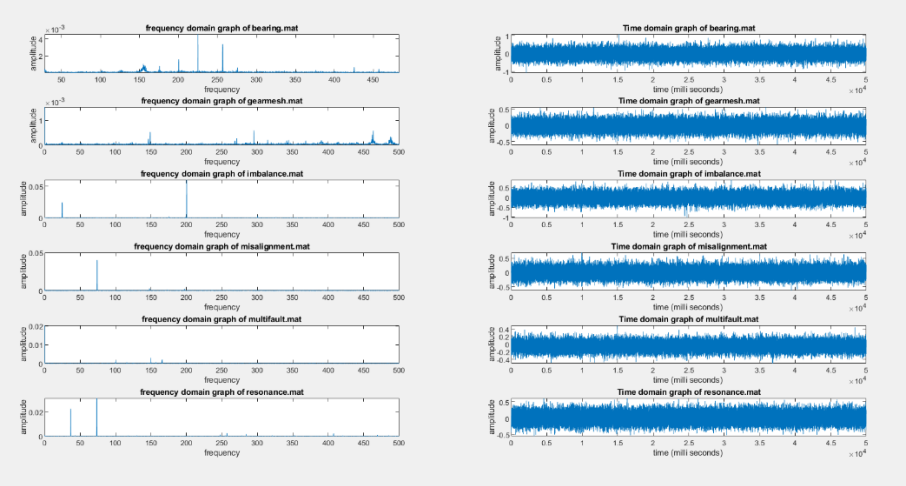


Figure 1 Comparison between frequency domain and time domain plot for various faults

Feature Extraction

Here we extract appropriate features from the vibration data indicative of the fault Condition, for this we consider the energy levels given by root mean square (RMS) of the signal. Before segregating these signals in different frequency band, we normalize the signals. **Normalization** is done to bring all the signals to same scale.

Butterworth filter is a filter used in signal processing and is designed to have a frequency response as flat as possible in the given passband. This is the reason it's also known as "maximally flat magnitude filter". Passband cand be defined as the range of frequency that can pass through the filter. The order of filter decides the rate of roll-off-response(slope) of the filter. Higher Order corresponds to more idealistic response. Vertical line represents an ideal response and any frequency surpassing this line would be cutoff. The generalized form of frequency response of n^{th}-order Butterworth low-pass filter is [1]

$$H(j\omega) = \frac{1}{\sqrt{1+\varepsilon^2\left(\omega/\omega_c\right)^{2^n}}}$$

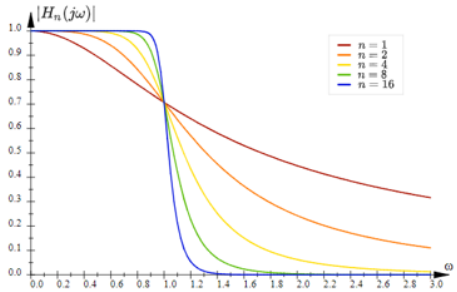


Figure 2 Slope of Butterworth filter for different orders

Note: The code for feature extraction is given in Appendix 2

Data Visualization

After feature extraction we have a set of five feature matrices and each one is composed of six features of signal energy level in different frequency bands. To visualize these features with six-dimension, six-dimensional feature vectors are mapped to two dimensions using principal component analysis (PCA). PCA is a dimensionality reduction technique, PCA seeks out lines, planes, and hyper-planes in K-dimensional space that best represent the data. This done by calculating Eigen vectors. Eigen Vectors are those axes that does not change direction in a transformation. These vectors are the least square approximation of data points that cluster together. [2]

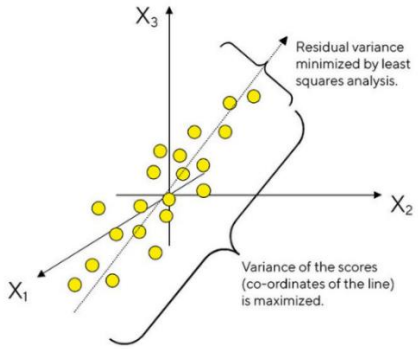


Figure 3 Visualization of data that minimizes residual variance in least square sense

Critical comparison with results obtained from Lab A.I.

- When we compare the results obtained from Lab A.I with those obtained from assignment it is clearly visible that the clusters have changed position this is because we have used different features and because of that the 6 dimensional space formed is different from 4 dimensional space , when we try to find the eigen vectors of a six-dimensional space we might have obtained eigen vectors pointing in different directions as compared to those obtained for a four dimensional space, which results in the change of position of the clusters in two dimensional space.
- In both the images 2 clusters are intermingling together this is because of some correlation between them. Which can be justified in some way because the red cluster represent bearing fault whereas black cluster represents imbalance fault so normally when the bearings are not properly aligned, they produced imbalance in the system. Which can probably result in similar frequency for both the signals.
- The clusters obtained for Lab A.I are more spread out when compared to those obtained for the assignment, the reason behind this is that in lab we were using only 3 filters and one of the dimensions was representing the rms value, so frequency space was more scattered as it was divided into just 3 bands. However, for assignment the same frequency range has been subdivided into 6 bands which results in less bandwidth being provided to each filter resulting in more compact clustering.

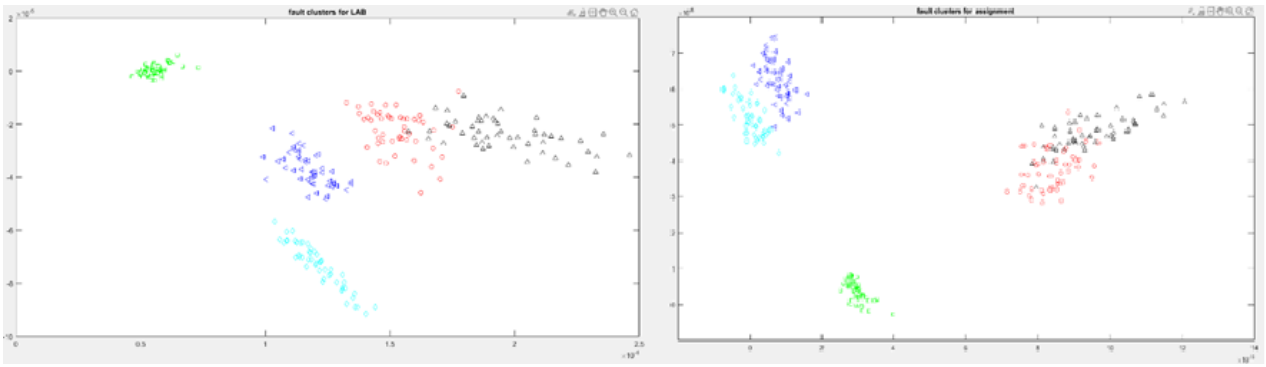


Figure 4 Figure Showing a comparison between clusters for Lab Task and assignment task

Problem-1-b

In this task machine condition is classified based on k-nearest neighbour algorithm. The steps involved are, splitting the dataset into two sets training data and test data. This is done in a ratio of 70:30. Training data is used as reference set of known fault types and based on this reference unknown sample of test data is assigned a class of its nearest sample. The Euclidean distance between each fault point is calculated then classifiers are assigned to each fault type and these classifiers are allotted to the test examples based on their proximity to nearest fault types. After this accuracy is calculated based on number of test set examples correctly classified, we calculate the classification accuracy.

Fault classification analysis results

The results obtained on application of **kNN** are shown in table below:

k-value	**Classification accuracy**
1	98.667
3	98.667
5	98.667
7	97.33
30	96
51	90.667

Critical Evaluation:

- Generally, the value of k is taken to be odd because its easier to find out mode of odd values.
- As it is visible the classification accuracy obtained for k=1,3,5 is quiet high this is since the fault clusters are quite compact, distinct, and far from each other. So, all the fault point are mostly near similar fault point. However, there are two clusters that are mixing. But if we observe closely although they are mixing the in most cases the nearest neighbour of a fault point data is similar fault type, so we achieve quiet good accuracy apart from those points that are but in blue boxes as shown in figure 6.
- When we increase the value of k more than 5 say 7 now the mode of 7 nearest fault types is taken in this case, we see a decrease in accuracy because some points in these 2 clusters are nearer to different category as compared to its own category therefore,

we see a drop in accuracy. As we go on increasing the value of k we see a further drop in Accuracy.

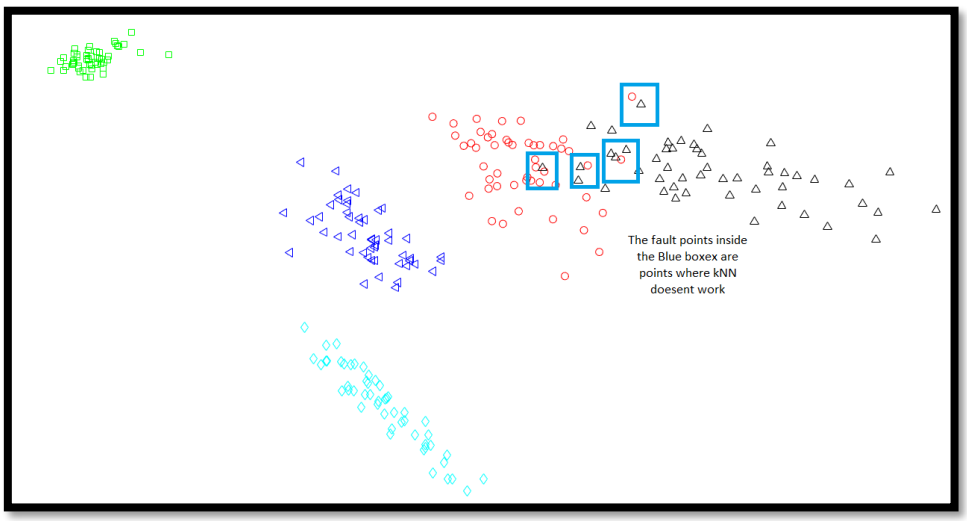


Figure 5 Figure showing data points of different types of faults that are clustering togather and resulting in accuracy reduction

Alternative approaches:

There are several approaches that can be used for fault classification apart from kNN . These approaches can be classified as Model based and Data-driven. These approaches have been discussed below with a detailed comparison of their advantages and disadvantages.

SVM (Support Vector Machines) identifies a hyperplane in an N-dimensional space that clearly classifies the data points. There are numerous alternatives to choose a hyperplane that could separate two classes of data points. The objective is to find a plane that has maximum gap from the data points of both classes.

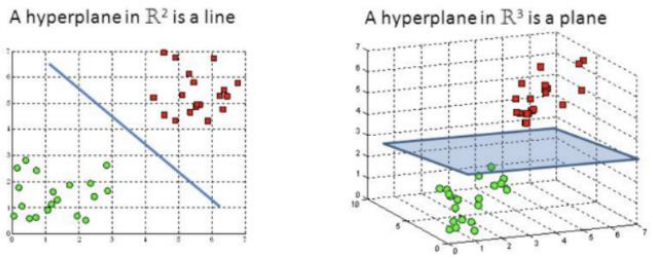


Figure 6 SVM hyper-planes in 2D and 3D space

NNPR (Neural Network Pattern Recognition): It most advantageous part of using neural networks for pattern recognition is that it generalizes well to every situation and mostly training, and testing performance are the same. It's easier to find parameters for NNPR. They are helpful in evaluating static patterns of each feature. Moreover, a variety of activation functions can be used based on the requirements.

ANFIS (Adaptive Network Based Fuzzy Inference System) has the network structure of a neural network, but it uses a hybrid learning procedure. It can assemble an input-output mapping attained from human knowledge and stipulated input-output data pairs. It can predict at an accuracy almost equivalent to that of a neural network. However, training is much faster.

LVQ (Learning Vector Quantization) Networks function like SVM's. The regions of different classes are divided by hyperplanes and are called Voronoi partitions. The number of classes are usually fixed because of this the training cost is lower than SVM's

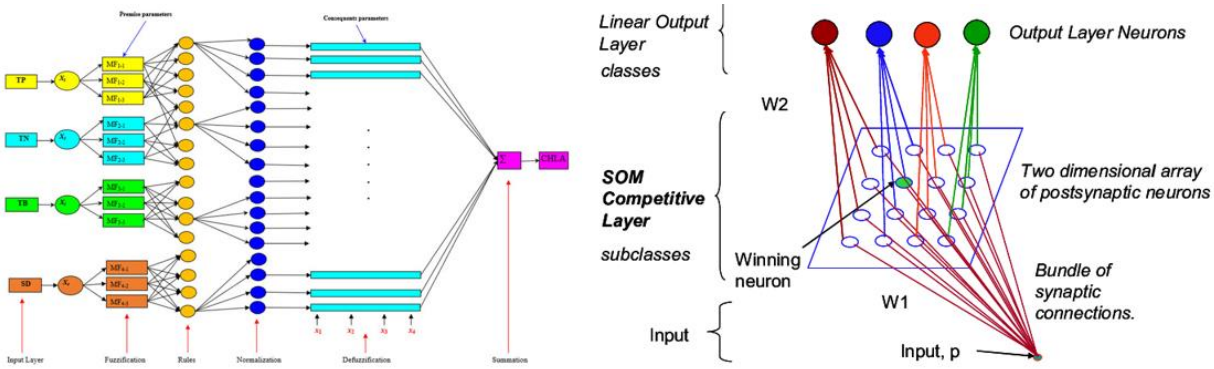


Figure 7 ANFIS Network & LVQ networks

A detailed comparison between these Classifier is given below:

Classifier	Advantages	Disadvantages
SVM	• The generalization for out-of-sample is good. • There is a flexibility in the choice for separating threshold. • Can deal with smaller training datasets with large number of inputs	• Lacks Transparency in results. • Algorithm is complex and requires a lot of memory. • Training process is time consuming.
NNPR	• Ability to model complicated relationships • Can work with incomplete and noisy dataset. • Fault detection rate is good • Can estimate all posterior probabilities based on predictor variable interactions.	• Training process is slow • Sometimes overfitting of data is observed. • Data requirement for training is large. • Exploration capability is low. • Inter repeatability of learned information is low.
LVQ	• Can be easily implemented for multi class problems • Computationally less expensive • Algorithm complexity can be adjusted as per requirement	• Sensitive to initial data • Neurons are not utilized properly • Accuracy low with high dimensional data
ANFIS	• Convergence during training is fast • Tolerance to uncertainty is high • Generalization, Interpolation and extrapolation capability is good	• Demands High computational capability • Limitations with number of inputs. • Interpretability of learned information is low.

Problem-II: Problem Statement

Design a multisensor signal estimation and health monitoring system for blade pitching mechanism for a wind turbine manufacturing company. The data used for this purpose is obtained from a rotary encoder connected to blade bearing and sensor noise is

$$\nu = N(\mu, \sigma^2) \quad where \;\; \mu = 0, \sigma^2 = 9$$

Problem-II-a :Calculate MMSE estimator, given prior knowledge of estimator is uniformly distributed

range $0^O \leq \omega \leq 30^0$

MMSE Estimator determined by conditional PDF is given by following equations:

Consider a Bayesian estimator x* with prior knowledge formulation:

Bayesian(x*) = $E[(x - x^*)^2] = \int \int (x - x^*)^2 p(Y_{1:T}, x)\, dy_{1:T}\, dx$ where $Y_{1:T}$ is the vector of observations.

We want to determine estimator x* that minimizes the MSE

Equation 1 Equation for MMSE Estimator

$$x^* = E[x|y_{1:T}] = \int_{-\infty}^{\infty} x p(x|y_{1:T})\, dx = \frac{\int_{-\Delta}^{\Delta} \frac{x}{\sqrt{2\pi \frac{\sigma^2}{T}}} \exp\left[-\frac{1}{2\frac{\sigma^2}{T}}\left(x - \frac{1}{T}\sum_{t=1}^{T} Y_t\right)^2\right] dx}{\int_{-\Delta}^{\Delta} \frac{1}{\sqrt{2\Pi \frac{\sigma^2}{T}}} \exp\left[\frac{-1}{2\frac{\sigma^2}{T}}\left(x - \frac{1}{T}\Sigma_{t=1}^{T} y_t\right)\right] dx}$$

Case 1: Entire measurement vector "encoder.mat" is provided

Where the data given to us is:

$\frac{1}{T}\Sigma_{t=1}^{T} y_t \quad represents\ the\ means\ of\ sensor\ data$

T is the number of sensors reading available after measurement = 100 for case 1 and 5 for case 2. $[-\Delta, \Delta] = [0^0, 30^0] \quad \sigma^2 = 9$

```
1. encoder_data=load('encoder.mat');          % loading encoder data
2. encoder_data=encoder_data.encoder;         % changing structure into a vector
3. T=length(encoder_data);                    % determining the length of vector
4. sensor_noise=9;                            % giving the value of sigma square
5. numerator_fun=@(x)  ((x/sqrt(2*pi*(sensor_noise/T))).*exp((-1/(2*sensor_noise/T)).*(x-
   mean(encoder_data)).^2));                  % writing numerator part of the equation 1
6. denominator_fun=@(x)  ((1/sqrt(2*pi*(sensor_noise/T))).*exp((-1/(2*sensor_noise/T)).*(x-
   mean(encoder_data)).^2));                  % denominator part of equation 1
7. numerator = integral(numerator_fun,0,30); % integrating the numerator separately
8. denominator=integral(denominator_fun,0,30); % integrating the denominator separately
9. x_star=numerator/denominator; % finding the value of x*
10. disp(x_star)                              %printing the value of x*
```

Case 2: Only five first element of the provided measurement vector is available.

```
1. encoder_data=load('encoder.mat');                    % loading encoder data
2. encoder_data=encoder_data.encoder;                   % changing structure into a vector
3. T=length(encoder_data);                              % determining the length of vector
4. sensor_noise=9;                                      % giving the value of sigma square
5. numerator_fun=@(x)  ((x/sqrt(2*pi*(sensor_noise/T))).*exp((-1/(2*sensor_noise/T)).*(x-
   mean(encoder_data(1:5))).^2));          % writing numerator part of the equation 1
6. denominator_fun=@(x)  ((1/sqrt(2*pi*(sensor_noise/T))).*exp((-1/(2*sensor_noise/T)).*(x-
   mean(encoder_data(1:5))).^2));          % denominator part of equation 1
7. numerator = integral(numerator_fun,0,30);            % integrating the numerator separately
8. denominator=integral(denominator_fun,0,30); % integrating the denominator seprately
9. x_star=numerator/denominator;                % finding the value of x*
10. disp(x_star)                                %printing the value of x*
```

Results:

Case 1: Entire measurement vector	20.313
Case 2: First five values	17.3125

Critical evaluation:

- If $\frac{\sigma^2}{T} \to 0$, then $x^* = \hat{x}$ (average of measurements) which implies that as the number of readings is more the estimator tends to incline more towards data and less towards prior knowledge which is quite evident here as $\frac{\sigma^2}{T} = \frac{9}{100} = 0.09$ so if we find out the mean of the data which also happens to be 20.3131 proving it to be right.
- MMSE estimator converges distribution to normal distribution

Problem-II-b : MMSE Estimator when the prior knowledge is Gaussian distributed

Given, sensor noise is $v = N(\mu, \sigma^2) \quad where\ \mu = 0, \sigma^2 = 9$

Mean of prior data $\mu_x = 15$ variance of prior data $\sigma_x^2 = 4$ and

T=number of readings =100 for case 1 and 5 for case 2

MMSE estimator is given by

$$E[x|Y_{1:T}] = \mu_{X|Y_{1:T}} = \frac{\sigma_X^2}{\sigma_X^2+\frac{\sigma^2}{T}}\left(\frac{1}{T}\sum_{t=1}^{T} y_t\right)(data\ part) + \frac{\frac{\sigma^2}{T}}{\sigma_X^2+\frac{\sigma^2}{T}}\mu_X(prior\ part)$$

Case 1: Entire measurement vector "encoder.mat" is provided

```
1. encoder_data=load("encoder.mat");                    % loading encoder data
2. encoder_data=encoder_data.encoder;                   % changing structure into a vector
3. T=length(encoder_data);                              % finding number of readings in vector
4. sensor_noise=9;
5. variance_of_prior_data=4;
6. mean_of_prior_data=15;
7. data_part=((variance_of_prior_data)/((variance_of_prior_data)+(sensor_noise/T))).*
   mean(encoder_data);
8. priori_part=((sensor_noise/T)/((variance_of_prior_data)+(sensor_noise/T))).*mean_
   of_prior_data;
9. MSME_Estimator=data_part+priori_part;
10. disp(MSME_Estimator)
```

Case 2: Only five first element of the provided measurement vector are available

```
1. encoder_data=load("encoder.mat"); % loading encoder data
2. encoder_data=encoder_data.encoder; % changing structure into a vector
3. T=5;          % taking number of readings = 5
4. sensor_noise=9;
5. variance_of_prior_data=4;
6. mean_of_prior_data=15;
7. data_part=((variance_of_prior_data)/((variance_of_prior_data)+(sensor_noise/T))).*
   mean(encoder_data(1:T));
8. priori_part=((sensor_noise/T)/((variance_of_prior_data)+(sensor_noise/T))).*mean_
   of_prior_data;
9. MSME_Estimator=data_part+priori_part;
10. disp(MSME_Estimator)
```

Result:

Case 1: Entire measurement vector	20.1962
Case 2: First five values	16.5948

Critical evaluation:

- When large amount of data is available: $\sigma_x^2 \gg \frac{\sigma^2}{T}$ which in this case $\sigma_x^2 = 4 \gg \frac{\sigma^2}{T} = \frac{9}{100} = 0.09$ then $E[x|Y_{1:T}] \approx \frac{1}{T}\sum_{t=1}^{T} y_t(mean)$ which is quite evident here as $20.1962 \approx 20.3131$.
- For the case where just 5 elements are taken the estimator becomes weighted average of mean of data and mean of prior data where the weights are $\frac{\sigma_X^2}{\sigma_X^2+\frac{\sigma^2}{T}}$ for mean of data and $\frac{\frac{\sigma^2}{T}}{\sigma_X^2+\frac{\sigma^2}{T}}$ for mean of prior data.
- When little data is available: say $\sigma_x^2 \ll \frac{\sigma^2}{T}$ taking the value of T=1 then 4 < 9 the $E[x|Y_1:T] \approx \mu_X$ in our case $E[x|Y_1:T] = 15.3539 \approx \mu_X\ (15)$.

Problem-II-c: Sequential MMSE Estimation with Gaussian Priors

Given, sensor noise is $v = N(\mu, \sigma^2) \quad where\ \mu = 0, \sigma^2 = 9$

Mean of prior data $\mu_x = 15$ variance of prior data $\sigma_x^2 = 4$ and

T=number of readings =100 for case 1 and 5 for case 2

The formulation for sequential MMSE is given by

$$\widehat{\boldsymbol{x}}_{\boldsymbol{T+1}} = \frac{\boldsymbol{\sigma_x^2}}{\boldsymbol{(T+1)\sigma_x^2+\sigma^2}} \sum_{\boldsymbol{t=1}}^{\boldsymbol{T+1}} \boldsymbol{y_t} + \frac{\frac{\boldsymbol{\sigma^2}}{\boldsymbol{T+1}}}{\boldsymbol{\sigma_x^2+\frac{\sigma^2}{T+1}}} \boldsymbol{\mu_x}$$

Case 1: Entire measurement vector "encoder.mat" is provided

```
clear all;
data = load("encoder.mat");
data = data.encoder
x = data(1:5,:);
sensor_noise = 9;
varience_sensor=4;
sensor_mean=15;
T = length(data) - 1;
% Taking each data in sequential form
for i = 1:T+1
    A(i) = sum(data(1:i,:));
    Prior_data = ((sensor_noise)/(varience_sensor*i + sensor_noise)) *sensor_mean;
    sensor_data = ((varience_sensor)/(varience_sensor*i + sensor_noise)) * A(i);
    sum_of_data=Prior_data+sensor_data;
end
disp(sum_of_data)
```

Case 2: Only five first element of the provided measurement vector are available

```
clear all;
data = load("encoder.mat");
data = data.encoder
x = data(1:5,:);
sensor_noise = 9;
varience_sensor=4;
sensor_mean=15;
T = length(data) - 1;
% Taking each data in sequential form
for i = 1:5
    A(i) = sum(data(1:i,:));
    Prior_data = ((sensor_noise)/(varience_sensor*i + sensor_noise)) *sensor_mean;
    sensor_data = ((varience_sensor)/(varience_sensor*i + sensor_noise)) * A(i);
    sum_of_data=Prior_data+sensor_data;
end
disp(sum_of_data)
```

Result:

Case 1: Entire measurement vector	20.1962
Case 2: First five values	16.5948

Critical evaluation:

- The main advantage of using sequential MMSE estimator is that se can access the data as soon as its available instead of storing it, which makes it particularly useful to use along with microcontrollers and real time applications.
- The results obtained are similar to Gaussian prior however the main disadvantage is that its more time consuming.

Problem-II-d: CUSUM Test Algorithm

A cumulative sum (CUSUM) chart is a tool for tracking tiny changes in the mean of a process. It makes use of the total sum of deviations from an objective. It's used to visualise the cumulative sum of deviations from a target for each subgroup's means.

Sensor estimations are gathered using a strain gauge sensor that detects the blade's vertical to rotating axis bending moment over k sample intervals. A two-sided CUSUM test on the assumption that the system's usual operation is 3000 KNm with a variance of 1. We have used a 20-point threshold and ignored the leaking term y.

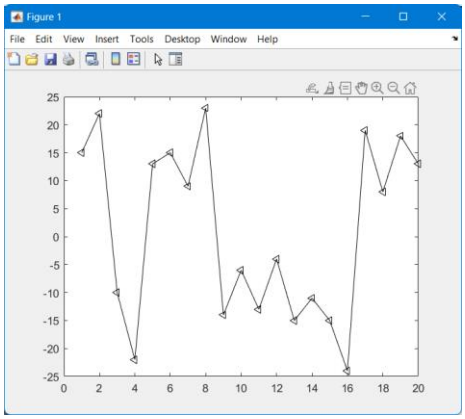
Code for CUSUM:

```
1.  data=load('straingauge.mat');           % loading data
2.  data=data.straingauge;                  % converting structure to vector
3.  normal_value = 3000;
4.  g(1) = (data(1) - normal_value);        % setting the initial value of g
5.  g_1(1) = g(1);
6.  for t=2:length(data)
7.      g(t) = g(t-1) + (data(t) - normal_value);
8.         g_1(t) = g(t);                     %value stored for plotting
9.          if g(t) <= -20                  %comparing for -ve Threshold
10.            fprintf("Negative Threshold reached at %d and its value is:%d\n",t,g(t));
11.            g_1(t) = g(t);               %value stored for plotting for plotting
12.            g(t) = 0;                    %resolvig error
13.         end
14.         if g(t) >= 20                   % comparing for +ve Threshold
15.            fprintf("Negative Threshold reached at %d and its value is:%d\n",t,g(t));
16.            g_1(t) = g(t);               %value stored for plotting
17.            g(t) = 0;                    %resolving error
18.         end
19.         plot(g_1,'-<k');
20. end
21.
```

Result:

Command Window

```
Negative Threshold reached at 2 and its value is:22
Negative Threshold reached at 4 and its value is:-22
Negative Threshold reached at 8 and its value is:23
Negative Threshold reached at 16 and its value is:-24
```



Conclusion:

This report addresses with two distinct types of problem set. In first part a fault classification system for five separate vibration signals is deployed which involves spectral analysis of the signal followed by the feature extraction. Feature extraction is done based on dimensionality and correlation. Dimensionality is reduced by deploying Principal component analysis and classification of faults is done using k-Nearest Neighbour algorithm. Accuracy of algorithm is calculated for different values of k. Different classifiers are suggested for fault and a comparison is done between these classifiers based on their advantages and disadvantages. The second part deals with deployment of different types of estimators based on Prior knowledge, of both collected and sequential data. And in the last part CUSUM algorithm is applied on strain gauge sensor data showing results for both positive and negative faults.

References

[1] P. P, “Butterworth-Filters,” tttapa.github.io, [Online]. Available: https://tttapa.github.io/Pages/Mathematics/Systems-and-Control-Theory/Analog-Filters/Butterworth-Filters.html.

[2] Data Analytics, “what-is-principal-component-analysis-pca-and-how-it-is-used,” 18 August 2020. [Online]. Available: https://www.sartorius.com/en/knowledge/science-snippets/what-is-principal-component-analysis-pca-and-how-it-is-used-507186.

[3] ResearchGate, “Learning-vector-quantization,” Research Gate , [Online]. Available: https://www.researchgate.net/figure/Learning-vector-quantization-neural-network_fig3_4225427.

[4] S. Zhu, “Adaptive network based fuzzy inference system,” Research Gate , [Online]. Available: https://www.researchgate.net/figure/Adaptive-network-based-fuzzy-inference-systems-architecture-TP-total-phosphorus-TN_fig2_335594777.

Appendix 1 Spectral Analysis of Signals

```
clear
%=========================Task_1=======================================%

%==================== Plot time-domain graph ==========================%

plot_time_domain('bearing.mat')
plot_time_domain('gearmesh.mat')
plot_time_domain('imbalance.mat')
plot_time_domain('misalignment.mat')
plot_time_domain('multifault.mat')
plot_time_domain('resonance.mat')


%==================== Plot frequency-domain graph and time domain graph side by side ====================%

 global pos_data;
 pos_data=1;
 figure( 'Name','Figure showing a comparison between time domain and frequency domain graph for various types of Faults ')
 plot_frequency_domain('bearing.mat')
 plot_frequency_domain('gearmesh.mat')
 plot_frequency_domain('imbalance.mat')
 plot_frequency_domain('misalignment.mat')
 plot_frequency_domain('multifault.mat')
 plot_frequency_domain('resonance.mat')
```

Code for function to plot in time domain

```

function [] = plot_time_domain(myMat)
y=struct2array(load(myMat));
figure( 'Name',sprintf('Time domain graph of %s',myMat))
plot(y);
xlabel('Time in milliSeconds');
ylabel('Amplitude');
title(sprintf('Time domain graph of %s',myMat));
end
```

Code for function to plot in frequency domain

```
function [] = plot_frequency_domain(myMat)
global pos_data;
y=struct2array(load(myMat));
[P,f]=pwelch(y,[],[],[],1000);
subplot(6,2,pos_data)
plot(f,P)
xlabel('frequency')
ylabel('amplitude')
title(sprintf('frequency domain graph of %s',myMat));
pos_data=pos_data+1;
subplot(6,2,pos_data)
plot(y)
xlabel('time (milli seconds) ')
ylabel('amplitude')
```

```
15. title(sprintf('Time domain graph of %s',myMat));
16. pos_data=pos_data+1;
17. end
18.
```

Appendix 2 Feature Extraction

```
1.  clear
2.  clc
3.  %==================   Bearing    ======================%
4.
5.  x_bearing=segment_data('bearing.mat');
6.  x_normalised_gear=normalize(x_bearing);
7.  f1_bearing=calculate_f1(x_normalised_gear);
8.  f2_bearing=calculate_f2(x_normalised_gear);
9.  f3_bearing=calculate_f3(x_normalised_gear);
10. f4_bearing=calculate_f4(x_normalised_gear);
11. f5_bearing=calculate_f5(x_normalised_gear);
12. f6_bearing=calculate_f6(x_normalised_gear);
13.
14. %================  Gearmesh    ======================%
15.
16. x_gearmesh=segment_data('gearmesh.mat');
17. x_normalised_gearmesh=normalize(x_gearmesh);
18. f1_gearmesh=calculate_f1(x_normalised_gearmesh);
19. f2_gearmesh=calculate_f2(x_normalised_gearmesh);
20. f3_gearmesh=calculate_f3(x_normalised_gearmesh);
21. f4_gearmesh=calculate_f4(x_normalised_gearmesh);
22. f5_gearmesh=calculate_f5(x_normalised_gearmesh);
23. f6_gearmesh=calculate_f6(x_normalised_gearmesh);
24.
25. %================== Imbalance   ======================%
26.
27. x_imbalance=segment_data('imbalance.mat');
28. x_normalised_imbalance=normalize(x_imbalance);
29. f1_imbalance=calculate_f1(x_normalised_imbalance);
30. f2_imbalance=calculate_f2(x_normalised_imbalance);
31. f3_imbalance=calculate_f3(x_normalised_imbalance);
32. f4_imbalance=calculate_f4(x_normalised_imbalance);
33. f5_imbalance=calculate_f5(x_normalised_imbalance);
34. f6_imbalance=calculate_f6(x_normalised_imbalance);
35.
36. %================  misalignment    ======================%
37.
38. x_misalignment=segment_data('misalignment.mat');
39. x_normalised_misalignment=normalize(x_misalignment);
40. f1_misalignment=calculate_f1(x_normalised_misalignment);
41. f2_misalignment=calculate_f2(x_normalised_misalignment);
42. f3_misalignment=calculate_f3(x_normalised_misalignment);
43. f4_misalignment=calculate_f4(x_normalised_misalignment);
44. f5_misalignment=calculate_f5(x_normalised_misalignment);
45. f6_misalignment=calculate_f6(x_normalised_misalignment);
46.
47. %================  resonance   ======================%
48.
49. x_resonance=segment_data('resonance.mat');
50. x_normalised_resonance=normalize(x_resonance);
51. f1_resonance=calculate_f1(x_normalised_resonance);
52. f2_resonance=calculate_f2(x_normalised_resonance);
53. f3_resonance=calculate_f3(x_normalised_resonance);
```

```
f4_resonance=calculate_f4(x_normalised_resonance);
f5_resonance=calculate_f5(x_normalised_resonance);
f6_resonance=calculate_f6(x_normalised_resonance);

%======= Preparation of data for Task_3 =================%

bearing_fault=[f1_bearing;f2_bearing;f3_bearing;f4_bearing;f5_bearing;f6_bearing]';
gearmesh_fault=[f1_gearmesh;f2_gearmesh;f3_gearmesh;f4_gearmesh;f5_gearmesh;f6_gearmesh]';
misalignment_fault=[f1_misalignment;f2_misalignment;f3_misalignment;f4_misalignment;f5_misalignment;f6_misalignment]';
imbalance_fault=[f1_imbalance;f2_imbalance;f3_imbalance;f4_imbalance;f5_imbalance;f6_imbalance]';
resonance_fault=[f1_resonance;f2_resonance;f3_resonance;f4_resonance;f5_resonance;f6_resonance]';
G=[bearing_fault;gearmesh_fault;misalignment_fault;imbalance_fault;resonance_fault];

save 'fault_data_merged' G bearing_fault gearmesh_fault misalignment_fault
imbalance_fault resonance_fault;
```

Function to Segment data

```
function [x] = segment_data(myMat)
x=struct2array(load(myMat));
x=reshape(x,[1000,50]);
end

```

Function to Normalize Data

```
function [x_normalised] = normalize(x)
 x_normalised=x-mean(x);
end

```

Functions to calculate rms value

```
function [rms_value] = rms(matrix)
for i=1:50
    matrix_psd(:,i)=norm(matrix(:,i),2);
    N(:,i)=length(matrix(:,i));
end
rms_value=matrix_psd./sqrt(N);
end

```

Functions to calculate different features

```
function [f1] = calculate_f1(x_norm)
[B,A]=butter(7,0.05);
y1=filter(B,A,x_norm);
[P,f]=pwelch(y1,[],[],[],1000);
f1=rms(P);
end
```

```
function [f2] = calculate_f2(x_norm)
[B,A]=butter(6,[0.05 0.1]);
y1=filter(B,A,x_norm);
[P,f]=pwelch(y1,[],[],[],1000);
f2=rms(P);
end
```

```
function [f3] = calculate_f3(x_norm)
[B,A]=butter(9,[0.1 0.2]);
y1=filter(B,A,x_norm);
[P,f]=pwelch(y1,[],[],[],1000);
f3=rms(P);
end
```

```
function [f4] = calculate_f4(x_norm)
[B,A]=butter(8,[0.2 0.4]);
y1=filter(B,A,x_norm);
[P,f]=pwelch(y1,[],[],[],1000);
f4=rms(P);
end
```

```
function [f5] = calculate_f5(x_norm)
[B,A]=butter(9,[0.4 0.7]);
y1=filter(B,A,x_norm);
[P,f]=pwelch(y1,[],[],[],1000);
f5=rms(P);
end
```

```
function [f6] = calculate_f6(x_norm)
[B,A]=butter(16,0.7,'high');
y1=filter(B,A,x_norm);
[P,f]=pwelch(y1,[],[],[],1000);
f6=rms(P);
end

```

Code to plot different types of faults

•

```
load fault_data_merged.mat;

c=corrcoef(G); % Calculates a correlation coefficient matrix c of G
[v,d] =eig(c); % Find the eigenvectors v and the eigenvalues d of G
T=[ v(:,end)';v(:,end-1)']; % Create the transformation matrix T from
% the first two principal components
z=T*G'; % Create a 2-dimensional feature vector z
a=0:50:250;
color={'or' 'or' 'sg' 'dc' '^k' '<b'};
figure();
for i=2:6
    plot(z(1,a(i-1)+1:a(i)),z(2,a(i-1)+1:a(i)),color{i}) % Scatter plot of the 2-
dimensional features
    hold on
end
title('fault clusters for assignment')

```

Appendix 3: k-Nearest-Neighbourclc;

```
clear;

load fault_data_merged.mat

%======Specify the number of training cases==============%

numberOfTrainingCases=35;
numberOfTestingCases= length(bearing_fault)-numberOfTrainingCases;


trainingSet = [bearing_fault(1:numberOfTrainingCases,:);
               gearmesh_fault(1:numberOfTrainingCases,:);
               imbalance_fault(1:numberOfTrainingCases,:);
               misalignment_fault(1:numberOfTrainingCases,:);
               resonance_fault(1:numberOfTrainingCases,:)];

testingSet = [bearing_fault(numberOfTrainingCases+1:end,:);
               gearmesh_fault(numberOfTrainingCases+1:end,:);
               imbalance_fault(numberOfTrainingCases+1:end,:);
               misalignment_fault(numberOfTrainingCases+1:end,:);
               resonance_fault(numberOfTrainingCases+1:end,:)];

trainingTarget = [ones(1,numberOfTrainingCases)';
    ones(1,numberOfTrainingCases)'*2;
    ones(1,numberOfTrainingCases)'*3;
    ones(1,numberOfTrainingCases)'*4;
    ones(1,numberOfTrainingCases)'*5];

testingTarget = [ones(1,numberOfTestingCases)';
    ones(1,numberOfTestingCases)'*2;
    ones(1,numberOfTestingCases)'*3;
    ones(1,numberOfTestingCases)'*4;
    ones(1,numberOfTestingCases)'*5];

k=11
% Calculate the total number of test and train classes
totalNumberOfTestingCases = numberOfTestingCases * 5;
totalNumberOfTrainingCases = numberOfTrainingCases * 5;
% Create a vector to store assigned labels
inferredLabels = zeros(1, totalNumberOfTestingCases);
% This loop cycles through each unlabelled item:
for unlabelledCaseIdx = 1:totalNumberOfTestingCases
unlabelledCase = testingSet(unlabelledCaseIdx, :);
% As any distance is shorter than infinity
shortestDistance = inf;
shortestDistanceLabel = 0; % Assign a temporary label
% This loop cycles through each labelled item:
for labelledCaseIdx = 1:totalNumberOfTrainingCases
labelledCase = trainingSet(labelledCaseIdx, :);
% Calculate the Euclidean distance:
currentDist(labelledCaseIdx,:) = [euc(unlabelledCase, labelledCase)
    trainingTarget(labelledCaseIdx)];
% distance of all the points are stored in this matrix called current distance
% Check the distance
end % inner loop
sorted_matrix=sortrows(currentDist);    % Current distance matrix is sorted
shortestDistanceLabel=mode(sorted_matrix(1:k,2));    % mode of k elements of sorted
    matrix is taken
% Assign the found label to the vector of inferred labels:
```

```
58. inferredLabels(unlabelledCaseIdx) = shortestDistanceLabel; % label of sorted elements is
    extracted
59. end % outer loop
60.
61.
62. inferredLabels=inferredLabels';
63. Nc=length(find(inferredLabels==testingTarget));
64. Na=length(testingTarget);
65.
66. Accuracy_Percentage=((Nc/Na)*100);
67.
68. disp(Accuracy_Percentage)
69.
```