ACS6124 Assessment (Spring 2022)
Part-I: Multi-sensor and Decision Systems

Saba Firdaus Ansaria
Reg No: 210110201

April 2022

Contents

List of Figures

List of Tables

1 Problem I

Problem 1 generally divulges into vibration signal analysis. In machines, the vibration signals show the dynamic information of the machine and its current state. By analysing these signals, one can monitor different faults of the machines and hence perform risk management. Five different test rigs have been given in the current scenario with the corresponding recorded samples over 50 sec at 1kHz frequency. These are

- Fault1: Bearing Defect - Samples are given in "bearing.mat" (50000x1 column vector).
- Fault 2: Gear mesh - Samples are given in "gearmesh.mat" (50000x1 column vector).
- Fault 3: Resonance - Samples are given in "resonance.mat" (50000x1 column vector).
- Fault 4: Imbalance - Samples are given in "imbalance.mat" (50000x1 column vector).
- Fault 5: Misalignment - Samples are given in "misalignment.mat" (50000x1 column vector).

There are two steps for analysing the signals. Firstly, features are extracted from these samples. Secondly, these features are used to model the fault types using statistical models or estimators.

Also, two scenarios are given for this report. These scenarios are two different sets of feature vectors given below.

Feature Name	Detail of the Feature
f_1	Root Mean Square (RMS) of the signal
f_2	RMS of low frequency: 0–50 Hz
f_3	RMS of middle frequency: 50–200 Hz
f_4	RMS of high frequency: 200–500 Hz

(a) Feature set I.

Feature Name	Filter	Filter Order	W_n
f_1	Low-pass 25 Hz	7	0.05
f_2	Band-pass 25 – 50 Hz	6	[0.05 0.1]
f_3	Band-pass 50 – 100 Hz	9	[0.1 0.2]
f_4	Band-pass 100 – 200 Hz	8	[0.2 0.4]
f_5	Band-pass 200 – 350 Hz	9	[0.4 0.7]
f_6	High-pass 350 Hz	16	0.7

(b) Feature set II.

Figure 1: Feature set I & II.

1.1 Feature Extraction(1a)

Features are statistical information which can be extracted from signals in various ways to describe different characteristics of the signal. Feature extraction can be seen as reducing the dimensionality in complex data while representing the data key properties related to the task. The main steps taken in this section are

1. Data acquisition.

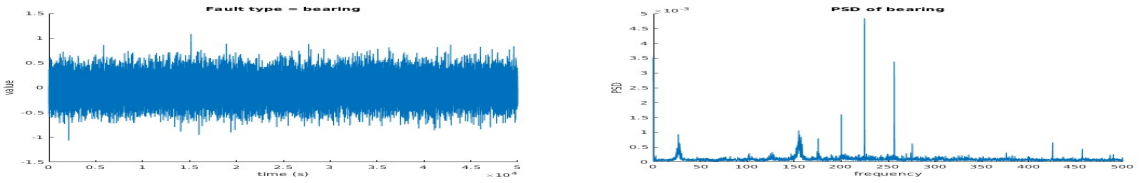
2. Creating filters for feature extraction.
3. Feature extraction.
4. Building feature vectors.
5. Dimensionality reduction and clustering.

1.1.1 Data acquisition

MATLAB is used to load the above mentioned data into variables. These variables are named as fault1, fault2, fault3, fault4, fault5. The data distribution is visualised and compared with their PSD counterparts. The raw signal data is used for feature extraction. Each of the vectors are `50000x1` dimension.

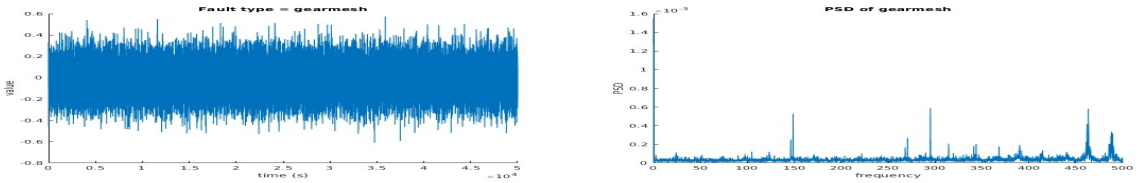
1.1.2 Feature extraction

The `50000x1` dimension vectors are divided into non-overlapping 50 blocks, and each of these 50 blocks contains features of length 1000. The raw data is treated as spatiotemporal data, and spectral analysis has been done on the frequencies.



(a) Bearing. (b) Bearing PSD.

Figure 2: Plot representation of raw signal of bearing vs after doing PSD on the signal



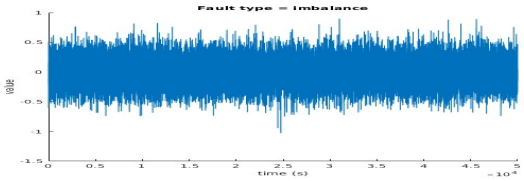
(a) Gearmesh. (b) Gearmesh PSD.

Figure 3: Plot representation of raw signal of Gearmesh vs after doing PSD on the signal

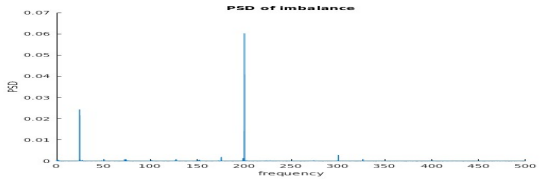
1.1.3 Feature Set I & II Preparation

The Feature Set I and II are calculated generally in the following way.

1. Each `1000x1` vector is normalised.

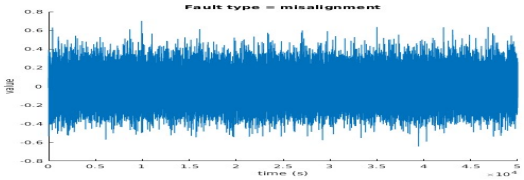


(a) Imbalance.

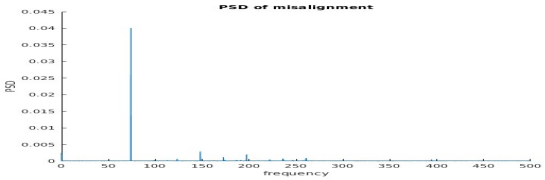


(b) Imbalance PSD.

Figure 4: Plot representation of raw signal of Imbalance vs after doing PSD on the signal

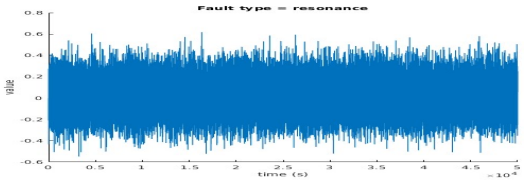


(a) Misalignment.

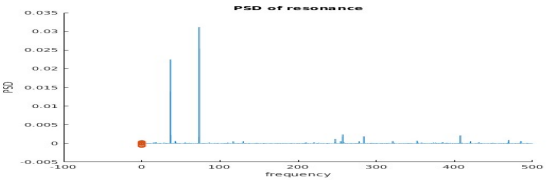


(b) Misalignment PSD.

Figure 5: Plot representation of raw signal of Misalignment vs after doing PSD on the signal



(a) Resonance.



(b) Resonance PSD.

Figure 6: Plot representation of raw signal of Resonance vs after doing PSD on the signal

2. If no filter is used, then the power spectral density of $x_1, x_2, .., x_n$ is calculated using Welch's method.

3. If low-pass, high-pass, band-pass filters are used, the corresponding filters are created, and the signal is passed through the filter. Then the power spectral density (PSD) is calculated using Welch's method.

4. Finally, the RMS is calculated by taking the norm of the PSD.

These steps are elaborated on below.

1.1.4 Normalisation

The vectors are normalised for zero mean by

$$x_{normalised} = x - x_{mean} \tag{1}$$

The vector windows are not overlapping in this case.

1.2 Power Spectra Density

Power spectral density estimates the power in a signal as a frequency component by dividing successive periodogram block and averaging. If a signal $x_1, x_2, ...x_n$ has n number of frames and they are divided into M number of blocks, the PSD according to welch's method will be

$$PSD(x_n) \implies \frac{1}{N} \sum_{m=0}^{N-1} P_{x_m,M}(x_n) \tag{2}$$

The PSD is calculated using MATLAB's pwelch function. For example, f1 calculation in Feature set I is shown below.

```
[P,f]=pwelch(ynorm,[ ],[ ],[ ],1000);
```

The raw signals and their corresponding PSDs are shown in Figures 2-6.

1.3 Filter

If the signal is filtered through a low pass filter or a high pass filter or a band pass filter, the **Butterworth** filter is used for providing a maximally flat response in pass-band and stop-band.

$$|H_n(j\omega)| \triangleq \frac{1}{\sqrt{1+\omega^{2n}}} \tag{3}$$

Where ω is the operating frequency and n is the order of the filter. The *Butterworth* filter is used in MATLAB using butter function. An example code snippet is shown below.

```
[B,A]=butter(16,0.7,"high"); ynormFilter = filter(B,A,ynorm);
```

Here, a high pass filter of 16th order has been created of 350 Hz.

1.4 RMS

The RMS is calculated by the following equation 4, where $||PSD_i||_2$ is the $2-norm$ of the power spectral density. The *norm* function is used in MATLAB for calculating the norm.

$$f_n = \frac{||PSD_i||_2}{\sqrt{N}} \tag{4}$$

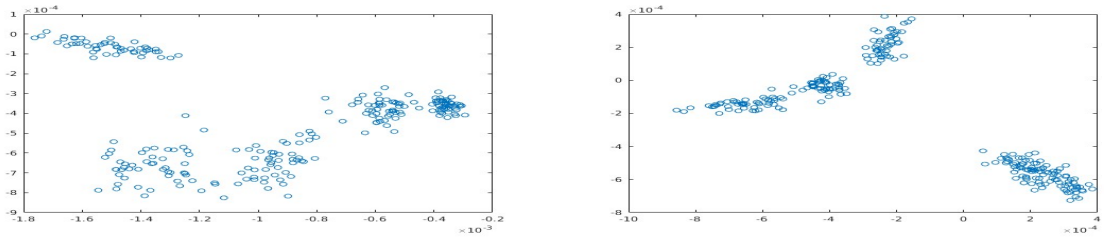
Here is an example of creating a feature vector in which f_5 feature from Feature set II is computed in MATLAB.

```
[B,A]=butter(9,[0.4 0.7],"bandpass");
ynormFilter = filter(B,A,ynorm);
[P,f]=pwelch(ynormFilter,[ ],[ ],[ ],1000);
ynormFilterF5 = norm(P)./sqrt(length(ynormFilter))
```

Here a band-pass filter is created $200-350$Hz using 9th order *butter* filter. Then the PSD is calculated using *pwelch*, and the norm is taken of PSD using *norm* function. Finally, the RMS is calculated by dividing the $2-norm$ with the frequency bin from the filter.

1.4.1 Dimensionality reduction and clustering

The features computed in Section 1.1 are used for unsupervised clustering. However, the feature dimension is four in the first case scenario and 6 in the second case scenario. So, the feature dimensions are reduced to two by using principal component analysis. Firstly, the correlation coefficient matrix is calculated from the combined set of features for each scenario. Next, the eigenvalues and eigenvectors are achieved from the correlation matrix by eigenvalue decomposition. Finally, the first two components are used to create a transformation matrix. The original feature vectors are multiplied with the transformation matrix to get 2-D projections. Generally, the first two principal components are taken (in the lab experiment). However, I have taken first two, first & third, first & fourth principal components in separate clustering scenarios and the scattered clusters are shown in Figure 7, 8, 9.

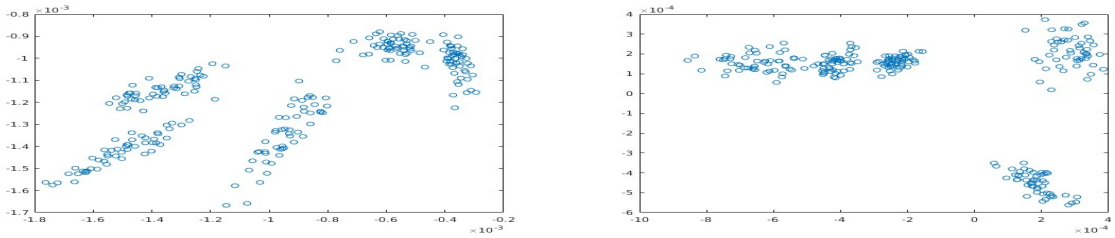


(a) Principal components with feature set I. (b) Principal components with feature set II

Figure 7: Unsupervised PCA cluster (first,second principal component) comparison between feature set I & II.

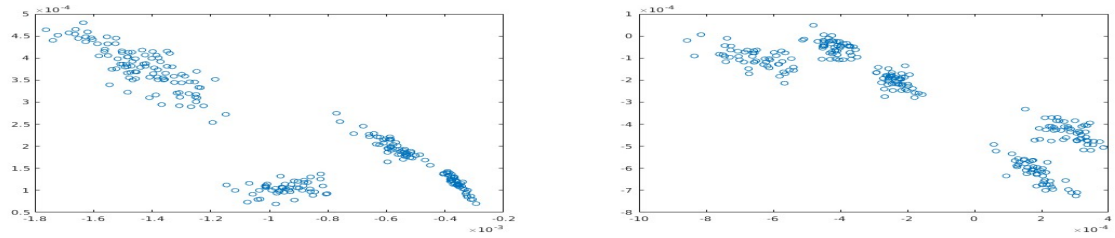
1.4.2 Results and Analysis

Feature extraction is a process of extracting or representing relevant data with low dimensionality. Figures 2-6 clearly demonstrate the difference between raw signal data and extracted feature representation. These features are less noisy and clearly show a trend in the distribution in the power spectral domain. The number of different features signifies different attributes of the signal. This has been explored with both Feature set I and Feature set II.



(a) Principal components with feature set I. (b) Principal components with feature set II

Figure 8: Unsupervised PCA cluster (first,third principal component) comparison between feature set I & II.



(a) Principal components with feature set I. (b) Principal components with feature set II

Figure 9: Unsupervised PCA cluster (first,fourth principal component) comparison between feature set I & II.

The difference between Feature set I and Feature set II is not only the dimensions but also the difference in frequency filters between them. Feature set II uses vast filters and different frequency bands with different orders, and these feature sets are more enriched in terms of information.

As instructed in the LAB tasks, principal components are projected on the feature space, and Figure 7 shows some clusters forming. However, this is not very clear that it has five clusters. Therefore, further experiments have been done. Rather than taking the first two principal components, I have taken the first & third, first & fourth principal components. All of them show different forms of clusters, and with six features, Figure 8 & 9 show clear five clusters, which tells us not only about the features but also the importance of different principal components while reducing dimensionality.

1.5 Pattern Classification (1b)

1.5.1 k-nearest neighbours

K-nearest neighbours algorithm and its variants are non-parametric supervised algorithms which assume that similar things have representations in close proximity. It basically compares the query data with all the training data and declares the nearest training data as the same label as the query data. In a simple setting for k=1, the algorithm tries to find the nearest train example for the query test data. The distance is generally

Nearest Neighbour	Number of Features	Accuracy (%)
k =1	4	98.67
k =2	4	98.67
k =5	4	98.67

Table 1: K nearest neighbour simulation accuracy with feature set I.

euclidean distance. In the Lab, we have been provided with k=1 case of the knn algorithm. However, in this section the algorithm has been modified to accommodate more numbers of k (k=1,2,3,4,5,6, etc.). As a modification, all the distances are measured, and k smallest distanced train object labels are chosen. Finally, a simple mode of the k-label set is taken to get the best possible nearest neighbour. The algorithm is shown in Algorithm 1.

Algorithm 1 k-nearest neighbours algorithm generalised for k.

procedure K-NN($k, trainset, trainlabels, testset, testlabels$)
  Initialise system parameters ▷ Read all the input values and initialise the params.
  Read the input values
  $N \leftarrow length(testset)$
  $M \leftarrow length(trainset)$
  $inferredLabels$ is initialised as an empty vector same length as testset
  for i=1,2,..N **do** ▷ Run loop for all the test cases
    $unlabelledSample \leftarrow testset[i]$
    $shortestDistance \leftarrow inf$
    $shortestDistanceLabel \leftarrow 0$
    for j=1,2,..M **do** ▷ Run loop for all the train cases
      $labelledSample \leftarrow trainset[j]$)
      $currentDistance = euclidDist(unlabelledSample, labelledSample)$
      Append $currentDistance$ to $DISTANCE$
    sort $DISTANCE$ for arranging the values
    SELECT lowest k values in $DISTANCE$
    SELECT corresponding $trainlabels$ for those lowest distances
    $KNEAREST \leftarrow SELECT[k][trainlabels]$
    $label \leftarrow mode(KNEAREST)$
    $inferredLabels[i] \leftarrow label$

1.5.2 Results

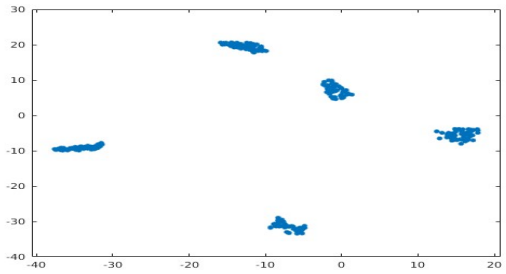
The feature vectors of length 4 (Feature set I) have been used from Section 1.1. These data have been split into train and test set. For each fault case, 35 samples have been used for training, and 15 samples have been used for testing. Therefore, a total of 175 samples have been used for training, and 75 samples have been used for test/query set. As prescribed, the algorithm is trained for k=1,2, 5, and the results are shown in Table 1. All these experiments retain the same accuracy of 98.67%. Some improvements are proposed in the next section.

k value	Feature length	Accuracy(%)
k =1	4	98.67
k =2	4	98.67
k =5	4	98.67
k = 3	4	100.00
k = 4	4	100.00
k = 6	4	97.34
k = 7	4	100.00

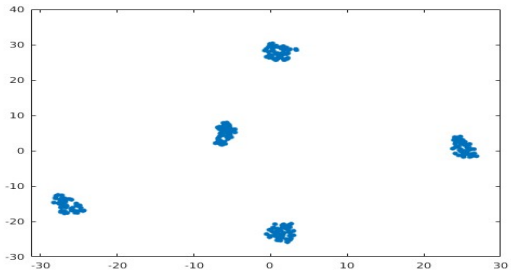
Table 2: K nearest neighbour simulation accuracy with feature set I.

k value	Feature length	Accuracy(%)
k =1	6	100.00
k =2	6	100.00
k =3	6	100.00
k = 4	6	100.00
k = 5	6	100.00
k = 6	6	100.00
k = 7	6	100.00

Table 3: K nearest neighbour simulation accuracy with feature set II.



(a) t-SNE clusters with feature set I.



(b) t-SNE clusters with feature set II

Figure 10: Unsupervised t-SNE comparison between feature set I & II.

1.6 Possible Improvements

The first significant improvement that can be made to the k-nn problem is to choose the number of neighbours carefully. As k-nn is hugely dependant on the euclidean distance, the outliers play a considerable role in confusing the algorithm. One way to reduce the outlier effect is to choose a different k and compare the results. In this case, further values of k have been explored, and the results are given in Table 2. k=3,4,7 achieved 100% accuracy on the test set. However, k=7 resulted in reduced accuracy as 97.34%. Therefore choosing the right k is a great way to improve.

As discussed in Section 1.4.2, the feature dimensionality plays a huge role in representing the data. The 6-length Feature set II has also been used with the k-nn, and it achieves 100% accuracy in k=1,2,...,7. The result is shown in Table 3. However, if the dimensionality of the feature is very high, it confuses the k-nn classifier because, in higher dimensions, it is possible to get many different vectors equidistant to the query/test vector.

Data normalisation is very important for improving k-nn. Because the decision making or voting of the neighbours are based on euclidean distance, unnormalised data can become very confusing because they will be in different spaces.

Finally, it would also be pragmatic to use other supervised classifiers like support vector machines or unsupervised algorithms like t-SNE. They mostly reply on proximity-based decision boundaries, which is also very helpful for this type of data. The feature

vectors are trained with the t-SNE algorithm (using MATLAB tsne function) and the clusters are shown in Figure 10.

2 Problem II

The task is to design a multisensor signal estimation and health monitoring system in a wind turbine. The pitch angle $\hat{\omega}$ is measured with a rotary encoder connected to the blade bearing. The sensor noise is distributed with zero mean and variance 9 ($v \sim N(0, 9)$).

2.1 Experiment Scenarios (1/2)

The measurement vector is given in *encoder.mat*. There are two scenarios for the experiments. In *scenario 1*, all the samples should be used in the experiments. In *scenario 2*, only the first five samples should be used in the experiments.

2.2 MMSE estimator with prior angle (II-a)

MMSE (minimum mean square error) is an estimation method that models values of a dependant variable with a cost function by minimising the mean square error. In this task, the mean and variance are finite. Therefore the MMSE estimator, in this case, is uniquely defined.

Aim: The prior knowledge of the angle is uniformly distributed in the range of $0° \leq \hat{\omega} \leq 30°$. The goal is to calculate $\hat{\omega}$.

Formulation: Let us assume the given input data *encoder.mat* contains the measurements Y in time N, so that $y_1, y_2, ..., y_n$ where $n \in N$. The noise component is given as $v_1, v_2, ..., v_n$ where $n \in N$. The goal is to estimate the ω components such as $\omega_1, \omega_2, ..., \omega_n$. The measurement model is given as

$$y_n = \omega_n + v_n, n \in N \tag{5}$$

Here we assume the noise model in independent with zero mean and σ^2 variance. The estimator with MMSE cost function will be

$$MSE(\omega) = \mathbb{E}[(y_n - \omega)^2] \tag{6}$$

The optimal estimate $\hat{\omega}$ would be

$$\hat{\omega} = argminMSE(\omega) \tag{7}$$

When the prior knowledge about ω_n becomes available as $\omega \sim U[a, b]$, the formulation of $\hat{\omega}$ is shown in Equation 8.

Number of samples	estimator (MMSE)
5 (scenario1)	17.3125
100 (scenario2)	20.3131

Table 4: MMSE estimator with scenario 1 and scenario 2.

$$
\begin{aligned}
\hat{\omega} &= \exp[\omega | Y_{1:N}] \\
\hat{\omega} &= \int_{-\infty}^{\infty} \omega p(\omega | Y_{1:N}) \, d\omega \\
\hat{\omega} &= \frac{\int_b^a \frac{\omega}{\sqrt{2\Pi \frac{\sigma^2}{N}}} \exp\left[-\frac{N}{2\sigma^2} \left(\omega - \frac{1}{N} \sum_{n=1}^{N} y_n \right)^2 \right] d\omega}{\int_b^a \frac{1}{\sqrt{2\Pi \frac{\sigma^2}{N}}} \exp\left[-\frac{N}{2\sigma^2} \left(\omega - \frac{1}{N} \sum_{n=1}^{N} y_n \right)^2 \right] d\omega}
\end{aligned}
\tag{8}
$$

The equation is solved using MATLAB. The implementation of the numerator and denominator is shown below and the integration is solved using *integral* function built in MATLAB.

```
% MMSE estimator (question II a)
% number of observations
n = length(samples);
% standard deviation
var_sigma = 9;
% mean over the total number of observations
mean_samples = sum(samples)/n;
% MMSE estimator integral objects for equation 8
numerator_mmse_fn = @(w) (w/sqrt(2.*pi.*(var_sigma/n))).*exp(-(1/2.*n/var_sigma).*((w-mean_samples).^2));
denominator_mmse_fn = @(w) (1/sqrt(2.*pi.*(var_sigma/n))).*exp(-(1/2.*n/var_sigma).*((w-mean_samples).^2));
%integration of numerator and denominator for equation 4
numerator_mmse = integral(numerator_mmse_fn, 0, 30);
denominator_mmse = integral(denominator_mmse_fn, 0, 30);
w_star = numerator_mmse/denominator_mmse;
disp(w_star);
```

2.2.1 Results and Discussion

The MMSE estimator is calculated using two scenarios with 5 samples and 100 data samples respectively. The results are shown in Table 4. The MMSE estimator hinges less on the prior distribution when the number of samples increases and relies more on the data. The MMSE is 17.31 with five samples and 20.21 with a hundred samples.

2.3 MMSE estimator with prior Gaussian distribution (II-b)

Aim: The goal is to extend the problem done in Section 2.2 with the improved prior knowledge which says the signal is Gaussian distributed with mean value 15 and variance 4.

Number of samples	estimator (MMSE)	Data Mean
5 (scenario1)	16.5948	17.3125
100 (scenario2)	20.1952	20.3131

Table 5: MMSE estimator with gaussian priors (scenario 1 & 2).

Formulation: With the previous variables and formulation at hand, i.e, $\mu_\omega = 15$ and $\sigma_\omega^2 = 4$. The MMSE estimator is given by $\mathbb{E}[\omega|Y_{1:N}]$ as Equation 9.

$$\begin{aligned}\mathbb{E}[\omega|Y_{1:N}] &= \frac{\frac{1}{\sigma^2}(\sum_{n=1}^{N} y_n) + \frac{\mu_\omega}{\sigma^2}}{\frac{N}{\sigma^2} + \frac{1}{{\sigma_\omega}^2}} \\ \mathbb{E}[\omega|Y_{1:N}] &= \frac{{\sigma_\omega}^2}{{\sigma_\omega}^2 + \frac{\sigma^2}{N}}\left(\frac{1}{N}\sum_{n=1}^{N} y_n\right) + \frac{\frac{\sigma^2}{N}}{{\sigma_\omega}^2 + \frac{\sigma^2}{N}}\mu_\omega\end{aligned} \tag{9}$$

Where μ_ω is the prior mean, σ_ω^2 is prior variance, N is the number of samples in temporal domain, σ^2 is the noise variance and y_n is the temporal sensory reading data.

The estimator is implemented in MATLAB and the code snippet is given below.

```
%declare variables
mean_prior = 15 ;
var_sigma_prior = 4;
% MMSE estimator with gaussian prior Eqation 9
data_observed = ((var_sigma_prior./(var_sigma_prior+(var_sigma/n))).*mean_samples);
prior = ((var_sigma/n)./(var_sigma_prior+(var_sigma/n))).*mean_prior;
w_star_p = data_observed+prior;
disp(w_star_p)
```

2.3.1 Results and Discussion

The estimator at Equation 9 is implemented using MATLAB, and the code snippet is given below. The MMSE estimator with prior Gaussian data gives an estimate of the signal depending on the amount of the training samples. It can be seen that with 5 samples, it gives 16.59, which is the prior mean and with 100 samples, MMSE is 20.1952, which is the actual mean of the data. Here the prior mean is 15.

Therefore, when large data is available, the Gaussian MMSE estimates closer to the mean of the actual data. When a small number of observations are available, the Gaussian MMSE estimates closer to the prior mean.

2.4 MMSE estimator with sequential data (II-c)

Aim: The aim of the task is to extend the tasks in Section 2.2 and 2.3 and treat the data as a sequential data in time. The goal is to get the update of the estimator ω_{n+1} given that ω_nth sample has already arrived.

Number of samples	Sequential estimator(t)	Sequential estimator(t+1)
5 (scenario1)	19.8044	19.8321, 19.8661
100 (scenario2)	10.6180	11.1525, 11.9397

Table 6: Sequential MMSE estimator with Gaussian prior results.

Formulation: Considering the observation model in Equation **??**, with the Gaussian priors for estimate signal and noise, $\hat{\omega_N}$ will be Equation 10.

$$\begin{aligned} \hat{\omega_N} &= \frac{\frac{1}{\sigma^2}\sum_{n=1}^{N} y_n}{\frac{N}{\sigma^2}+\frac{1}{\sigma_\omega^2}} \\ \hat{\omega_N} &= \frac{{\sigma_\omega}^2}{N{\sigma_\omega}^2+\sigma^2}\sum_{n=1}^{N} y_n \end{aligned} \tag{10}$$

Where μ_ω is the prior mean, σ_ω^2 is the prior variance, N is the number of samples in the temporal domain, σ^2 is the noise variance, and y_n is the temporal sensory reading data.

When a new signal reading comes at time $n+1$, the estimator will be updated sequentially considering the previous $\hat{\omega_N}$ estimator. The updated is shown at Equation 11.

$$\begin{aligned} \hat{\omega_{N+1}} &= \frac{{\sigma_\omega}^2}{(N+1){\sigma_\omega}^2+\sigma^2}\sum_{n=1}^{N+1} y_n \\ \hat{\omega_{N+1}} &= \hat{\omega_N} + \frac{{\sigma_\omega}^2}{(N+1){\sigma_\omega}^2+\sigma^2}(y_{N+1}-\hat{\omega_N}) \end{aligned} \tag{11}$$

The implementation is performed using MATLAB and the code snippet is given below.

```
total_samples = length(samples);
for i = 1:(total_samples-1) % iterate through the samples
    t_sample = samples(1:i);
    sample_sum = sum(t_sample);
    % equation 10 and 11
    w_t_star = ((var_sigma_prior)./(i.*var_sigma_prior+var_sigma)).*sample_sum;
    w_t_1_star = w_t_star +
    (var_sigma_prior./((i+1).*var_sigma_prior+var_sigma)).*(samples(i:i+1,:)-w_t_star);
end
```

2.4.1 Results and Discussion

The experiments are conducted with both scenarios 1 and 2 (100 length measurement and five length measurement, respectively), and the results are shown in Table 6. The

$\hat{\omega_{N+1}}$ and $\hat{\omega_N}$ are 19.86 and 19.80 for five samples. The $\hat{\omega_{N+1}}$ and $\hat{\omega_N}$ are 11.93 and 10.61 for hundred samples. It can be inferred that the step size and number of the sample are essential for sequential MMSE.

2.5 Two-sided CUSUM test (II-d)

CUSUM (Cumulative Sum) algorithm is used for change detection in a sequential signal. If a temporal signal value makes an abrupt change at a given time n_{change}, it is necessary to detect it for system health monitoring purposes. The change can occur in different directions.

Aim: For this task, sensor estimates over k samples have been given from a strain gauge sensor. The prior knowledge is that the system's normal operation is 3000KNm with a variance of 1. The change threshold is ± 20. The goal is to develop a two-sided CUMSUM algorithm.

Traditionally, CUSUM was proposed as a one-sided test, i.e., a positive CUSUM test or negative CUSUM test, to observe if the change has a positive or negative trend. However, it would be efficient to evaluate change detection in either direction simultaneously. For this purpose the instantaneous log-likelihood ratios for the change becomes modified as Equation 12 and 14 [1].

$$signal_i[n] = +\frac{|\delta|}{{\sigma_\omega}^2}\left(\omega[n] - \mu_{\omega o} - \frac{|\delta|}{2}\right) \tag{12}$$

$$G_i[n] = G_i[n-1] + signal_i[n] \tag{13}$$

$$signal_d[n] = -\frac{|\delta|}{{\sigma_\omega}^2}\left(\omega[n] - \mu_{\omega o} + \frac{|\delta|}{2}\right) \tag{14}$$

$$G_d[n] = G_d[n-1] + signal_d[n] \tag{15}$$

Where δ is taken as the absolute value in the change magnitude, $\mu_{\omega o}$ is the prior mean about the signal data and $w[n]$ is the value at time n and σ_ω^2 is the prior signal variance. It checks if the change occur between two simultaneous readings exceeds the threshold change magnitude. For this scenario $\mu_{\omega o} = 3000$, $\sigma_\omega^2 = 1$ and $\delta = \pm 20$. The pseudo code of the algorithm is given below.

Algorithm 2 Two-sided CUSUM algorithm.

procedure 2-SIDE-CUSUM($sample, prior, \delta$)
  Initialise system parameters ▷ Read all the input values and initialise the params.
  Read the input values
  $G_i(0) \leftarrow 0, G_d(0) \leftarrow 0$
  $n \leftarrow 1, N \leftarrow k$
  while $n \leq N$ **do** ▷ Run loop for all the sample readings
    $s_i[n] \leftarrow +\frac{|\delta|}{{\sigma_\omega}^2}\left(\omega[n] - \mu_{\omega o} - \frac{|\delta|}{2}\right)$ ▷ Equation 5
    $s_d[n] \leftarrow -\frac{|\delta|}{{\sigma_\omega}^2}\left(\omega[n] - \mu_{\omega o} + \frac{|\delta|}{2}\right)$ ▷ Equation 6
    $G_i[n] \leftarrow G_i[n-1] + s_i[n]$ ▷ Update G
    $G_d[n] \leftarrow G_d[n-1] + s_d[n]$
    if $G_i[n] < 0$ **then** ▷ Prevent from going to negative continuously
      $G_i[n] \leftarrow 0$
    if $G_d[n] < 0$ **then**
      $G_d[n] \leftarrow 0$
    if $G_i[n] > \delta$ **then** ▷ If G value exceeds the threshold
      System alarm notification
    if $G_d[n] > \delta$ **then**
      System alarm notification

The code snippet of the algorithm is give below.

```
for i = 1:(total_samples)
    s_i = (abs(delta_threshold)./var_sigma_prior).*(samples(i)-mean_prior-(abs(delta_threshold)/2));
    s_d = -(abs(delta_threshold)./var_sigma_prior).*(samples(i)-mean_prior+(abs(delta_threshold)/2));
    g_t_i = g_t_i + s_i;
    g_t_d = g_t_d + s_d;
    if g_t_i < 0
        g_t_i = 0;
    end
    if g_t_d < 0
        g_t_d = 0;
    end
    if g_t_i > delta_threshold
        fprintf("Set alarm");
    end
    if g_t_d > delta_threshold
        fprintf("Set alarm");
    end
end
```

2.5.1 Results

```
Set alarm (signal change more than threshold), current signal value is 3015.
Set alarm (signal change more than threshold), current signal value is 3007.
Set alarm (signal change more than threshold), current signal value is 2988.
Set alarm (signal change more than threshold), current signal value is 3013.
Set alarm (signal change more than threshold), current signal value is 3014.
Set alarm (signal change more than threshold), current signal value is 2986.
Set alarm (signal change more than threshold), current signal value is 3019.
```

References

[1] Pierre Granjon. The cusum algorithm-a small review. 2013.

3 Appendix

```
% frequency_analysis.m
% load and plot data
% (bearing, gearmesh, misalignment, imbalance,resonance).mat

%loading
load("bearing.mat","bearing")
load("gearmesh.mat","gearmesh")
load("misalignment.mat","misalignment")
load("imbalance.mat","imbalance")
load("resonance.mat","resonance")

%options
simple_plot=false; %true
number_of_features = 6;
save_features = false;
visualise_pca = true;
visualise_tsne = true;

%plotting the given fault data
if simple_plot
    counter = 1;
    myPlotting(bearing,"bearing",counter);
    counter=counter+2;
    myPlotting(gearmesh,"gearmesh",counter);
    counter = counter+2;
    myPlotting(misalignment,"misalignment",counter);
    counter = counter+2;
    myPlotting(imbalance,"imbalance",counter);
    counter = counter+2;
    myPlotting(resonance,"resonance",counter);
end


if number_of_features == 4
    %feature extraction from the given fault data
    feature_vector_bearing = feature_extraction_4(bearing);
    feature_vector_gearmesh = feature_extraction_4(gearmesh);
    feature_vector_misalignment = feature_extraction_4(misalignment);
    feature_vector_imbalance = feature_extraction_4(imbalance);
```

```
    feature_vector_resonance = feature_extraction_4(resonance);

    combined = vertcat(feature_vector_bearing, feature_vector_gearmesh,
    feature_vector_misalignment, ...
    feature_vector_imbalance, feature_vector_resonance);

elseif number_of_features == 6
    %feature extraction from the given fault data
    feature_vector_bearing = feature_extraction_6(bearing);
    feature_vector_gearmesh = feature_extraction_6(gearmesh);
    feature_vector_misalignment = feature_extraction_6(misalignment);
    feature_vector_imbalance = feature_extraction_6(imbalance);
    feature_vector_resonance = feature_extraction_6(resonance);

    combined = vertcat(feature_vector_bearing,
    feature_vector_gearmesh, feature_vector_misalignment, ...
        feature_vector_imbalance, feature_vector_resonance);
end

% visualize principal components (initially) of the extracted features
if visualise_pca
     principalComponent(combined);
end

% visualize tSNE (initially) of the extracted features
if visualise_tsne
     y = tsne(combined);
     gscatter(y(:,1),y(:,2));
end

% save feature vectors

if save_features
    save("fault1.mat","feature_vector_bearing");
    save("fault2.mat","feature_vector_gearmesh");
    save("fault3.mat","feature_vector_misalignment");
    save("fault4.mat","feature_vector_imbalance");
    save("fault5.mat","feature_vector_resonance");
end

############################################################
```

```
% function definition
% myPlotting.m

function myPlotting(sample,name,counter)
    figure(counter);hold on
    plot(sample);
    xlabel('time (s)')
    ylabel('value')
    title_text = "Fault type = "+ name;
    title(title_text);
    [P,f]=pwelch(sample,[],[],[],1000);%1000 is the sampling frequency (Hz)
    figure(counter+1);hold on
    plot(f,P);
    xlabel('frequency')
    ylabel('PSD')
    name = "PSD of " + name;
    title(name);
end


################################################

% feature extraction
% feature_extraction_4.m
function feature_vector=feature_extraction_4(sample)
    for i=1:50
        y = sample(((i-1)*1000)+1:i*1000);
        ynorm = normalize(y);
        % feature f1
        % PSD with pwelch
        [P,f]=pwelch(ynorm,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_f1 = norm(P)./sqrt(length(ynorm)) % RMS
        % feature f2
        [B,A]=butter(11,0.1); % butterworth filter
        ynorm_filter = filter(B,A,ynorm); % applying butterworth
        [P,f]=pwelch(ynorm_filter,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_filter_f2 = norm(P)./sqrt(length(ynorm_filter))
        %feature f3
        [B,A]=butter(13,[0.1 0.4]);
        ynorm_filter = filter(B,A,ynorm);
```

```
        [P,f]=pwelch(ynorm_filter,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_filter_f3 = norm(P)./sqrt(length(ynorm_filter))
        %feature f4
        [B,A]=butter(18,0.4,'high');
        ynorm_filter = filter(B,A,ynorm);
        [P,f]=pwelch(ynorm_filter,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_filter_f4 = norm(P)./sqrt(length(ynorm_filter))
        %create feature matrix
        feature_vector(i,:) = [ynorm_f1 ynorm_filter_f2 ynorm_filter_f3 ynorm_filter_f4]
    end
end


################################################

% feature_extraction_6.m
function feature_vector=feature_extraction_6(sample)
    for i=1:50
        y = sample(((i-1)*1000)+1:i*1000);
        ynorm = normalize(y);
        % feature f1
        [B,A] = butter(7,0.05,"low"); % butterworth filter
        ynorm_filter = filter(B,A,ynorm);
        [P,f]=pwelch(ynorm_filter,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_filter_f1 = norm(P)./sqrt(length(ynorm)) %RMS
        % feature f2
        [B,A]=butter(6,[0.05 0.1],"bandpass");
        ynorm_filter = filter(B,A,ynorm);
        [P,f]=pwelch(ynorm_filter,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_filter_f2 = norm(P)./sqrt(length(ynorm_filter))
        %feature f3
        [B,A]=butter(9,[0.1 0.2],"bandpass");
        ynorm_filter = filter(B,A,ynorm);
        [P,f]=pwelch(ynorm_filter,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_filter_f3 = norm(P)./sqrt(length(ynorm_filter))
        %feature f4
        [B,A]=butter(8,[0.2 0.4],"bandpass");
        ynorm_filter = filter(B,A,ynorm);
        [P,f]=pwelch(ynorm_filter,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_filter_f4 = norm(P)./sqrt(length(ynorm_filter))
        %feature f5
```

```
        [B,A]=butter(9,[0.4 0.7],"bandpass");
        ynorm_filter = filter(B,A,ynorm);
        [P,f]=pwelch(ynorm_filter,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_filter_f5 = norm(P)./sqrt(length(ynorm_filter))
        %feature f6
        [B,A]=butter(16,0.7,"high");
        ynorm_filter = filter(B,A,ynorm);
        [P,f]=pwelch(ynorm_filter,[],[],[],1000);%1000 is the sampling frequency (Hz)
        ynorm_filter_f6 = norm(P)./sqrt(length(ynorm_filter))
        % creating feature matrix
        feature_vector(i,:) = [ynorm_filter_f1 ynorm_filter_f2
            ynorm_filter_f3 ynorm_filter_f4 ynorm_filter_f5 ynorm_filter_f6]
    end
end



#########################################################

function principalComponent(G)
    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-3)']; % Create the transformation matrix T from
    % the first two principal components
    z=T*G'; % Create a 2-dimensional feature vector z
    plot(z(1,:),z(2,:),'o') % Scatter plot of the 2-dimensional features

end

#########################################################

function distance = euc(a, b)
%Euclidean Distance
% Calculates the Euclidean distance between two cases which an equal
% number of features.
    if nargin  ~= 2
        error('Two input arguments required.');
        return;
    end

    if  ~all(size(a) == size(b))
```

```
        error('Dimensions of inputs are not equal.');
        return;
    end

    if min(size(a)) ~= 1
        error('Input is not a vector');
        return;
    end
    % Calculate the Euclidean Distance using the MATLAB's norm function
    distance = norm(a - b);

end

########################################


% knn_classification.m
% load feature files

load('fault1.mat','feature_vector_bearing');
Fault1 = feature_vector_bearing;
load("fault2.mat","feature_vector_gearmesh");
Fault2 = feature_vector_gearmesh;
load("fault3.mat","feature_vector_misalignment");
Fault3 = feature_vector_misalignment;
load("fault4.mat","feature_vector_imbalance");
Fault4 = feature_vector_imbalance;
load("fault5.mat","feature_vector_resonance");
Fault5 = feature_vector_resonance;

% split train/test
% Specify the number of training cases:
numberOfTrainingCases = 35;
trainingSet = [Fault1(1:numberOfTrainingCases,:);Fault2(1:numberOfTrainingCases,:);Fault3(1:
    Fault4(1:numberOfTrainingCases,:);Fault5(1:numberOfTrainingCases,:)];
testingSet = [Fault1(numberOfTrainingCases+1:end,:); Fault2(numberOfTrainingCases+1:end,:);
    Fault3(numberOfTrainingCases+1:end,:);Fault4(numberOfTrainingCases+1:end,:);Fault5(numbe

% Note the below works because all faults are of equal lengths.
numberOfTestingCases = length(Fault1) - numberOfTrainingCases;
trainingTarget = [ones(1,numberOfTrainingCases), ones(1,numberOfTrainingCases)*2,ones(1,numb
```

```
    ones(1,numberOfTrainingCases)*4,ones(1,numberOfTrainingCases)*5];
testingTarget = [ones(1,numberOfTestingCases),ones(1,numberOfTestingCases)*2,ones(1,numberOf
    ones(1,numberOfTestingCases)*4,ones(1,numberOfTestingCases)*5];

%define k value for k nearest neighbour
k = 1;

% 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
    unlabelledSample = 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
        labelledSample = trainingSet(labelledCaseIdx, :);
        % Calculate the Euclidean distance:
        currentDist = euc(unlabelledSample, labelledSample);
        % assign distance to a distance matrix
        distance(labelledCaseIdx, :) = currentDist;
    end % inner loop
    [sorted_distance, sortindex] = sort( distance );
    sorted_distance = sorted_distance(1:k);
    sortindex = sortindex(1:k);

    % take k labels from k nearest samples
    for index = 1:k
        k_labels = trainingTarget(sortindex(index));
    end

    %take mode of k neighbours
    [val, freq] = mode(k_labels(:));

    % Assign the found label to the vector of inferred labels:
    inferredLabels(unlabelledCaseIdx) = val;
end % outer loop
```

```
% accuracy calculation
Nc = length(find(testingTarget == inferredLabels));
acc = 100 * (Nc/totalNumberOfTestingCases);
fprintf("The accuracy is: %f \n", acc);
% disp(acc);


##################################################################

% mmse.m

clear all;

% loading the encoder.mat readings

load("encoder.mat","encoder");

% choose whether use whole encoder.mat data or the first five samples (1/2)

data_selection = 1;

% total observations

if data_selection == 1
    samples = encoder;
elseif data_selection == 2
    samples = encoder(1:5,:);
end

% MMSE estimator (question II a)

% number of observations
n = length(samples);

% standard deviation
var_sigma = 9;
% mean over the total number of observations
mean_samples = sum(samples)/n;
% MMSE estimator
numerator_mmse_fn = @(w)
```

```
    (w/sqrt(2.*pi.*(var_sigma/n))).*exp(-(1/2.*n/var_sigma).*((w-mean_samples).^2));
denominator_mmse_fn = @(w)
    (1/sqrt(2.*pi.*(var_sigma/n))).*exp(-(1/2.*n/var_sigma).*((w-mean_samples).^2));

numerator_mmse = integral(numerator_mmse_fn, 0, 30);
denominator_mmse = integral(denominator_mmse_fn, 0, 30);

w_star = numerator_mmse/denominator_mmse;
disp("mean:");
disp(mean_samples);
disp("w_star_IIa");
disp(w_star);

% prior mean and variance (question II b)
mean_prior = 15 ;
var_sigma_prior = 4;

data_observed = ((var_sigma_prior./(var_sigma_prior+(var_sigma/n))).*mean_samples);
prior = ((var_sigma/n)./(var_sigma_prior+(var_sigma/n))).*mean_prior;
w_star_p = data_observed+prior;
disp("w_star_p");
disp(w_star_p)


% sequential update (question II c)

total_samples = length(samples);
for i = 1:(total_samples-1)
    t_sample = samples(1:i);
    sample_sum = sum(t_sample);
    w_t_star = ((var_sigma_prior)./(i.*var_sigma_prior+var_sigma)).*sample_sum;
    w_t_1_star = w_t_star +
    ((var_sigma_prior./((i+1).*var_sigma_prior+var_sigma)).*(samples(i:i+1,:)-w_t_star));
end
disp("w_t_star");
disp(w_t_star);
disp("w_t_1_star");
disp(w_t_1_star);


##################################################################
```

```
% cusum.m
% loading data
load("straingauge.mat", "straingauge");
samples = straingauge;

% prior
mean_prior = 3000;
var_sigma_prior = 1;

%threshold

delta_threshold = 20;

total_samples = length(samples);
g_t_i = 0;
g_t_d = 0;
for i = 1:(total_samples)
    % log-likelihoods
    s_i = (abs(delta_threshold)./var_sigma_prior)
        .*(samples(i)-mean_prior-(abs(delta_threshold)/2));
    s_d = -(abs(delta_threshold)./var_sigma_prior)
        .*(samples(i)-mean_prior+(abs(delta_threshold)/2));
    g_t_i = g_t_i + s_i;
    g_t_d = g_t_d + s_d;
    if g_t_i < 0
        g_t_i = 0;
    end

    if g_t_d < 0
        g_t_d = 0;
    end

    %decision

    if g_t_i > delta_threshold

        fprintf("Set alarm (signal change more than threshold),
        current signal value is %d\n",samples(i));
    end
```

```
    if g_t_d > delta_threshold
        fprintf("Set alarm (signal change more than threshold),
        current signal value is %d\n",samples(i));
    end
end
```