Predictive Analytics for Structural Health Monitoring

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

B.Tech

in

Mechanical Engineering

By

Kailash Jagadeesh (111118045)

Nandhini S R (111118073)

R Surya Narayan (111118091)

MECHANICAL ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY
TIRUCHIRAPPALLI – 620015

MAY 2022

dedicated to all Machine Learning Enthusiasts who wish to explore and solve world's innovative problems...

BONAFIDE CERTIFICATE

This is to certify that the project titled **Predictive Analytics for Structural Health Monitoring** is a bonafide record of the work done by

Kailash Jagadeesh (111118045)

Nandhini S R (111118073)

R Surya Narayan (111118091)

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Mechanical Engineering** of the **NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI**, during the year 2021-22.

Dr. Raghu Ram Desu	**Dr. AR. Veerappan**
Guide	Head of the Department

Project Viva-voce held on ________________________

Internal Examiner	**External Examiner**

ABSTRACT

Machine elements like shafts and bearings in large machinery, such as gas turbines, are prone to develop faults due to continuous operation and harsh working conditions. As a result, continuous monitoring of their health and prediction of imminent defects becomes a necessity to avoid the catastrophic failure of these components. Intelligent analytics on real-time vibration-based sensor data prove instrumental in fault detection and diagnosis. Statistical algorithms that are able to decipher patterns in the signal and identify fault-related features are hence required. With the expanding scope of the use of Machine Learning techniques in solving such problems, this work aims to explore suitable algorithms that identify the presence of defects and faults in vibration-based signals collected from sensors. Four dimensionality reduction techniques for the identification of a single fault in a cracked rotor were explored and 2 clustering algorithms for multiple faults in bearings, the datasets of which were picked up from suitable experimental facilities were also investigated. The algorithms show very close mutual agreement in predicting the onset of transverse fatigue-induced cracks in the rotor data and also show a close correlation to the experimentally observed onset of crack. Clustering algorithms used for identifying multiple faults on the bearing dataset also show good demarcation of data between the faults and correlation with experimentally observed results.

Keywords:
Structural Health Monitoring, Machine Learning, Fault Detection and Diagnosis

ACKNOWLEDGEMENTS

We would like to express our deepest gratitude to the following people for guiding us through this course and without whom this project and the results achieved from it would not have reached completion.

Dr. Raghu Ram Desu, Assistant Professor, Department of Mechanical Engineering, for helping us and guiding us in the course of this project. Without his guidance, we would not have been able to successfully complete this project. His patience and genial attitude is and always will be a source of inspiration to us.

Dr. AR. Veerappan, the Head of the Department, Department of Mechanical Engineering, for allowing us to avail the facilities at the department.

In addition, we also express our sincerest thanks to **Dr. Rathna Prasad Sagi** for the datasets and timely guidance throughout the project. We are also thankful to the faculty and staff members of the Department of Mechanical Engineering, our individual parents and our friends for their constant support and help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Structural Health Monitoring

Broadly speaking, our project comes under the domain of Structural Health Monitoring (or SHM in short). SHM can be defined as the process of monitoring engineering structures under continuous load or operation [3]. When this is done by periodically collecting data pertinent to their health via suitable sensors it can be termed "vibration-based" structural health monitoring [5]. This process is crucial to ensure that the structures and components perform throughout their intended life and possibly, enable prompt action in case of premature failure. SHM can also be viewed as a predictive maintenance strategy. If this is done on a continuous basis, then it can be termed Condition-Based Monitoring or CBM.

1.2 Elements in SHM

Any SHM system consists of:

1. The structure or the component
2. A data acquisition system
3. A data transfer and storage system
4. A data management system
5. Data interpretation and diagnosis

Data acquisition is done using suitable sensors like accelerometers, Inertial Measurement Units (IMUs), thermocouples, etc. This data is then stored suitably, cleansed, and assessed using suitable statistical techniques. This phase is called the Structural Health Assessment (SHA) phase which broadly involves:

1. Damage detection
2. Damage localization
3. Damage classification
4. Quantifying damage severity

Any industry-scale SHM project goes through four main phases:

1. **Feasibility / Operational assessment**: This phase determines the operational conditions where SHM can be performed. These are chosen such that there are meaningful gains with the project.

2. **Data acquisition:** This is the next phase where the required sensor hardware, number of sensors, their locations, and the frequency of data collection is decided. These variables are highly application-specific and are also driven by the economic constraints of the problem at hand. Raw data from sensors is not directly used for analysis due to significant variability in the surrounding conditions and hence the inherent noise in the data. Raw data acquired is cleansed/de-noised and normalized.

3. **Processing:** This step involves two processes, namely, feature extraction and compression. The feature extraction step involves carefully identifying suitable quantities or transforms on the cleansed data that enable accurate identification of damage and offer an objective reflection of the life of the inspected component. Furthermore, since acquisition happens over an extended period of time, industrial systems often end up with mammoth amounts of sensor data, requiring an additional *compression* step that does the job of throwing away redundant data and retaining meaningful data.

4. **Modelling:** Additionally, one can also develop statistical models tailored to an application to demarcate between faulty and healthy structures or possibly estimate the "state" of the structure, which in turn is critical to prognosticate impending failure. Primarily, this requires the identification of suitable algorithms that can discriminate and predict the state of damage to the structure. Depending on the availability of data, they fall into two classes:

    (a) **Supervised Learning:** When examples of both the damaged and undamaged structures are available. Classification and regression are common examples.

    (b) **Unsupervised Learning:** When examples of either the damaged or the healthy structure are not available. Anomaly/rare/extreme event detection algorithms like Hidden Markov Models and Long Short Term Memory (LSTM) neural networks are candidate examples. Note however that the choice of these algorithms is again highly application-specific and a decision on the same requires careful analysis.

1.3 Applications

SHM has traditionally been used to inspect the life of bridges [5] by comparing mode shapes from numerically computed data and sensor data. Furthermore, this practice had been used extensively in off-shore oil and gas and aerospace industries.

In the offshore oil and gas industry, this was mainly targeted towards failing drilling equipment and monitoring expensive oil-pump lives so that they were not rendered inoperable [5]. For instance, off-shore oil rig platforms were monitored continuously for mass addition due to marine growths, and drilling pipes for their mode shapes. Mode-shapes offered information on the failure state due to the fact that certain higher vibration modes didn't get excited due to the presence of faults.

However, Carden *et al.* [5] also note that the first successful application of vibration-based SHM was for monitoring rotating machinery. Statistical pattern recognition-based methods were employable for such setups due to the ease of data collection and reduced scale as opposed to the more harsh environments that underwater drilling pipes faced. Therefore, the monitoring of rotating equipment enjoys benefits that traditional SHM doesn't, leading to the growth of data-driven techniques like Neural Networks or Machine Learning instead of more classical approaches. Several other methods that were used previously are available in greater depth and detail in Carden *et al.*[5].

As noted before, rotating equipment SHM is much more feasible due to easier access to data. This also opens up new possibilities for using current state-of-the-art ML algorithms. This motivated us to explore more on this front and hence, the rest of the thesis focuses on rotating equipment. This includes shafts, bearings, gears, blades, hubs, and a plethora of other components typically encountered in rotodynamic machines like compressors, and turbines. However, a full and detailed exploration of SHM of each of these components is beyond the scope of this thesis. This follows directly from the fact that SHM is fundamentally application-specific and thereby, component-specific. Therefore, we decided to explore the two most commonly encountered rotodynamic components-namely shafts and bearings. The rest of the thesis is organized as follows. Chapter 2 expounds further on the methods that have been used for fault-detection in shafts and bearings by reviewing the literature in the area. Chapters 3 and 4 detail the algorithms we have implemented for a transverse fatigue crack in a shaft and for bearing faults respectively. Finally, Chapter 5 summarizes and concludes our findings. The codes we wrote for the purpose of this thesis can be found additionally in the Appendices.

CHAPTER 2
LITERATURE REVIEW

2.1 Single fault detection

On-line detection of cracks in shafts from vibration data is a popular fault detection problem. Shafts are one of the most commonly encountered mechanical components in rotating equipment. Lathes, turbojet engine compressors, turbines, wind-turbines, and a host of industrial process and utility plant equipment house rotating shafts in one subsystem or another. However, continuous operation and harsh working conditions (in gas-turbines for instance) can result in the development of cracks, ultimately leading to failure. Furthermore, misalignment in shafts can also result in excessive vibration. Hence, fault-detection and diagnosis of rotating shafts has been an active area of research. The origin of these cracks can be attributed to factors that cause stress-concentration, which initially give rise to *microcracks*, that ultimately propagate to form larger cracks. Several factors like sudden changes in geometry or surface imperfections lead to localized stresses that cause cracks. In case of steam and gas-turbines, thermal stresses additionally contribute to their formation.

Transverse fatigue cracks are a special type of cracks that are perpendicular to the axis of rotation of the shaft [2]. Though the initial formation of the crack can occur in any other orientation, the propagation takes place radially [17][2]. Propagation velocities of such cracks can be quite small ranging from 2500 hours of operation to 101000 hours [2] (*Secton 1.2, Chapter 1*). Furthermore, their occurrence is further augmented as they get lighter[17]. Several methods have been proposed to ascertain and prognose their propagation during operation. Classical models using *fracture mechanics* have been proposed in [2]. However, over the years, it has been noted that statistical pattern based vibration signature analysis is much more accurate and efficient [15] making SHM a big data problem [8]. Commonly used features are the 1X-3X components [15] however these also show possibilities of misalignment. Nicoletti *et.al* propose an Approximated Entropy Algorithm to differentiate between these faults. Lu *et.al* [13] develop a union method that works on the wavelet transform of the Acoustic Emission (AE) signal to decompose the signal and localize the crack. A.S.Shekhar [19] develops a dynamical-systems based crack-detection algorithm that uses an FFT to determine *on-line* crack parameters while modelling the rotor with Finite Elements. Sagi Rathna Prasad and Shekar [17], for the first time in literature, propose a completely statistical feature based algorithm for on-line transverse fatigue crack detection.

2.2 Multiple fault detection

Note however that this analysis is ideal only for detecting a single fault in the vibration data. In the event that the signal possesses data corresponding to multiple faults, the pipeline has to be modified by adding suitable clustering algorithms to achieve separation between the various faults. Based on our review, we find that rolling element bearing is a critical component present in every rotating machinery whose function is to support the machines and permit the rotation of shafts with respect to a fixed structure[11] and hence fault detection is important in these elements. More than 90 percent of rotating machines use bearings to support the machine structure and sudden breakdown of this element may cause fatal failure of the machines resulting in loss of production, increased downtime, and higher risk for safety [12, 16]. Hence, there is a need for maintaining and monitoring the condition of bearings in high production volume systems where the number of rotating machines are very high. Also, there is a need to identify any defect in bearing on time so that damage to machinery or production can be avoided. Thus for condition monitoring and quality inspection of the bearings, such defect detection is of vital importance. When the bearings are loaded radially, they generate vibration even when they are geometrically perfect and the presence of defects in them causes even more significant increase in vibration level[22, 20]. Several researchers have explained the mechanism of vibration in the bearing[20, 24, 21, 7] by various methods like shock pulse method, acoustic emission technique. A variety of factors, the most commonly encountered including the wear, fatigue, faulty installation, corrosion, plastic deformation, brinelling, poor lubrication, and incorrect design, causes premature failures in bearings. The identification of such defects or more importantly which element it occurs in and vibration produced by them is important for condition monitoring of bearings.

Traditional methods like vibration measurement in time, frequency, time-frequency domains, shock pulse methods, acoustic emission techniques and vibration signal processing techniques have been reviewed by previous researchers. This thesis* focuses on the data driven techniques which is a fast growing research domain. Choudhary *et al.*[6] has used the Convolutional Neural Network (CNN) method for bearing fault diagnosis using thermal images as input to the CNN algorithm. The supervised Isometric Mapping (ISOMAP) was used to extract lower-dimensional data from original data and these features were fed in the Grasshopper Optimization-SVM method to classify various bearing faults was done by Wang *et al.* [23]. Ding *et al.*[9] applied an LSTM-based prognosis method for journal bearing. William Gousseau *et al.* [10] have also extensively analysed the rolling element bearing dataset of University of Cincinnati using modern signal processing techniques.

CHAPTER 3

SINGLE FAULT DETECTION: SHAFT

From the literature review we presented pertaining to shafts (*Chapter 2, section 2.1*), it is clear that there may arise several faults in a rotating shaft, commonly encountered of which is the *transverse fatigue crack*. Furthermore, statistical pattern analysis and modelling has proven more fruitful compared to more classical, *physics-based* models to predict such cracks. Central to any such endeavor, is the availability of suitable data-sets on which such an analysis can be performed. We were fortunate to have access to the accelerated fatigue test data-set outlined in Sagi Rathna Prasad and Shekhar [17] and also close interactions with the first author at *IIT Madras*. In addition the work outlined in [17] also happens to be representative of the state-of-the-art completely-statistical SHM techniques, providing us a strong motivation to use the work as a good starting point. The source of the dataset is an *accelerated fatigue test* experiment where the process of crack generation and propagation is artificially fastened up. While Bachschmid *et.al* [2] note this propagation speed to be in the $\mathcal{O}(10^4 - 10^5)$ hours, the work outlined in [17] reports time to *full-failure* as 790s. Hence details of the experimental facility, the procedures adpoted and the data-collection methodology are detailed in subsection 3.2. The dataset we received is further described and investigated in subsection 3.2.1. While [17] perform a PCA-based statistical analysis on this obtained data, we explored alternative ML methods and the methodology for the same is outlined in subsection 3.3. We summarize and conclude our findings in 3.4.

3.1 Experiment

3.1.1 Test Rig & Sensor Hardware

All details of the test-rig components are presented along with the ASTM standards followed in Table 3.1 and the sensor stack used to collect data in Table 3.2. The exact layout is available in Figure 3.1. The experiment is carried out on a mild-steel shaft with a V-notch to (the specifications are available in Table 3.1) to artificially create stress-concentration, which in-turn can lead to the propagation of the crack. The frequency of data-collection using the DAQ is 2kHz.

Component	Dimensions	ASTM	Material
Shaft	ϕ 0.016m, L 1m	EN-3B 080M15	Mild-Steel
Bearings[1]	–	SKF 6205-2Z	–
V-notch	d=1.5mm, ϕ_R,0.02 mm, 60^o	ASTM STP924	–
Impulse hammer	100 mV/lbF	5800B2	Dytran –

Table 3.1: Description of experimental facility

Sensor	Location	Specs
Accelerometer	Bearings	DYTRAN 3145AG
Vibrometer	Shaft	RLV-5500
Strain gauge	Shaft	T1-PCM-IND, KMT telemetry
Tachometer	–	–
DAQ	–	Dewe-43 V

Table 3.2: Sensor hardware stack

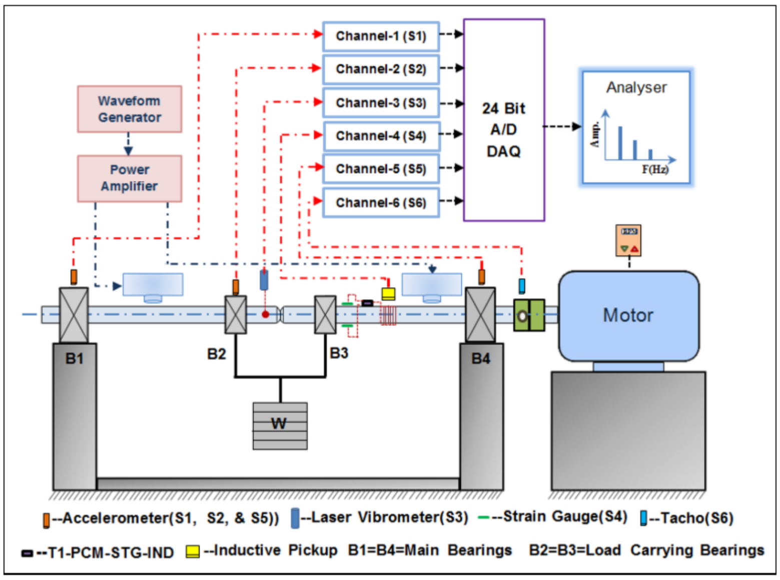


Figure 3.1: Experimental facility in the work by Sagi Rathna Prasad *et al.* [17]

3.1.2 Experiments Conducted

In order to achieve effective condition monitoring results, the fatigue test is conducted at a constant speed of 990 r/min (16.5 Hz). To trigger vibration of the first two natural frequencies of the shaft, a waveform generator is used to generate a random profile, shape-burst in nature. In-addition, a power amplifier is used to control excitation levels. Throughout the fatigue test, appropriate parameters such as the RMS, kurtosis, standard deviation and crest-factor are monitored continuously at all the sensors during the entire

[1]Self-sealed spherical roller

course of the fatigue test. The test is terminated whenever a sudden rise in the monitored parameters in observed within $780 - 790$ s.

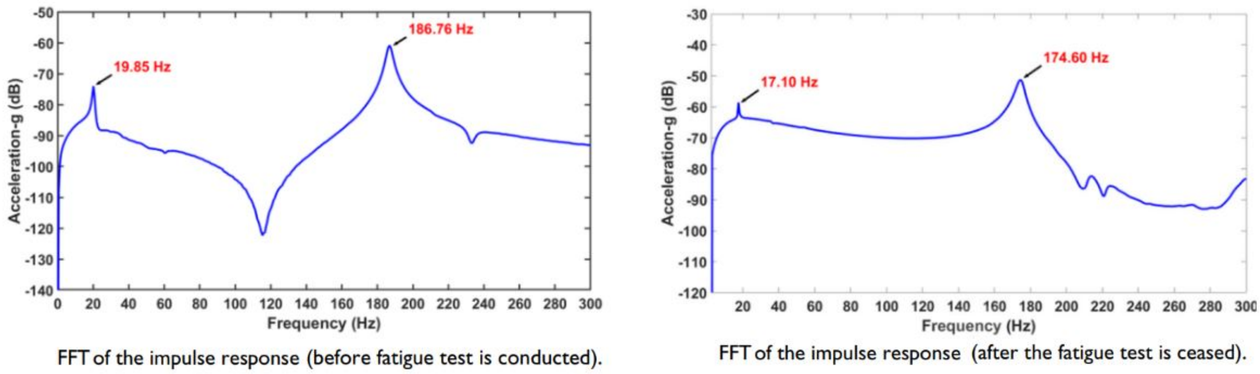


Figure 3.2: FFT of Impulse Response shown in Sagi Rathna Prasad and Shekhar [17]

To ensure that the spurious rise in the observed metrics indeed arises out of a crack growth (not due to other extraneous parameters such as extreme-misalignment), an impact hammer test is conducted. FFTs hence can be useful in understanding if a crack has indeed developed as the natural frequencies for a cracked shaft are much lower compared to a healthy shaft, as abundantly visible from the figure 3.2, strongly indicating the presence of a crack. To quantify crack depth, X-ray computed tomography was performed. This revealed a 40% penetration into the shaft's radius, ensuring the safe applicability and utilization of the dataset for statistical analysis relavant to crack detection.

3.2 Methodology

We examine the model outlined in [17] for statistical pattern-based fatigue-crack detection. We also explore alternative dimensionality reduction techniques in identifying the crack. The data from vibration and strain sensors (as discussed in 3.1.2), are used for predictive analysis.

The pipeline followed for this project is presented schematically as a flow chart in figure 3.3. It can be thought of as a sequence of four major steps: preprocessing, augmentation/feature extraction, dimensionality reduction and damage detection. The pre-processing step involves segmentation of the dataset into 'm' smaller sub-data sets or time frames of equal size and cleaning the data of any noise. This is done with the help of PCA as a denoising technique. Fault-detection fundamentally involves transforming the raw data to another space that enhances the sensitivity. Hence, one can extract several "features" each of which do this job of transformation. This is performed for each sub-dataset, resulting in 'n' columns. Furthermore, note that these features can make use of both time-domain data and frequency-domain data. Stacking all these features up, we arrive at a global matrix $\mathbf{Y}_{m \times n}$. Since we have essentially increased the complexity of the problem due to a multitude of features to now choose from, we now explore methods that will tell us which of these n features really matter. Dimensionality reduction is hence employed to "fuse" features into a smaller number ensuring variance in the data is retained while redundant data is discarded. On this reduced data-set we now compute parameters such as the Hotelling's T^2-statistic and Q-index, that act as suitable "indicators of damage", to ascertain the existence of the transverse fatigue crack. We also explore a crack-localization technique called Partial Decomposition where the contributions from each of the n features is computed after de-fusing. This shall hence enable the identification of the features that contribute most towards the detection of the crack. Following this, a heuristic metric, called the Fused Health Indicator (FHI) is computed which is purely a function of these features that contribute the most, making it the most sensitive to the onset of the fault. In our thesis it is calculated as the sum of first three highest contributing statistical features as opposed to the multiplication of the features in the work by Rathan *et al* [17].

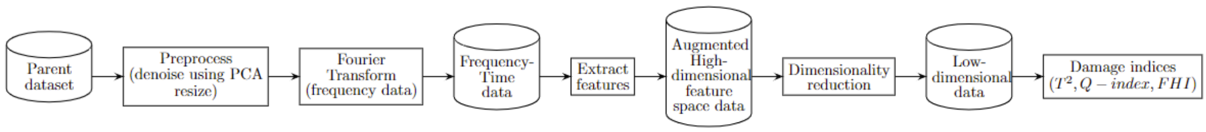


Figure 3.3: Flowchart describing our methodology

3.2.1 Description of dataset

While the work in [17] consists of datasets from several sensors $[S1, S2, S3, .., S6]$, we were provided with only data from sensor $S2$, which is an accelerometer. This single dataset itself is composed of 15.6 million data points covering a physical time of 780 seconds.

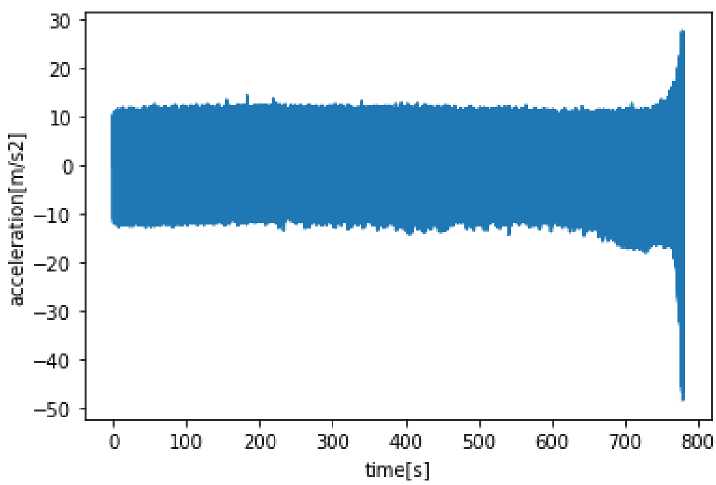


Figure 3.4: Raw Acceleration-Time Data

Hence, following the methodology outline previously, we divide this physical time of 780 s into 390 smaller sub-data sets. Therefore, each sub-dataset contains data spanning 2 seconds of physical time. This resized matrix $\mathbf{S}_{m\times s}$, can be represented as -

$$\mathbf{S}_{m\times s} = \begin{bmatrix} s_{11} & s_{12} & \dots & s_{1s} \\ \vdots & & \ddots & \vdots \\ s_{m1} & s_{m2} & \dots & s_{ms} \end{bmatrix}_{m\times s} = \begin{bmatrix} \mathbf{s_1} \\ \mathbf{s_2} \\ \vdots \\ \mathbf{s_i} \\ \vdots \\ \mathbf{s_m} \end{bmatrix}_{m\times 1}$$

Where,

$$\mathbf{s_i} = \begin{bmatrix} x_{i1}, x_{i2}, \dots, x_{is} \end{bmatrix}_{1\times s}$$

Note that the rows of this matrix represent a single subdataset and that the columns span a physical time of 2s. Thus a total of 390 rows retrieves the physical time of 780 worth of acceleration data.

During the course of the experiment, there is every possibility of external noise from the evironment creeping into the vibration data being recorded by the sensors. Sources include both mechanical and electrical, thus, one needs to denoise this data before attempting any statistical analysis for fault detection. While classicaly, PCA can be used for dimensionality reduction, it has also been applied to reduce noise from signals. The mathematics behind PCA essentially involves Eigen-decomposition and reconstruction of the parent data-matrix $\mathbf{S}_{m\times s}$. Eigenvalues that arise out of this decomposition can be proven to be the variance contained in the resulting "reduced" dataset. Hence, the eigenvalues corresponding to the noise are smaller compared to the signal frequency components.

Denoised signal reconstruction is hence done by retaining only the largest eigenvalues, specifically, those that are greater than 1. Once the data is denoised the filtered matrix is restored or re-written in $\mathbf{S}_{m\times s}$ for further procedures. While classical denoising techniques introduce distortions in the phase-space, PCA-based denoising is known to keep it untouched or with minor skewing.

The next step hence is the feature extraction step, where we increase the size of the matrix by extracting features that are highly sensitive to even minor changes. Several features exist that make use of both time and frequency domain data. Works like [4] extract as high as 172 features. In this work, we extract 18 features. Frequency domain features need data transformed into the frequency space, hence, requiring an additional step of a 2-D Fast Fourier Transform. Historically, the 1X components and the Band powers were found to be good crack or fault indicators in general. These are frequency domain features. This fact can be further verfied using this analysis by computing their contributions.

Feature Name	Description of Features	Expression
Time Domain		
Mean	Expected Value	$\frac{1}{n}\sum_i^n y_i$
Root-Mean-Square	Normalised Power Content	$\sqrt{\frac{1}{n}\sum_i^n y_i^2}$
Crest Factor	Signal's impulsiveness	$\frac{max(\lvert y_i \rvert)}{RMS}$
Peak-to-Peak	Separation between max and min	$max(y_i) - min(y_i)$
Energy	Absolute Power Content of signal	$\sum_i^n y_i^2$
Skewness	Asymmetry of Signal's PDF	$\frac{\sum_i^n (x-\mu)^3}{(n-1)\sigma^3}$
Kurtosis	Tails of signal's PDF[2]	$\frac{\sum_i^n (y_i-\mu)^3}{(n-1)\sigma^4}$
Shape Factor	Affected by object's shape	$\frac{RMS}{\mu_{\lvert y_i \rvert}}$
Impulse Factor	Level of impact due to fault	$\frac{max(\lvert y_i \rvert)}{\mu_{\lvert y_i \rvert}}$
Margin Factor	Degree of impact due to fault	$\frac{max(\lvert y_i \rvert)}{\mu^2_{\lvert y_i \rvert}}$
Kurtosis Factor	Signal's level of peakedness	$\frac{Kurt}{RMS^4}$
Hybrid Kurtosis	Spread about centroid	$\frac{\sqrt{Kurt \times \sigma}}{n}$
Frequency Domain		
Shaft Order I	Magnitude of 1X Component in Fourier Spectrum	1^{st} peak in $\mathcal{F}(y_i)$
Shaft Order II	Magnitude of 2X Component in Fourier Spectrum	2^{nd} peak in $\mathcal{F}(y_i)$
Shaft Order III	Magnitude of 3X Component in Fourier Spectrum	3^{rd} peak in $\mathcal{F}(y_i)$
Band Power I	Average power of 1X freq band	$\sqrt{\sum \frac{y_i^2 \bar{y_i}}{N} 3}$
Band Power II	Average power of 2X freq band	$\sqrt{\sum \frac{y_i \times \bar{y_i}}{N}}$
Band Power III	Average power of 3X freq band	$\sqrt{\sum \frac{\bar{y_i} \times y_i)}{N}}$

Table 3.3: Statistical Time and Frequency Domain Features

[3]Probability Density Function
[3]$\forall y_i \in [y_i^{1X} - h, y_i^{1X} + h]$, similarly for other harmonics

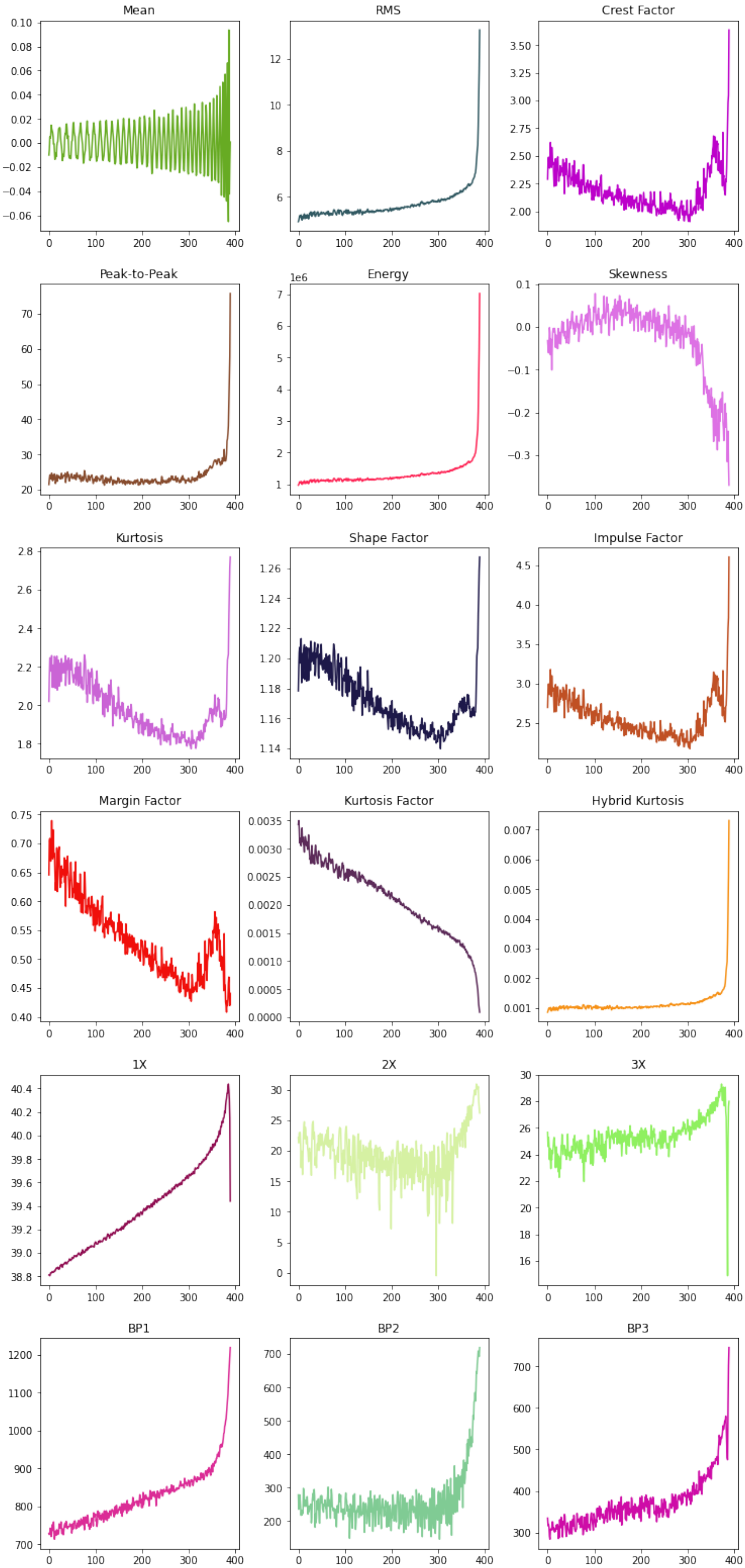
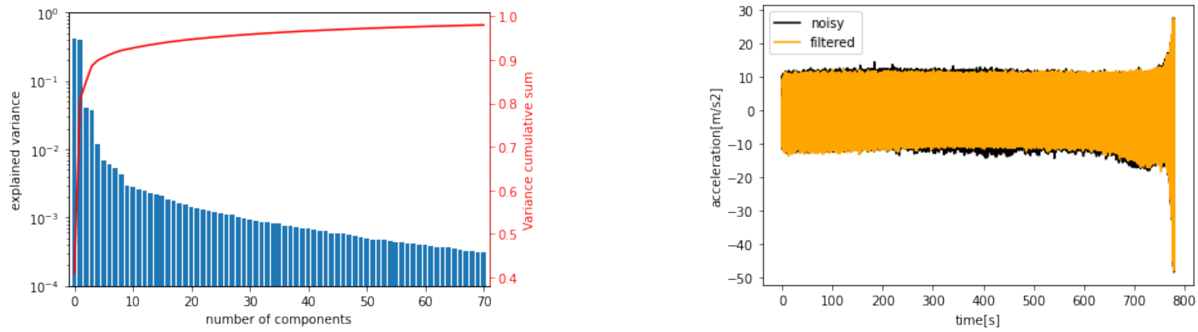


Figure 3.5: 18 Statistical Time and Frequency Domain Features

A list of all the features exted for this body of work is shown in table 3.5. The Feature Matrix can be represented by $\mathbf{Y}_{m \times n}$, where 'm'is the total number of sub-data sets (m = 390) and 'n' is the total number of features extracted (n = 18).

$$\mathbf{Y}_{m\times n} = \begin{array}{c} \\ t_1 \\ t_2 \\ \vdots \\ t_m \end{array} \begin{array}{c} \begin{array}{cccc} F_1 & F_2 & \dots & F_n \end{array} \\ \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & x_{m2} & \cdots & y_{mn} \end{bmatrix}_{m\times n} \end{array}$$



(a) PCs Variance Explained

(b) Denoised Data

Figure 3.6: Denoising using PCA

3.2.2 PCA implementation

Usually there is redundancy of information due to inter dependency of features in a high-dimensional matrix. Hence, a single feature is used to replace all these correlated features using dimensionality reduction techniques. This results in transforming the n-dimensional space into a new coordinate system of 'p' dimensions (where $p < n$) by preserving most of the variance possible.

Finally, the feature matrix $\mathbf{Y}_{m \times n}$ obtained is used to build the PCA Model. To obtain equal weights in the PCA model, the features with different units and magnitudes are normalized to zero mean and unit variance i.e. the data we essentially operate upon is the $Z-$ score. Therefore PCA decomposes the feature matrix $\mathbf{Y}_{m \times n}$ into a new set of 'n' features that are "linear combinations" or "fused". These can also be called as *principal components*. PC matrix or Transformed data matrix or score matrix obtained is as follows-

$$
\mathbf{T}_{m\times n} = \mathbf{P}_{m\times n} = \begin{array}{c} \\ \mathbf{l_1} \\ \mathbf{l_2} \\ \vdots \\ \mathbf{l_m} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{PC^{(1)}} & \mathbf{PC^{(2)}} & \dots & \mathbf{PC^{(n)}} \end{array} \\ \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mn} \end{bmatrix}_{m\times n} \end{array}
$$

where, $\mathbf{l_i}$ s are **Loading Scores** for each Principal Component $\mathbf{PC^{(i)}}$ and the coefficient matrix also called PC loading matrix is denoted by $\mathbf{P}_{m\times n}$. The columns of $\mathbf{P}_{m\times n}$ are arranged in the descending order of component variance magnitude and these PCs are mutually orthogonal to each other. In short, the eigenvalues of the covariance matrix $\{Cov(\mathbf{f_i}, \mathbf{f_j})\}_{n\times n}$ are the PC variances. The projection of initial dataset in the direction of these PCs is called the transformed data matrix denoted by $\mathbf{T}_{m\times n}$, also called the score matrix. Further, to attain unit variances of the normalised scores, the matrix $\mathbf{P}_{m\times n}$ is scaled using eigen value matrix. The mathematical expressions to obtain scaled $\mathbf{P}_{m\times n}$ and normalized scores through diagonalization is as follows-

$$
\mathcal{K}_{n\times n} = \mathbf{P}_{n\times n}\mathbf{L}_{n\times n}\mathbf{P}^T_{n\times n}
$$

where:

$$
\mathbf{L}_{n\times n} = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}
$$

we normalize $\mathbf{P}_{n\times n}$ as:

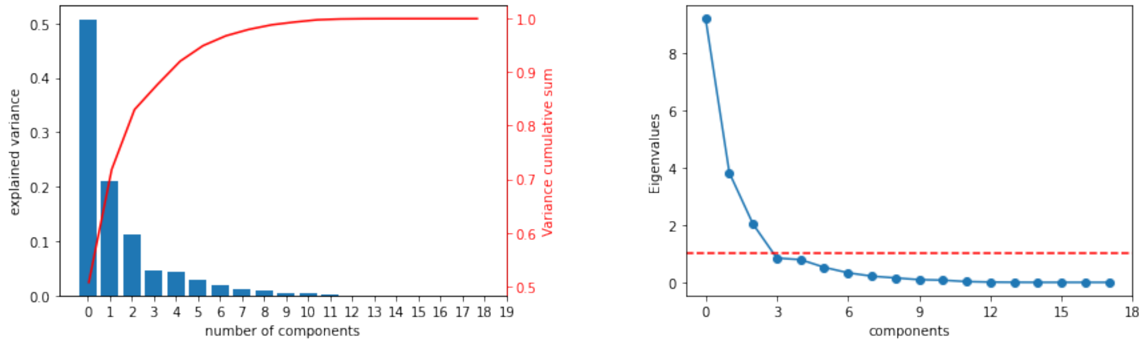
$$
\mathbf{P}^s_{n\times n} = \mathbf{P}_{n\times n}\mathbf{L}^{-1/2}_{n\times n}
$$

where:

$$
\mathbf{L}^{-1/2}_{n\times n} = \{\frac{1}{\sqrt{L_{ij}}}\}_{n\times n} = diag\left[\tfrac{1}{\sqrt{\lambda_i}}\right]_{n\times n}
$$

To reduce the number of dimensions of the scaled matrix $\mathbf{P}^s{}_{n\times n}$, we begin dropping the last $n-p$ entries. Hence, the reduced, scaled and transformed matrix is obtained by projecting $\mathbf{Y}_{m\times n}$ on $\mathbf{P}^{sr}_{n\times p}$

$$
\mathbf{T}^{sr}_{m\times p} = \mathbf{Y}_{m\times n}\mathbf{P}^{sr}_{n\times p} = \begin{array}{c} \\ \mathbf{l_1} \\ \mathbf{l_2} \\ \vdots \\ \mathbf{l_m} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{PC^{(1)}} & \mathbf{PC^{(2)}} & \dots & \mathbf{PC^{(p)}} \end{array} \\ \begin{bmatrix} p^{sr}_{11} & p^{sr}_{12} & \cdots & p^{sr}_{1p} \\ p^{sr}_{21} & p^{sr}_{22} & \cdots & p^{sr}_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ p^{sr}_{m1} & p^{sr}_{m2} & \cdots & p^{sr}_{mp} \end{bmatrix}_{m\times p} \end{array}
$$



(a) Explained Variance (b) Eigen Values

Figure 3.7: PCA on the shaft dataset

where the diagonal matrix with eigenvalues as the principal diagonal elements are denoted by $\mathbf{L}_{n\times n}$, and $\mathbf{T}^{sr}_{m\times p}$ is the reduced transformed data matrix; $\mathbf{P}^{sr}_{n\times p}$ is the loading matrix with a reduced number of PCs. The hidden pattern in the dataset is revealed by the highest order of variance represented by $(L > 1)$. To reduce dimensionality hence we choose a reduced number of PCs $(p < n)$ resulting in a modified feature matrix that can now be used for crack detection. The equations to obtain reconstruction from projected space to original space using a reduced PCA model are as follows-

$$\mathbf{Y}^{rt}_{m\times n} = \mathbf{T}^{sr}_{m\times p}\mathbf{P}^{sr^T}_{p\times n} = \mathbf{Y}_{m\times n}\mathbf{P}^{sr}_{n\times p}\mathbf{P}^{sr^T}_{p\times n}$$

Residuals or errors hence is simply the difference between reconstucted data and old data:

$$\mathbf{Y}^{rd}_{m\times n} = \mathbf{Y}_{m\times n} - \mathbf{Y}^{rt}_{m\times n} = \mathbf{Y}_{m\times n} - \mathbf{T}^{sr}_{m\times p}\mathbf{P}^{sr^T}_{p\times n}$$

$$= \mathbf{Y}_{m\times n} - \mathbf{Y}_{m\times n}\mathbf{P}^{sr}_{n\times p}\mathbf{P}^{sr^T}_{p\times n}$$

$$= \mathbf{Y}_{m\times n}(\mathbf{I}_{m\times n} - \mathbf{P}^{sr}_{n\times p}\mathbf{P}^{sr^T}_{p\times n})$$

$\mathbf{Y}^{rt}_{m\times n}$ represents the constructed data using p-dimensional space; I is the identity matrix and the residual matrix obtained using $(n-p)$ dimensional space is denoted by $\mathbf{Y}^{rd}_{m\times n}$

The results of explained variance and eigen values of the components obtained through PCA Implementation is as follows-

3.2.3 Damage Indices for Crack Detection

A variety of damage indices can be found in the work by Alcala *et al.* [1]. Our primary interest is to understand and calculate Hottelling's T^2-statistic and $Q-$index.

Hottelling's T^2-statistic:

The measure of variations in scores within principal subspace is given by T^2-statistic. The equation is defined as the sum of the squares of standardized scores corresponding to each variable as shown below -

$$T_j^2 = \mathbf{y_j}_{1\times n}\mathbf{D}_{n\times n}\mathbf{y_j}^T_{n\times 1}$$

$$\mathbf{D}_{n\times n} = \mathbf{P^{sr}}_{n\times p}\mathbf{P^{sr}}^T_{p\times n}$$

Hence note that:

$$T_j^2 = \mathbf{y_j}_{1\times n}\mathbf{D}_{n\times n}\mathbf{y_j}^T_{n\times 1} = T_j^2 = \{\mathbf{y_j}_{1\times n}\mathbf{P^{sr}}_{n\times p}\}\{\mathbf{P^{sr}}^T_{p\times n}\mathbf{y_j}^T_{n\times 1}\}$$

$$= \{\mathbf{y_j}_{1\times n}\mathbf{P^{sr}}_{n\times p}\}\{\mathbf{y_j}_{1\times n}\mathbf{P^{sr}}_{n\times p}\}^T = \mathbf{T^{sr}}_{1\times p}\mathbf{T^{sr}}^T_{p\times 1}$$

Hence we can compute the above index $\forall\ j \in \{1, .., m\}$

The results of Hottelling's T^2-statistic executed in python code is as follows-

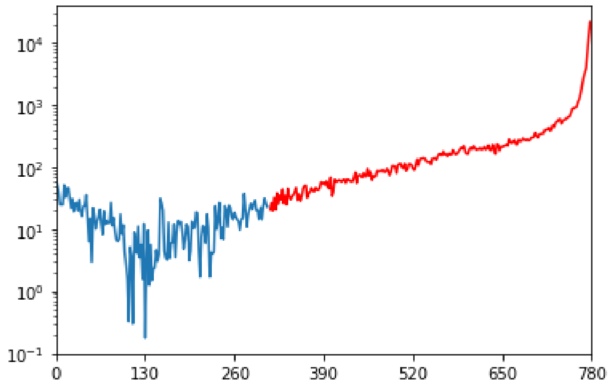


Figure 3.8: Hottelling's T^2-statistic

Q-Index:

The deviations of data projected in the residual subspace that is not obtained in PCA model is analysed by Q-index. It is denoted as the sum of the squares of the residual matrix $\mathbf{Y}^{rd}_{m\times n}$.

$$Q - index = \mathbf{Y}_{1\times n}(\mathbf{I}_{n\times n} - \mathbf{P}^{sr}_{n\times p}\mathbf{P}^{sr}_{p\times n})\mathbf{Y}_{n\times 1} = \mathbf{X}^{rd}_{1\times n}\mathbf{X}^{rd}_{n\times 1}$$

The results of Q-index executed in python code is as follows-

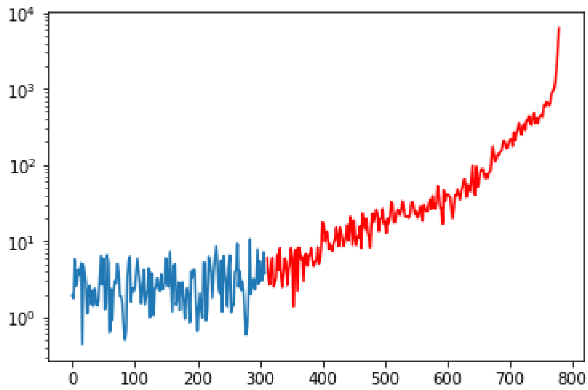


Figure 3.9: Q-index

3.2.4 Normal Operating Region

Damage indices are ideal transformed indicators compared to the original feature set. However, the next challenge is to carefully decide their limits or threshold values beyond which we can safely say that a crack has indeed developed. Below this threshold, the data is healthy and beyond, there's a crack. The threshold values are computed in [17] by fitting a kernel density function on the damage-index data and ascertaining where 99% of the data lies in the sample space. The region where majority of this healthy data lies, hence, is called the Normal Operating Region or NOR for short.

Consider this sample to be denoted by:

$$\mathbf{s_h} = \begin{bmatrix} s_1 & s_2 & \dots & s_n \end{bmatrix}$$

The Kernel density function defined below gives the distribution of the sample:

$$K(s_{m_i}) = K(\frac{s - s_i}{h})$$

Kernel density estimate is computed by summing up all the healthy distributions:

$$\hat{f}(s) = \sum_{i=1}^{n} \frac{K(s_{m_i})}{nh}$$

Then the cumulative distribution of $\hat{f}(s)$ can hence be obtained by integrating upto the threshold as:

$$\hat{f}(s) = \int_0^{s_{thresh}} \hat{f}(s)ds$$

The control limit or threshold value of the sample space is hence the point below which $100(1 - \alpha)$th percentile of the data lies. This is done on the Kernel Density using (where

TV represents the "Threshold Value"):

$$TV = F_h^{-1}(s)(1-\alpha)$$

Nonetheless, fitting a KDE and computing the inverse for the threshold from the integral proved to be a challenging task. Hence the essence of the procedure was still retained by visualizing the distribution of the indices as histograms (with really fine bins) and computing the X_{thresh} at which 80-90% of the distribution lay. Therefore, the point at which the corresponding damage indices crossed the threshold will give us the time of onset of crack based on either of them. This hence gives us a validation metric. If the PCA model was implemented correctly it would predict the time of onset of the crack quite close to the paper [17]. Also note that this cross-over occurs much later for the T^2-statistic compared to the Q-index and based on the phase-time plots, the crack was predicted accurately by the Q-index rather than the T^2 for even in [17]. We also observe similar results. The results of NOR computed for each damage index in python code is as follows-

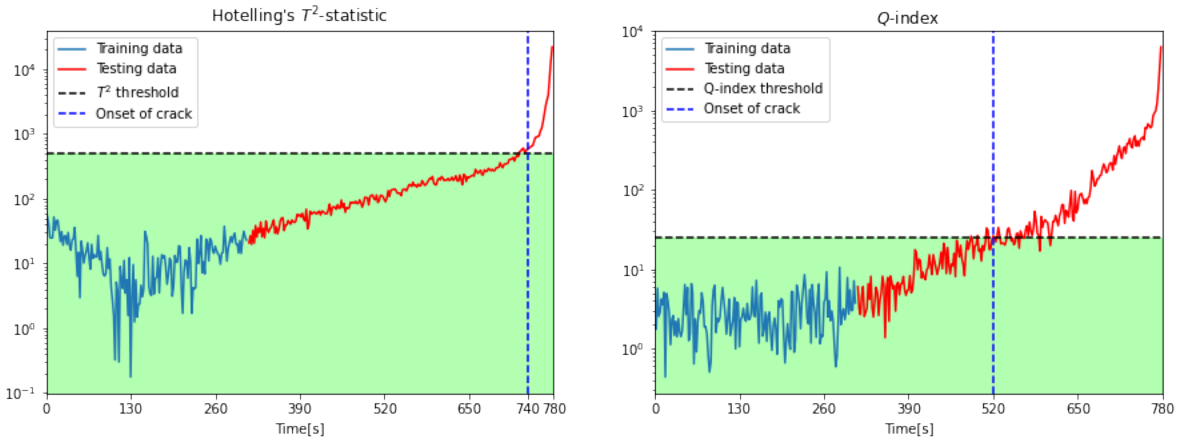


Figure 3.10: NOR for Damage Indices

The NOR threshold for for T^2-statistic is found at 740^{th} timestamp and for Q-index is found at 520^{th} timestamp. Also note that this model develops the PCA Principal components using the training data which forms the initial 40% of the dataset. Therefore, the loading matrix, and eigen values correspond to the training data as it is assumed that the damaged data will not be available in real time monitoring. Hence, the model, after due "training" on the undamaged dataset, is applied in the same manner on the testing dataset which houses the faulty datapoints. The red portions of the damage indices graphs represent the tested data and an out-of-control situation is identified for the same. At this time-step the crack is supposed to have onset.

3.2.5 Partial Decomposition Contribution

When there is an onset of crack, it is imperative to identify features that contribute the maximum for the cause of defect. These sensitive features can be used for further conditional monitoring. Therefore, to identify these features, Partial Decomposition Contribution method is adopted in the case of detecting fatigue crack in rotor shafts. PDC partially decomposes damage indices by summation of feature contribution and obtaining the region with more abnormalities as shown in the equations below -

$$PDC_i^{T^2-index} = \{\mathbf{y_j}_{1\times n}\mathbf{P^{sr}}_{n\times p}\}\{\boldsymbol{\xi}_{1\times n}^{(i)}\boldsymbol{\xi}_{n\times 1}^{T(i)}\}\{\mathbf{y_j}_{1\times n}\mathbf{P^{sr}}_{n\times p}\}^T$$

$$PDC_i^{Q-index} = \mathbf{y_j}_{1\times n}(\mathbf{I}_{n\times n} - \mathbf{P}_{n\times p}^{sr}\mathbf{P}_{p\times n}^{sr})\boldsymbol{\xi}_{1\times n}^{(i)}\boldsymbol{\xi}_{n\times 1}^{T(i)}\mathbf{y_j}_{1\times n}$$

This measured weights of path from different sensors mounted adjacent to each other indicates the region between respective sensors to localize the crack. The highest weighted region reveals the location of the crack. From the results of PDC based on Q-index of data-set obtained below, we can conclude that 1X Harmonic, Band Power and Kurtosis contribute highest to crack detection and localization.

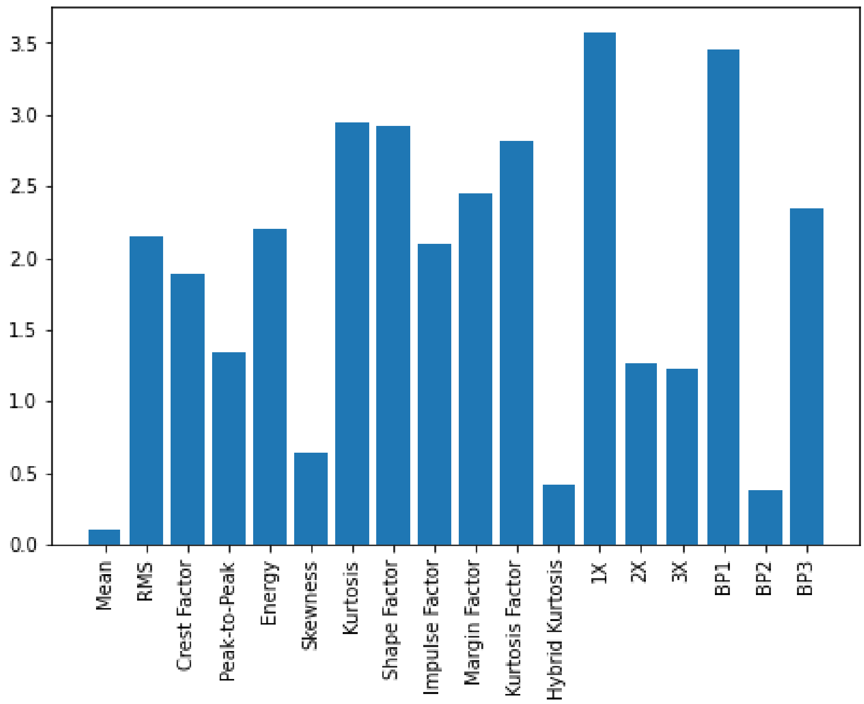


Figure 3.11: Partial Decomposition Contribution

3.2.6 Fused Health Indicator

Fused Health Indicator is developed to monitor the state of critical components of rotating equipment like shafts. Here, the top three highest contributing features obtained from PDC corresponding to the Q-index are taken for the development of FHI. From equation below, FHI is defined as an absolute sum of three highest contributing vectors.

$$FHI = |\alpha_1 + \alpha_2 + \alpha_3|$$

Here, α_1, α_2, and α_3 are three highest contributing vectors comprising of integer values. The resulting FHI is then normalized with respect to its maximum value to rescale values between $(0, 1)$. The results of FHI calculation executed in python is shown below-

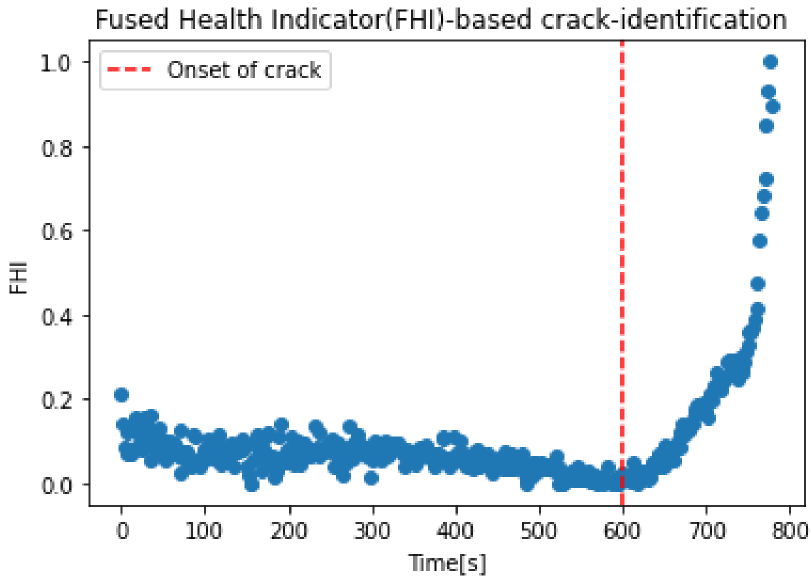


Figure 3.12: Fused Health Indicator

The Fused Health Indicator or FHI is hence an ideal alternative to the raw data and the features that we thus extracted from it. The three most sensitive features (out of the 18 we generated initially) are hence used as indicators. Note that by generating 18 or more features (in our case) we have increased the size of the data-set close to 18 fold or so, which renders the method infeasible for larger datasets. Hence, FHI can also be seen as a method to reduce the size of the parent dataset.

3.3 Alternative ML Methods

This section focuses on alternatives to PCA as a dimensionality reduction method in the pipeline we just described. Our objective was to find methods that are more data efficient and sensitive to the presence of crack, damage indices and result in enhanced data-reconstruction and fusing.

To this end, Linear Discriminant Analysis (LDA), Kernel Principal Component Analysis (KPCA), Independent Component Analysis (ICA), Sparse PCA and Incremental PCA have been suggested. While LDA is primarily not used for dimensionality reduction, it is used on labelled datasets to attain maximum inter-class separation. However, our dataset is unlabelled and needs $a - posteriori$ labelling using methods like PCA that operate on unlabelled data. Hence, Kernel PCA and Independent Component Analysis, Sparse PCA and Incremental PCA that are variants of PCA have been discussed below.

3.3.1 Kernel PCA

While PCA is used to reduce the dimension of the data retaining variance by finding the linear combinations of parent features, it is a linear method which means it cannot provide optimal results with non-linear datasets. The given dataset is non-linear due to the inherent non-linearities in vibration data and noise. Kernel PCA implements linear separability, by projecting the dataset into higher dimension feature matrix using a kernel function. PCA can now be performed on this high dimensional data which is linearly separable achieving better results. The code snippet for Kernel PCA can be found at Appendix A.2. The results of Kernel PCA obtained are as follows-

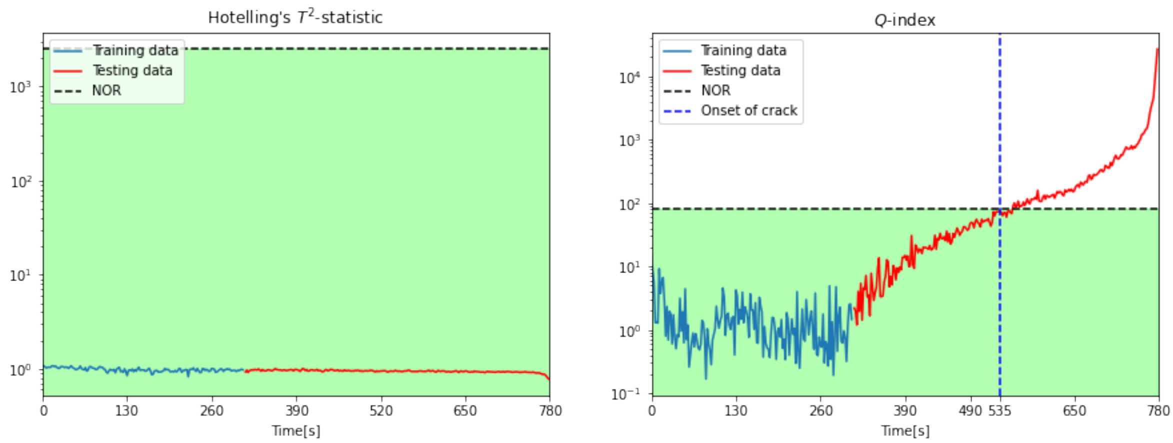


Figure 3.13: Kernel PCA

Kernel PCA gives us 8 PCs with eigenvalue>1 whereas linear PCA gave us only 5 PCs. Hence, Kernel PCA gives more stable features that show high variance. The cosine kernel function gave good separability compared to other kernels like “RBF” or “poly”. However, T^2-statistic of this model is much less sensitive to crack initiation and propagation. The Q-index is still highly sensitive with minor differences in the prediction of the time of onset of the crack.

3.3.2 Independent Component Analysis

Independent Component Analysis, unlike PCA, uses independent components (IC). These ICs may not satisfy the mutual orthogonality but they are independent of other components. This allows us to have lesser redundant features. Through ICA we aim to separate information rather than compressing it. The code snippet for Independent Component Analysis can be found at Appendix A.3. The results of ICA obtained are as follows-

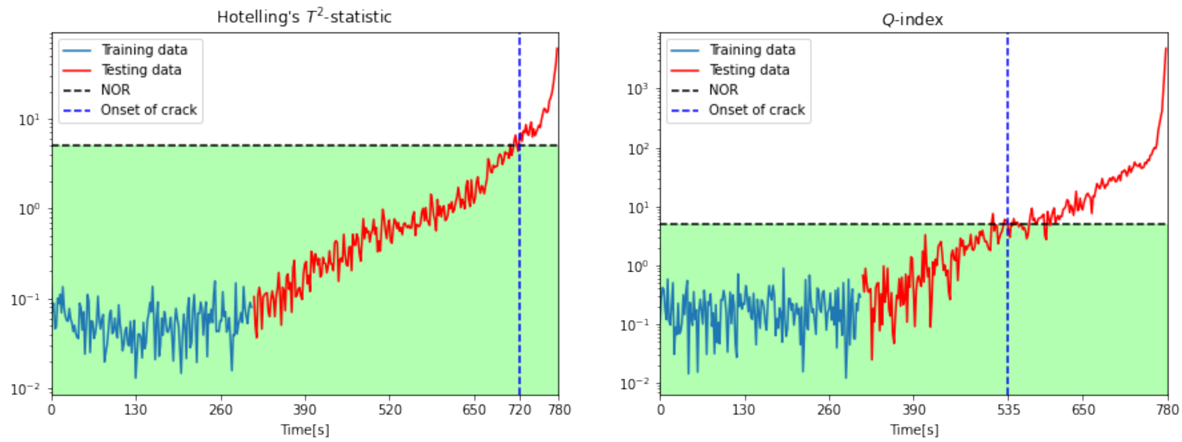


Figure 3.14: Independent Component Analysis

Here, 9 independent components from the raw denoised data are extracted and a good variance in the T^2-statistic and Q-index for the initiation and propagation of the crack is observed. Differences in prediction of the onset of crack aren't radically different once again.

3.3.3 Sparse PCA

A significant drawback of PCA is that, the Principal Components are dense. Since each PC is a linear combination of all original features, it is difficult to interpret. To overcome this drawback, Sparse PCA, a variant of PCA which attempts to produce easily interpretable models through sparse loading is used. In this implementation, each PC is a linear combination of a subset of the original features. This is an advanced technique used in statistical analysis of multivariate data sets. It extends the traditional method of PCA for the dimensionality reduction by introducing sparsity structures to the input features. The code snippet for Sparse Principal Component Analysis can be found at Appendix A.4. The results of Sparse PCA obtained are as follows-

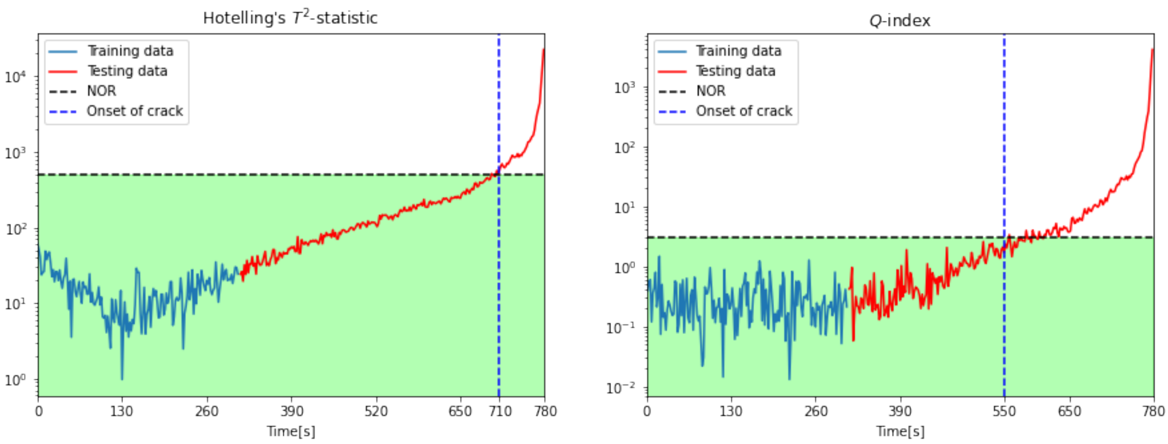


Figure 3.15: Sparse PCA

From the results, it can be observed in Sparse PCA the cross-over points are nearly similar, producing a deviation of 3.4% wrt Q-index compared to the paper [17] results.

3.3.4 Incremental PCA

When the dataset to be decomposed is really huge to fit in memory, PCA is replaced with Incremental Principal Component Analsysis (IPCA). In this technique, a low-rank approximation of the input samples is built using the amount of memory that is independent of the number of input samples. Changing batch size gives the control of memory usage while still making it dependent on the input features. Here the dataset is split into many mini-batches and then we feed one mini-batch at a time to the IPCA algorithm. The code snippet for Independent Component Analysis can be found at Appendix A.5. The results of Incremental PCA obtained are as follows-

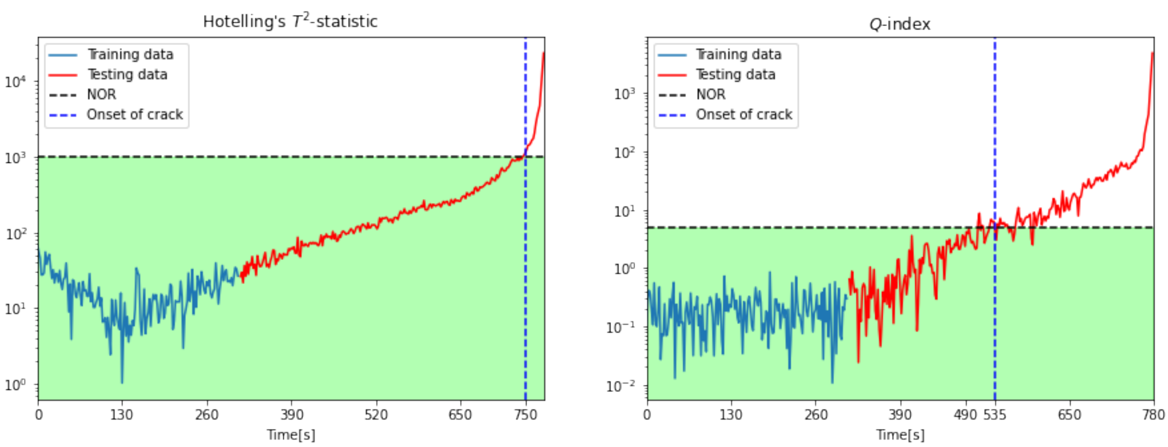


Figure 3.16: Incremental PCA

Incremental PCA can be viewed as an on-the-fly PCA method and hence we expect the results of both Linear PCA and Incremental PCA to be the same. We further validated these results by running PDC on the resulting lower-dimensional data and computing the FHI. The results look identical.

3.4 Conclusion

From the NOR graphs, the table below compares the predicted time of onset of crack for the various methods we tried based on both the damage indices. Since these are the primary objectives of any algorithm, the closer it is to the actual data and results as outlined in [17], the better the algorithm. Also, slightly earlier crack detection is also possibly preferred to allow for sufficient reaction time in case real-time industrial systems.

Method	T^2**-crossover**	**Deviation (%)**	Q**-index crossover**	**Deviation(%)**
PCA by [17]	750	0	530	0
Linear PCA	740	-1.3	520	-1.8
KPCA	–	–	535	1.0
ICA	720	-4	535	1.0
Sparse PCA	710	-5.4	550	3.7
Incremental PCA	750	0	535	1.0

Table 3.4: Comparison of results from various methods

We note that the deviation from the base model in [17] is the least for KPCA, ICA and IPCA while it is the maximum for Sparse PCA. However, the T^2 statistic is unable to predict the onset of crack for KPCA. IPCA is able to satisfactorily recover the results of a Linear PCA. ICA, though not a dimensionality reduction algorithm, is able predict accurately the onset of the crack. It is also worth noting that our implementation of the Linear PCA model shows a slightly earlier onset of crack compared to the original work. This might possibly be due to the omission of the features like IKaZ, spectral kurtosis and entropy which may have a higher say on the detection. Note that their respective average contributions are comparable to that of the three most contributing features. Furthermore, our model was implemented using the pre-coded PCA modules of the Scikit-learn framework. While a slightly modified version of PCA is presented in [17]. Scaling of the PC matrix is done before transforming it to the low-dimensional space.

CHAPTER 4

MULTIPLE FAULT DETECTION: BEARING

4.1 Introduction

Reviewing the literature presented for SHM for bearings(*Chapter 2, section 2.2*) it is evident that bearings are capable of having multiple faults present in multiple components simultaneously. With the growth of research interest in the field of predictive maintenance and improvement in data collection of sensor data allows us to get access to these data-sets. In case of bearings, we have used the data-set recorded by Center for Intelligent Maintenance Systems of the University of Cincinnati which is available under NASA prognostic data repository. The data-set is obtained by performing an endurance test where natural degradation occurs[10]. They have also conducted the endurance tests with artificially initiated defects in order to accelerate the damage growth. The detailed explanation of the experimental set up and the bearings are mentioned in subsection 4.2

4.2 Experimental setup

Test Rig & Sensor Hardware

Four bearings were installed on the shaft with was attached to an AC motor. The motor was coupled with the shaft via rub belts during which the motor maintained a constant speed of 2000 RPM. To induce real life conditions, a 6000 lbs radial load was applied onto the shaft and bearing via a spring mechanism. The lubrication for the bearings were properly maintained by force lubrication that also regulates the flow and temperature. Rexnord AZ-2115 double row bearings were used in the setup which is shown in the fig 4.1. To measure the vibration accurately PCB 353B33 high sensitivity Quartz ICP accelerometers were installed on the bearing housing. The first data-set was formed with 2 accelerometers for each bearing(x and y axes) and remaining two data-sets were formed with 1 accelerometer for each bearing. The sensor placement is also shown in fig 4.1. All failures has occurred after exceeding designed life time of the bearings which was more than 100 million revolutions. The bearing characteristics can be verified in table 4.1.

Rexnord ZA-2115 Characteristics		
Pitch diameter	2.815 inch	71.5mm
Rolling element diameter	0.331 inch	8.4mm
Number of rolling element per row	16	16
Contact angle	15.17	15.17
Static load	6000 lbs	26690 N

Table 4.1: Details of the bearing design [10]

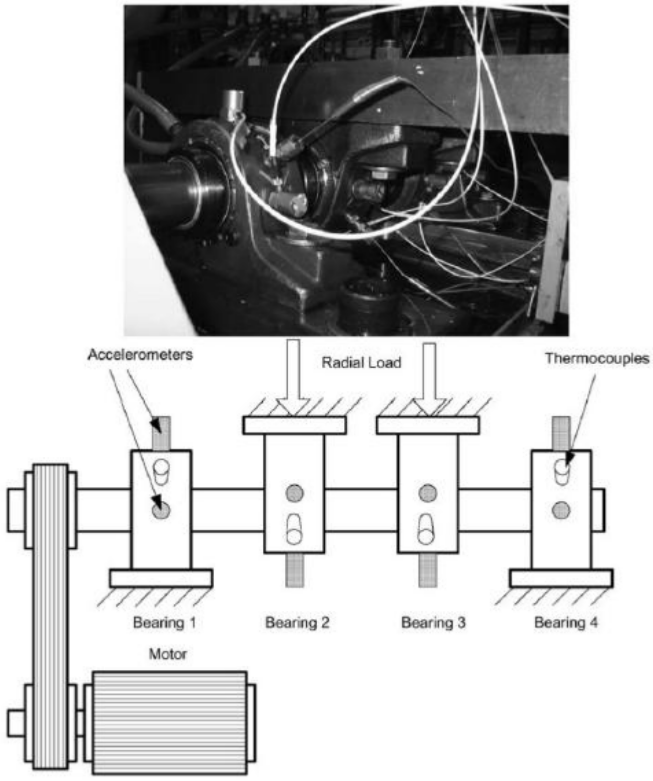


Figure 4.1: Experimental setup corresponding to the dataset take from [14]

Description of data-set

The repository has a compressed file which consists of three data-sets,composed of numerous files which is 6.2GB in size when extracted. Each file is a snapshot of 1 second duration with 20.48KHz sampling rate. The file name indicates when the data has been recorded. The data collection was facilitated by NI DAQ card 6062E. The one second acquisition has been made every ten minutes except for the first data-set for which the first forty three files had been acquired every five minutes. The description of the data-sets is given in table 4.2. The fault frequencies which are used for the traditional mode of diagnosis of the rolling element bearing have been calculated from the bearing characteristics and tabulated in table 4.3.

	Files	Channels	Endurance Duration	Duration of recorded signal	Announced damages at the end of the endurance
Dataset 1	2156	8	34 days 12 hours	36 min	Bearing 3: inner race Bearing 4: rolling element
Dataset 2	984	4	6 days 20 hours	16 min	Bearing 1: outer race
Dataset 3	6324	4	31 days 10 hours	74 min	Bearing 3: outer race

Table 4.2: Bearing Dataset Description in [14]

Characteristic Features	
Shaft frequency	33.3 Hz
Ball Pass Frequency Outer race (BPFO)	236 Hz
Ball Pass Frequency Inner race (BPFI)	297 Hz
Ball Spin Frequency (BSF)	278Hz (2x139 Hz)
Fundamental Train Frequency (FTF)	15 Hz

Table 4.3: Characteristic frequencies of the test rig [10]

4.3 Methodology

In this thesis, we have explored the clustering technique mentioned by Rehab et al. [18] along with principal component analysis to identify and classify the type of defect present in the bearings. The accelerometer data is used for feature extraction and prediction of defect based on these features. The entire data-set consists of 3 sub data-sets with varying defects present in varying bearing in each sub data-set. In each data-set, there are numerous files where each file contains the 20480 samples for each bearing arranged along the columns. This data is extracted and 12 time domain features are extracted from them which are mentioned in the table 4.4. The features are then plotted for visualisation as shown in fig 4.2. These time domain features are then exported as csv file to perform principal component analysis and clustering.

(a) Time domain Features (b) Time domain Features (c) Time domain Features

Figure 4.2: Time Domain features of bearing data

Feature Name	Description of Features	Expression
Time Domain		
Mean	Expected or average Value	$\frac{1}{n}\sum_i^n y_i$
Root-Mean-Square	Power content (normalized)	$\sqrt{\frac{1}{n}\sum_i^n y_i^2}$
Crest Factor	Signal's impulsiveness	$\frac{max(\lvert y_i\rvert)}{RMS}$
Peak-to-Peak	Separation between max and min	$max(y_i) - min(y_i)$
Energy	Absolute Power Content of signal	$\sum_i^n y_i^2$
Skewness	Asymmetry of Signal's PDF	$\frac{\sum_i^n (x-\mu)^3}{(n-1)\sigma^3}$
Kurtosis	Tails of signal's PDF[1]	$\frac{\sum_i^n (y_i-\mu)^3}{(n-1)\sigma^4}$
Shape Factor	Affected by object's shape	$\frac{RMS}{\mu_{\lvert y_i\rvert}}$
Impulse Factor	Level of impact due to fault	$\frac{max(\lvert y_i\rvert)}{\mu_{\lvert y_i\rvert}}$
Margin Factor	Level of impact due to fault	$\frac{max(\lvert y_i\rvert)}{\mu^2_{\lvert y_i\rvert}}$
Entropy	Measure of signal irregularity	$\sum_k p_k log(p_k)$
Clearance factor	Maximum for healthy bearings	$\frac{max(y_i)}{\frac{1}{n}\sum_i^n \lvert y_i^2\rvert}$

Table 4.4: Features extracted from the time domain for the bearing problem

As explained in section 3.2.2, we use PCA to reduce the feature space from 12 to a lower dimensional feature space while retaining the maximum amount of variance to reduce the redundancy of the information in the data set By applying principal component analysis to the feature matrix we notice that the first 2 principal components hold the maximum variance of 85 percent as shown in fig 4.4. We also notice that only 3 principal components have eigenvalues greater than one which can be seen in fig 4.3. Hence we chose to reduce it to 2D feature space thereby retaining the variance and making sure the data is not compressed. Fig 4.5 shows the reduced 2D feature space for bearing 1 of data-set one. Now we implement K means clustering.

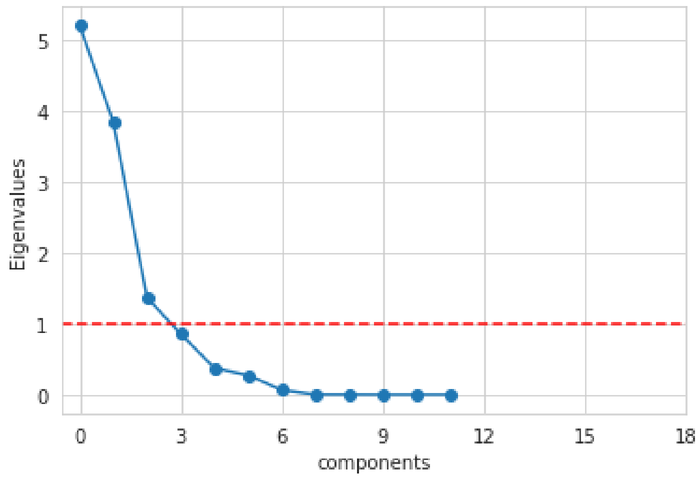


Figure 4.3: Eigen decomposition on the bearing dataset

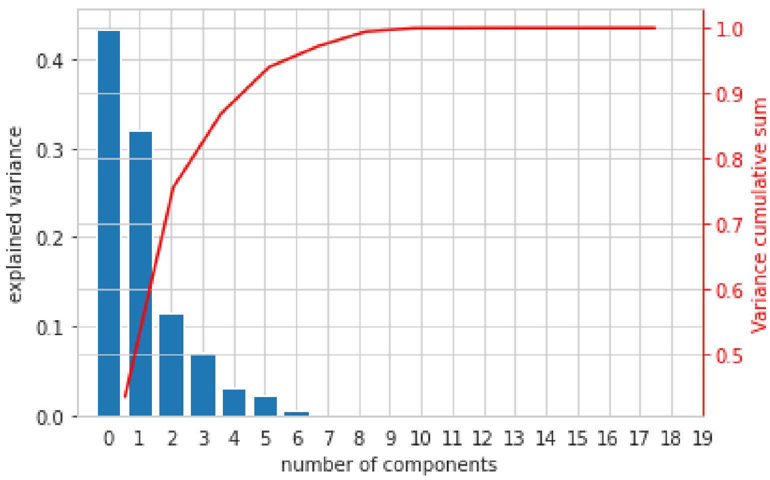


Figure 4.4: Explained variance after PCA on bearing dataset

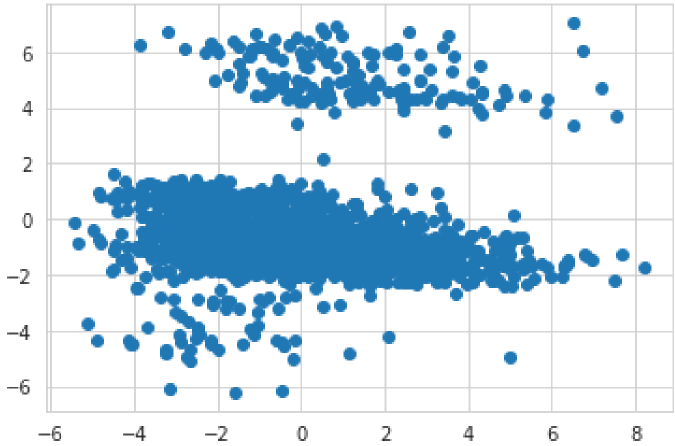


Figure 4.5: Reduced 2-D feature space from PCA showing separation

K means clustering is one of the most common clustering algorithm. The workflow of the algorithm is as follows:

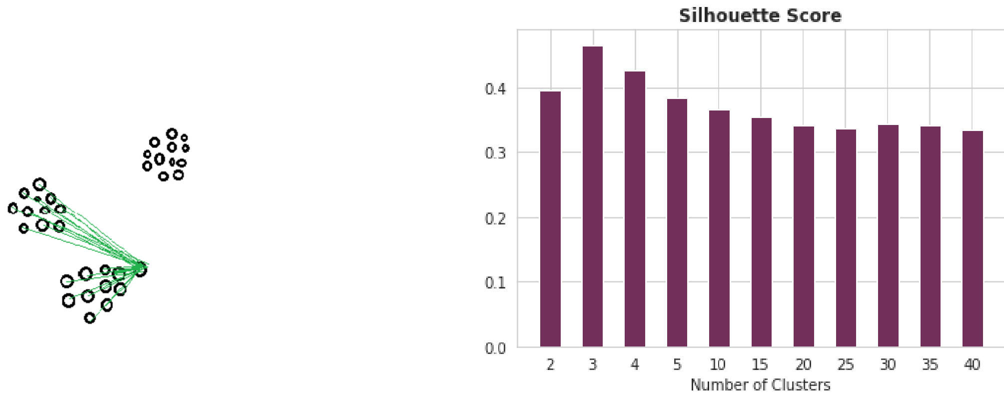
1. Initialisation of the loop by selecting suitable number of clusters.
2. Given an initial clustering of the data, we relocate each point to its new nearest center.
3. Update the clustering centers by calculating the mean of the member points.
4. The relocating and updating procedure is repeated until a convergence criterion is met.

The K-means method works well for clustering but that is only if we know the number of clusters present in the data. Since our data-set is unlabelled, we do not know the right number of clusters. To choose the number of clusters for K means we have 2 methods:

1. Elbow Method
2. Silhouette Method

In this thesis, we explore the clusters using the Silhouette Method.

Silhouette Method Silhouette is one of the most famous methods which allows us to select the suitable number of clusters increasing our accuracy. We also perform the hyper-parameter tuning to choose the best value of k. The Silhouette method picks up the range of K values and draws the silhouette graph as shown in fig 4.6a. It calculates the silhouette coefficient of every point. Then we calculate the average distance of points within its cluster and the average distance of the points to its next closest cluster. Optimising for the Silhouette coefficient, we can find the optimum number of clusters which is shown in fig 4.6b.



(a) Silhouette Method (b) Denoised Data

Figure 4.6: Silhouette Coefficient

We allot the optimum number of clusters to k and initiate the clustering. Fig 4.7 shows us the clustered data with 3 clusters along with the cluster centroids.

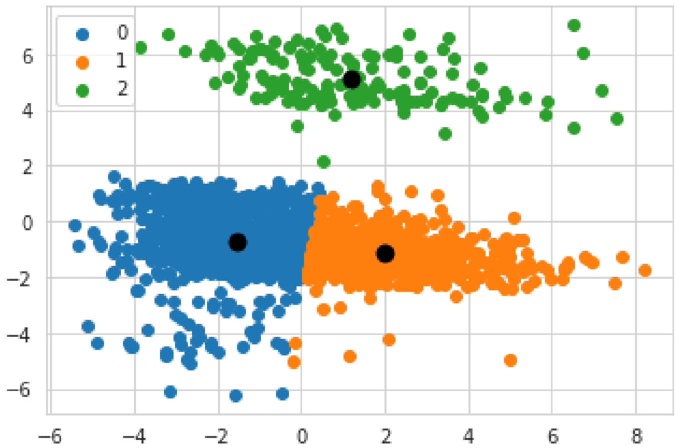


Figure 4.7: Clustered data after K-means clustering

Fig 4.7 shows the clustering done for all the different bearings of the different data-sets using K-means clustering. We are able to observe good accuracy when correlated with the fault occurrence observed in the experimental setup. The code snippet for all the above work can be found at Appendices A.6 & A.7.

4.4 Conclusion

From this chapter, we are able to conclude that it is possible to identify defects and monitor health for components which have multiple defects occurring simultaneously. We also realise that we need to try to get a labelled data-set to validate our results. This allows us to implement better algorithms which has been trained on other data through transfer learning. We also realised that we are not able to visualise the clustering in 3D or higher feature space due to lack of compute resources which can be further investigated.

CHAPTER 5

SUMMARY AND CONCLUSION

The use of state-of-the-art Machine Learning techniques to identify the presence of faults in rotating elements is investigated. Statistical analysis of vibration-based sensor data is performed using suitable algorithms on two datasets, with one containing a single fault and the other containing multiple faults. The first dataset, obtained from an accelerated fatigue-test experiment conducted at IIT Madras [17], pertains to transverse fatigue cracks in shafts while the second contains vibration-based sensor data for multiple bearing defects such as inner race, outer race, and ball defects from the University of Cincinnati[14]. The use of dimensionality reduction, coupled with clustering for classifying multiple defects, was investigated to arrive at damage indices capable of identifying the onset of defects. Furthermore, results from this analysis were further leveraged to formulate a fused-health indicator to reduce the overheads associated with computing for enabling sensor-level *in-situ* analytics.

Accelerometer data of the shaft, after suitable feature-extraction and dimensionality reduction, is decomposed into three fused features capable of identifying the onset of crack. While feature extraction resulted in an increase in the size of the dataset to 18-fold, dimensionality reduction algorithms generated fused features that reflected greater sensitivity and variance towards the identification of the crack. A first-principles-based, rudimentary Principal Component Analysis (PCA) was performed on the high-dimensional data to obtain these fused features. The results of this basic analysis were validated with both the computational and experimental results of the work outlined in [17]. Subsequently, the use of non-linear dimensionality reduction techniques was explored due to the inherent non-linearities associated with vibration-based sensor data. Four more techniques including Kernel Principal Component Analysis(KPCA), Independent Component Analysis(ICA), Sparse Principal Component Analysis(SPCA), and Incremental Principal Component Analysis (IPCA) were tested for robustness, accuracy, and speed. While KPCA showed no significant variations in Hotelling's T^2-statistic, the Q-index-based time of onset of crack detection was accurately reproduced and retained. While ICA is not traditionally used for dimensionality reduction (rather used for separation of constituent signals), it showcased greater sensitivity in T^2 and accurately identify the onset of crack with the Q-index. IPCA, aimed as a replacement for PCA in terms of better memory usage, displayed close agreement with the linear PCA results while being faster. Specifically in such applications, where available onboard compute and memory can be low, IPCA serves as an ideal alternative enabling on-the-fly analytics at the sensor level. The differences in the observed time of onset of cracks for the various algorithms are noted.

The performance of these algorithms is also tested by classifying multiple defects in the vibration data using the bearing dataset. Bearings, being fundamentally more complicated in terms of design compared to a shaft, are hence much more prone to develop faults in either of their constituents, giving rise to a number of combinations of faults spanning the inner race, outer race, and the ball. PCA was applied to a high-dimensional feature matrix spanning a total of 12 features each of which is sensitive to the onset of either of the three defects. This is further decomposed to a low dimensional space consisting of 3 fused features without separation between the multiple faults. To achieve classification $K-$means clustering algorithm with a Silhouette method is used. This method accurately identified the 3 faults present in the dataset with sufficient separation between the clusters.

Although the work presented in the thesis explores only the tip of the iceberg in the vast domain of statistical fault detection and feature engineering, it gave the authors a birds-eye view of the methods behind their work and implementation. This motivates us strongly to pursue this work further in the future. More advanced transforms on the parent dataset such as continuous wavelet transform unlock doors to the extraction of more sensitive features such as spectral kurtosis and entropy, that can possibly influence the fused indicators towards greater sensitivity. Such features have been omitted due to a paucity associated with time. Furthermore, the concept of fused health indicators for multiple faults has not taken shape yet, leaving plenty of room for the authors to carry out more detailed investigations. In addition, compute times associated with analytics on such large datasets are also identified to be a bottleneck. A scalable analysis thus becomes critical when one requires on-the-fly *in-situ* fault detection. These are some areas the authors wish to further explore beyond the aims and objectives of this thesis. However, for the sake of brevity, we close our investigations with these learning-packed explorations as a part of our Final Year Project.

APPENDIX A

PYTHON CODES

We present to you the codes we wrote to obtain the results in this work. All of these were run in Google Collaboratory environment and are produced as-is cell-by-cell for reference.

A.1 Linear Principal Component Analysis

```
#—— Part 1: Import Libraries and understand the dataset ———#

# Import libraries
import scipy.io as scpy
import scipy as sp
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import random
from sklearn.decomposition import PCA
from sklearn import preprocessing

# Get data from remote repository
!git clone https://gitlab.com/kailashjagadeesh/temp_files.git

# Visualise Dataset for understanding
time = scpy.loadmat('/content/temp_files/Accmidtime.mat')
time = time['Data1_time_Acc_Mid']
acc = scpy.loadmat('/content/temp_files/Accmid.mat')
acc = acc['Data1_Acc_Mid']
plt.plot(time,acc)
plt.xlabel("time[s]")
plt.ylabel("acceleration[m/s2]")

# A generic utility function to visualize the feature matrix
def visualize_feature(Y):
  fig, axs = plt.subplots(6, 3,figsize=(12,15))
  names= ['Mean','RMS','Crest Factor','Peak-to-Peak','Energy','Skewness','Kurtosis','Shape Factor','Impulse Factor','Margin Factor','Kurtosis Factor','Hybrid Kurtosis','1X','2X','3X','BP1','BP2','BP3']
  for i in range(6):
    for j in range(3):
      r = random.random()
      b = random.random()
      g = random.random()
      color = (r, g, b)
      axs[i,j].plot(Y[:,3*i+j],color=color)
      axs[i,j].set_title(names[3*i+j])
  plt.subplots_adjust(left=0.1,
                      bottom=0.1,
```

```
                        right=0.9,
                        top=1.5,
                        wspace=0.25,
                        hspace=0.25)

#--- Part 2: Pre-process and clean the obtained data ---#

# Resizing data into 390 subdata of 2 sec equal time intervals
n_steps_orig = len(acc)
total_time = 780 #sec
per_step = total_time/n_steps_orig
freq = 1/per_step/1000
cols = 2/per_step
rows = n_steps_orig*per_step/2
print("deltaT (per_step) = ", per_step)
print("frequency of data collection = ", freq," kHz")
print("Desidered segment length = 2s")
print("Size of sub-data-set = ",cols)
print("Numer of time-steps = ", rows)

# Resizing acceleration data into new array S
acc.resize((int(rows),int(cols)))
S = acc #assign to a new array just to be consistent
print("Subdatasets = " , S)

#Standardize Data and Denoise using PCA
scaler = preprocessing.StandardScaler().fit(S)
S_scaled_train = scaler.transform(S)
pca = PCA(0.98).fit(S_scaled_train) #tune for 98% variance
components = np.linspace(0,len(pca.explained_variance_ratio_)-1,len(pca.explained_variance_ratio_))

#Plotting PCA graphs
ax = plt.gca()
ax2 = ax.twinx()
ax2.plot(np.cumsum(pca.explained_variance_ratio_),color="r")
ax.bar(components, pca.explained_variance_ratio_)
ax.set_yscale('log')
ax.set_xlim([-1,71])
ax.set_ylim([1e-4,1])
ax.set_xlabel('number of components')
ax.set_ylabel('explained variance')
ax2.set_ylabel('Variance cumulative sum')
ax2.spines['right'].set_color('red')
ax2.tick_params(colors='red')
ax2.yaxis.label.set_color('red')

# Obtain filtered data
components = pca.transform(S_scaled_train)
filtered = pca.inverse_transform(components)
filtered = scaler.inverse_transform(filtered)
filtered.resize(int(rows*cols),1)
acc.resize(int(rows*cols),1)

# Visualising the noisy and filtered data
plt.plot(time, acc, color='black',label='noisy')
```

```
plt.plot(time, filtered ,color='orange',label='filtered')
plt.xlabel("time[s]")
plt.ylabel("acceleration[m/s2]")
plt.legend()

# Check if SNR has improved
acc.resize(1,int(rows*cols))
filtered.resize(1,int(rows*cols))
print("SNR of acc data",-np.log10(np.abs(np.mean(acc[0,:])/np.std(acc[0,:])
    )))
print("SNR o filtered data",-np.log10(np.abs(np.mean(filtered[0,:])/np.std(
    filtered[0,:]))))

#-- Part 3: Fourier transform to frequency space ---#

# Obtaining Frequency Domain Data
from numpy.fft import fft
filtered.resize(int(rows),int(cols))
frequency_data = np.zeros((int(rows),int(cols)))
freq = np.zeros((int(rows),int(cols)))
for i in range(int(rows)):
  signal = filtered[i,:]
  freq_signal = fft(signal)
  freq_signal = 10*np.log10(np.abs(freq_signal/16.5))
  frequency_data[i,:] = freq_signal
  N = len(signal)
  n = np.arange(N)
  sr = 20000
  T = N/sr
  freq[i,:] = n/T

# Plotting Frequency Domain Data
plt.plot(freq[0,:],frequency_data[0,:], color = 'blue') #<--toggle the row
    number to get different freq data at different times (yellow too feeble)
plt.axvline(x=16.5,color='red')
plt.text(16.6,-60,'16.5Hz',rotation=0,color='red')
plt.axvline(x=33,color='red')
plt.text(33.6,-60,'33.0Hz',rotation=0,color='red')
plt.axvline(x=49.5,color='red')
plt.text(50.0,-60,'49.5Hz',rotation=0,color='red')
plt.axvline(x=66.0,color='red')
plt.text(66.6,-60,'66.0Hz',rotation=0,color='red')
ax = plt.gca()
ax.set_xlim([0,100])
ax.set_xlabel("frequency(Hz)")
ax.set_ylabel("Mag(dB)")

#---- Part4: Feature Extraction -----#

# Generating feature matrix

S = filtered
n_features = 18
Y = np.zeros((int(rows),int(n_features)))   # Y is the feature matrix

# TIME DOMAIN FEATURES
```

```
#mean
Y[:,0] = np.mean(S,axis=1)

#RMS
Y[:,1] = np.sqrt(np.mean(S**2,axis=1))

#Crest factor
Y[:,2] = np.max(np.abs(S),axis=1)/np.sqrt(np.mean(S**2,axis=1))

#peak to peak
Y[:,3] = np.max(S,axis=1)-np.min(S,axis=1)

#Energy
Y[:,4] = np.sum(S**2,axis=1)

#skewness
from scipy.stats import skew
Y[:,5] = skew(S,axis=1,bias=False)

#kurtosis
from scipy.stats import kurtosis
kurt = kurtosis(S,axis=1,fisher=False)
Y[:,6] = kurt

#shape factor
RMS = np.sqrt(np.mean(S**2,axis=1))
mu_mod = np.mean(np.abs(S),axis=1)
Y[:,7] = np.divide(RMS, mu_mod)

#Impulse factor
Y[:,8] = np.divide((np.max(np.abs(S),axis=1)), mu_mod)

#Margin factor
Y[:,9] = np.divide(np.max(np.abs(S),axis=1), np.power(mu_mod,2))

#kurtosis factor
Y[:,10] = np.divide(kurt, np.power(RMS, 4))

#hybrid kurtosis
variance_2 = np.power(np.var(S,axis=1),2)
HK = np.sqrt(np.multiply(kurt, variance_2))/cols
Y[:, 11] = HK


# FREQUENCY DOMAIN FEATURES

#1st Harmonic
for i in range(int(rows)):
  Y[i,12] = frequency_data[i,list(freq[i]).index(16.5)]

#2nd Harmonic
for i in range(int(rows)):
  Y[i,13] = frequency_data[i,list(freq[i]).index(33)]

#3rd Harmonic
```

```
for i in range(int(rows)):
  Y[i,14] = frequency_data[i,list(freq[i]).index(49.5)]

# Need to cross-check Band Power, SK,SE,SS Formula ?!!

#Band Power 1
for i in range(int(rows)):
  #psd = frequency_data[i,:] * np.conj(frequency_data[i,:])/cols
  #Y[i,15] = psd[list(freq[i]).index(16.5)]
  bpf_data_1 = frequency_data[i,list(freq[i]).index(14.5):list(freq[i]).
      index(19)]
  band_power_1 = bpf_data_1*np.conj(bpf_data_1)
  Y[i,15] = np.sqrt(np.mean(band_power_1**2))

#Band Power 2
for i in range(int(rows)):
  #psd = frequency_data[i,:] * np.conj(frequency_data[i,:])/cols
  #Y[i,16] = psd[list(freq[i]).index(33)]
  bpf_data_2 = frequency_data[i,list(freq[i]).index(31):list(freq[i]).index
      (35.5)]
  band_power_2 = bpf_data_2*np.conj(bpf_data_2)
  Y[i,16] = np.sqrt(np.mean(band_power_2**2))

#Band Power 3
for i in range(int(rows)):
  #psd = frequency_data[i,:] * np.conj(frequency_data[i,:])/cols
  #Y[i,17] = psd[list(freq[i]).index(49.5)]
  bpf_data_3 = frequency_data[i,list(freq[i]).index(47.5):list(freq[i]).
      index(52)]
  band_power_3 = bpf_data_3*np.conj(bpf_data_3)
  Y[i,17] = np.sqrt(np.mean(band_power_3**2))

print(Y, Y.shape)

# Viewing Array in Data Frame
dataframe_Y = pd.DataFrame(Y, columns=['Mean', 'RMS', 'Crest Factor', 'Peak
    to Peak', 'Energy', 'Skewness', 'Kurtosis', 'Shape Factor',
                                        'Impulse Factor', 'Margin Factor', '
                                           Kurtosis Factor', 'Hybrid 
                                           Kurtosis', '1st Harmonic', '2nd 
                                           Harmonic',
                                        '3rd Harmonic', 'Band Power I', '
                                           Band Power II', 'Band Power III'
                                           ])
dataframe_Y

#Visualize Feature Matrix
visualize_feature(Y)

#--- Part 5: Perform dimensionality reduction using PCA ---#

# Splitting Train and Test Data
Y_train = Y[0:155,:] # Train Data
Y_test = Y[155:-1,:]

#Visualise Train Data
```

```
visualize_feature(Y_train)

# PRINCIPAL COMPONENT ANALYSIS

#import stuff
from sklearn.decomposition import PCA
from sklearn import preprocessing

#preprocess by standardizing
scaler = preprocessing.StandardScaler().fit(Y_train)
Y_scaled_train = scaler.transform(Y_train)
Y_scaled_test = scaler.transform(Y_test)

#perform PCA
pca = PCA(n_components=3)
pca.fit(Y_scaled_train)

#visualize variance explained
components = np.linspace(0,len(pca.explained_variance_ratio_)-1,len(pca.
    explained_variance_ratio_))
ax = plt.gca()
ax2 = ax.twinx()
ax2.plot(components, np.cumsum(pca.explained_variance_ratio_),color="r")
ax.bar(components, pca.explained_variance_ratio_)
ax.set_xlabel('number of components')
ax.xaxis.set_ticks(np.arange(0, 20, 1))
ax.set_ylabel('explained variance')
ax2.set_ylabel('Variance cumulative sum')
ax2.spines['right'].set_color('red')
ax2.tick_params(colors='red')
ax2.yaxis.label.set_color('red')

#Obtaining Eigen Values
plt.plot(pca.explained_variance_,"-o")
ax = plt.gca()
plt.axhline(y=1,linestyle='--',color='red')
ax.set_xlabel('components')
ax.xaxis.set_ticks(np.arange(0, 20, 3))
ax.set_ylabel('Eigenvalues')

#--- Part 6: Extract damage indices ---#

# Create arrays for damage indices
T_sr= pca.transform(Y_scaled_train)
T_sr_test=pca.transform(Y_scaled_test)
X_rt= pca.inverse_transform(T_sr)
X_rd= Y_scaled_train-X_rt
X_rt_test= pca.inverse_transform(T_sr_test)
X_rd_test= Y_scaled_test-X_rt_test

# Hotelling's T2-statistic
T2_train= np.sum(T_sr**2,axis=1) # training data T2 index
T2_test=np.sum(T_sr_test**2,axis=1) # testing data T2 index
T_2_net = np.concatenate((np.array(T2_train),np.array(T2_test))) #
    concatenate for NOR computation purposes
x1 = 2*np.arange(len(T2_train))
```

```
plt.plot(x1,T2_train)
len2 = len(T2_test)
x2 = 2*np.arange(len2)+2+2*155
x_net = np.concatenate((np.array(x1),np.array(x2))) #concatenate for NOR
    computation purposes
plt.plot(x2, T2_test,'r')
ax = plt.gca()
ax.set_yscale('log')
x_ticks = [0, 130, 260, 390, 520, 650, 780]
plt.xticks(x_ticks, x_ticks)
plt.xlim([0,780])

#Q-index
Qi_train= np.sum(X_rd**2,axis=1) # training data T2 index
Qi_test=np.sum(X_rd_test**2,axis=1) # training data T2 index
Qi_net = np.concatenate((np.array(Qi_train),np.array(Qi_test)))
x1 = 2*np.arange(len(Qi_train))
plt.plot(x1,Qi_train)
len2 = len(Qi_test)
x2 = 2*np.arange(len2)+2+2*155
plt.plot(x2, Qi_test,'r')
ax = plt.gca()
ax.set_yscale('log')

#Histogram to compute NOR
import scipy.stats as stat
figure, axis = plt.subplots(1, 2, figsize=(10,15))

# For T2 statisitic
axis[0].hist(T_2_net,bins=100)
axis[0].set_title("Distribution of $T^2$ statistic")
axis[0].set_xlabel("$T^2$-statistic")
axis[0].set_ylabel("Frequency")
plt.sca(axis[0])
x_ticks = [500]
y_ticks = [0,50,100,200,300,320, 340, 350]
plt.xticks(x_ticks, x_ticks)
plt.yticks(y_ticks, y_ticks)

# For Q-index
axis[1].hist(Qi_net, bins=100)
axis[1].set_title("Distribution of $Q-$index")
axis[1].set_xlabel("$Q$-index")
axis[1].set_ylabel("Frequency");
#axis[1].set_xscale("log")
plt.sca(axis[1])
x_ticks = [25, 1000]
y_ticks = [0,50,100,200,300,320, 340, 350]
plt.xticks(x_ticks, x_ticks)
plt.yticks(y_ticks, y_ticks)

# redo the plots with the NOR
figure, axis = plt.subplots(1, 2, figsize=(15,5))

#T2
axis[0].plot(x1,T2_train,label="Training data")
```

```
axis[0].plot(x2, T2_test,'r',label="Testing data")
axis[0].set_yscale('log')
plt.sca(axis[0])
x_ticks = [0, 130, 260, 390, 520, 650, 740, 780]
plt.xticks(x_ticks, x_ticks)
plt.xlim([0,780])
plt.title("Hotelling's $T^2$-statistic")
plt.xlabel("Time[s]")
plt.fill_between(x_net,
        y1=500, #play with this to change the NOR region
        color= "lime",
        alpha=0.3)
plt.axhline(y=500, color='k', linestyle='--',label="$T^2$ threshold")
plt.axvline(x=740, color='b', linestyle='--',label="Onset of crack") #
    change here too
plt.legend(loc="upper left");

#Q-index
axis[1].plot(x1,Qi_train,label="Training data")
axis[1].plot(x2, Qi_test,'r',label="Testing data")
axis[1].set_yscale('log')
plt.sca(axis[1])
x_ticks = [0, 130, 260, 390, 520, 650, 780]
plt.xticks(x_ticks, x_ticks)
plt.xlim([0,780])
plt.title("$Q$-index")
plt.xlabel("Time[s]")
plt.fill_between(x_net,
        y1=25, #play with this to change the NOR region
        color= "lime",
        alpha=0.3)
plt.axhline(y=25, color='k', linestyle='--',label="Q-index threshold") #
    change here too
plt.axvline(x=520, color='b', linestyle='--',label="Onset of crack") #
    change here too
plt.legend(loc="upper left");

#--- Part 7: Partial Decomposition Contribution ---#

pca = PCA(n_components=3)
pca.fit(Y_scaled_test)
P_n_n = pca.components_
L_half = np.diag(pca.explained_variance_**-0.5)
scalingfactor = np.dot(np.array((-1.*pca.components_.T)),L_half);
zeta= np.eye(len(scalingfactor))
nf=18;  # number of features
npcs = 3;
pdc=np.zeros((120,18));
for j in range(0,110):  #for fault data 261:300
    for i in range(0,nf):
      G = (zeta)-np.dot(scalingfactor[:,1:npcs],scalingfactor[:,1:npcs].T);
      I = zeta[:,i]*zeta[:,i].T;
      first_dot = np.dot(Y_scaled_test[j,:], G)
      second_dot = np.dot(first_dot, I)
      third_dot = np.dot(second_dot, Y_scaled_test[j,:].T)
      pdc[j-110,] = third_dot
```

```

pdcacclt1=pdc[0:100,:];
pdcacclt2=np.mean(pdcacclt1, axis=0)
names= ['Mean','RMS','Crest Factor','Peak-to-Peak','Energy','Skewness','Kurtosis','Shape Factor','Impulse Factor','Margin Factor','Kurtosis Factor','Hybrid Kurtosis','1X','2X','3X','BP1','BP2','BP3']
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
len(names)
dict_pdc = {names[i]: abs(pdcacclt2[i]) for i in range(len(names))}
ax.bar(names,abs(pdcacclt2))
sorted_x = sorted(dict_pdc.items(), key=lambda kv: kv[1])
import collections
sorted_dict = collections.OrderedDict(sorted_x)
plt.xticks(rotation=90)
key_ordered = list(sorted_dict.keys())
print("Features for FHI are ", key_ordered[-3:])
plt.show()

# compute the Fused Health Indicator
scaler = preprocessing.StandardScaler().fit(Y)
Y_scaled = scaler.transform(Y)
a1  = Y_scaled[:,6]
a2 = Y_scaled[:,12]
a3 = Y_scaled[:,15]
FHI = abs(a1 + a2 + a3 / (np.sqrt(a1**2 + a2**2 +a3**2 )))
FHI = FHI/np.max(FHI)
plt.scatter(np.arange(0,len(FHI)), FHI)
ax = plt.gca()

#--- The End -----#
```

A.2 Kernel PCA

```
#--- Part 1: Import Libraries ---#

import scipy.io as scpy
import scipy as sp
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import random
from numpy.fft import fft
from sklearn import datasets
from sklearn import preprocessing
from sklearn.decomposition import PCA, KernelPCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
colors = ['royalblue','red','deeppink', 'maroon', 'mediumorchid', 'tan', 'forestgreen', 'olive', 'goldenrod', 'lightcyan', 'navy']

#---- Part 2: Extraction and Pre-processing of data ---#
#clone parent dataset
!git clone https://gitlab.com/kailashjagadeesh/temp_files.git
```

```
#extract
time = scpy.loadmat('/content/temp_files/Accmidtime.mat')
time = time['Data1_time_Acc_Mid']
acc = scpy.loadmat('/content/temp_files/Accmid.mat')
acc = acc['Data1_Acc_Mid']

#resizing calcs
n_steps_orig = len(acc)
total_time = 780 #sec
per_step = total_time/n_steps_orig
freq = 1/per_step/1000
cols = 2/per_step
rows = n_steps_orig*per_step/2

#resize
acc.resize((int(rows),int(cols)))
S = acc

#Standardize Data
scaler = preprocessing.StandardScaler().fit(S)
S_scaled_train = scaler.transform(S)
pca = PCA(0.98).fit(S_scaled_train) #tune for 98% variance
components = np.linspace(0,len(pca.explained_variance_ratio_)-1,len(pca.
    explained_variance_ratio_))

#Denoise
components = pca.transform(S_scaled_train)
filtered = pca.inverse_transform(components)
filtered = scaler.inverse_transform(filtered)
filtered.resize(int(rows*cols),1)
acc.resize(int(rows*cols),1)

#FFT for freq domain
filtered.resize(int(rows),int(cols))
frequency_data = np.zeros((int(rows),int(cols)))
freq = np.zeros((int(rows),int(cols)))
for i in range(int(rows)):
   signal = filtered[i,:]
   freq_signal = fft(signal)
   freq_signal = 10*np.log10(np.abs(freq_signal/16.5))
   frequency_data[i,:] = freq_signal
  N = len(signal)
  n = np.arange(N)
   sr = 20000
  T = N/sr
   freq[i,:] = n/T

#Extract features
S = filtered
n_features = 18
Y = np.zeros((int(rows),int(n_features)))    # Y is the feature matrix

# TIME DOMAIN FEATURES

#mean
```

```
Y[:,0] = np.mean(S,axis=1)

#RMS
Y[:,1] = np.sqrt(np.mean(S**2,axis=1))

#Crest factor
Y[:,2] = np.max(np.abs(S),axis=1)/np.sqrt(np.mean(S**2,axis=1))

#peak to peak
Y[:,3] = np.max(S,axis=1)-np.min(S,axis=1)

#Energy
Y[:,4] = np.sum(S**2,axis=1)

#skewness
from scipy.stats import skew
Y[:,5] = skew(S,axis=1,bias=False)

#kurtosis
from scipy.stats import kurtosis
kurt = kurtosis(S,axis=1,fisher=False)
Y[:,6] = kurt

#shape factor
RMS = np.sqrt(np.mean(S**2,axis=1))
mu_mod = np.mean(np.abs(S),axis=1)
Y[:,7] = np.divide(RMS, mu_mod)

#Impulse factor
Y[:,8] = np.divide((np.max(np.abs(S),axis=1)), mu_mod)

#Margin factor
Y[:,9] = np.divide(np.max(np.abs(S),axis=1), np.power(mu_mod,2))

#kurtosis factor
Y[:,10] = np.divide(kurt, np.power(RMS, 4))

#hybrid kurtosis
variance_2 = np.power(np.var(S,axis=1),2)
HK = np.sqrt(np.multiply(kurt, variance_2))/cols
Y[:, 11] = HK


# FREQUENCY DOMAIN FEATURES

#1st Harmonic
for i in range(int(rows)):
  Y[i,12] = frequency_data[i,list(freq[i]).index(16.5)]

#2nd Harmonic
for i in range(int(rows)):
  Y[i,13] = frequency_data[i,list(freq[i]).index(33)]

#3rd Harmonic
for i in range(int(rows)):
  Y[i,14] = frequency_data[i,list(freq[i]).index(49.5)]
```

```
# Need to cross-check Band Power, SK,SE,SS Formula ?!!

#Band Power 1
for i in range(int(rows)):
  #psd = frequency_data[i,:] * np.conj(frequency_data[i,:])/cols
  #Y[i,15] = psd[list(freq[i]).index(16.5)]
   bpf_data_1 = frequency_data[i,list(freq[i]).index(14.5):list(freq[i]).
       index(19)]
   band_power_1 = bpf_data_1*np.conj(bpf_data_1)
  Y[i,15] = np.sqrt(np.mean(band_power_1**2))

#Band Power 2
for i in range(int(rows)):
  #psd = frequency_data[i,:] * np.conj(frequency_data[i,:])/cols
  #Y[i,16] = psd[list(freq[i]).index(33)]
   bpf_data_2 = frequency_data[i,list(freq[i]).index(31):list(freq[i]).index
       (35.5)]
   band_power_2 = bpf_data_2*np.conj(bpf_data_2)
  Y[i,16] = np.sqrt(np.mean(band_power_2**2))

#Band Power 3
for i in range(int(rows)):
  #psd = frequency_data[i,:] * np.conj(frequency_data[i,:])/cols
  #Y[i,17] = psd[list(freq[i]).index(49.5)]
   bpf_data_3 = frequency_data[i,list(freq[i]).index(47.5):list(freq[i]).
       index(52)]
   band_power_3 = bpf_data_3*np.conj(bpf_data_3)
  Y[i,17] = np.sqrt(np.mean(band_power_3**2))

print(Y, Y.shape)

#Splitting Train and Test Data
Y_train = Y[0:155,:] # Train Data
Y_test = Y[155:-1,:] # Test Data

#Standardize
scaler = preprocessing.StandardScaler().fit(Y_train)
Y_scaled_train = scaler.transform(Y_train)
Y_scaled_test = scaler.transform(Y_test)

#----- Part 3: Perform KPCA -----#

kernel_pca = KernelPCA(n_components=9,
                       kernel="cosine",
                      gamma=10,
                       fit_inverse_transform=True,
                       alpha=0.1)
kernel_pca.fit(Y_scaled_train)
plt.plot(kernel_pca.eigenvalues_,'o-')
ax = plt.gca()
plt.axhline(y=1,linestyle='--',color='red')
ax.set_xlabel('components')
ax.set_ylabel('Eigenvalues')
ax.xaxis.set_ticks(np.arange(0, 20, 3))

```

```
#visualize eigen values
T_sr_train = kernel_pca.transform(Y_scaled_train)
T_sr_test = kernel_pca.transform(Y_scaled_test)
X_rt_train = kernel_pca.inverse_transform(T_sr_train)
X_rt_test = kernel_pca.inverse_transform(T_sr_test)
X_rd_train = Y_scaled_train-X_rt_train
X_rd_test = Y_scaled_test-X_rt_test

#--- Part 4: Compute damage indices ---#

# Hotelling's T2 statistic
T2_train= np.sum(T_sr_train**2,axis=1) # training data T2 index
T2_test=np.sum(T_sr_test**2,axis=1) # testing data T2 index
T_2 = np.concatenate((np.array(T2_train),np.array(T2_test))) #concatenate
    for NOR computation purposes
x1 = 2*np.arange(len(T2_train))
plt.plot(x1,np.reciprocal(T2_train))
len2 = len(T2_test)
x2 = 2*np.arange(len2)+2+2*155
x_net = np.concatenate((np.array(x1),np.array(x2))) #concatenate for NOR
    computation purposes
plt.plot(x2, np.reciprocal(T2_test),'r')
ax = plt.gca()
x_ticks = [0, 130, 260, 390, 520, 650, 780]
plt.xticks(x_ticks, x_ticks)
plt.xlim([0,780])

#Q-index
Qi_train= np.sum(X_rd_train**2,axis=1) # training data T2 index
Qi_test=np.sum(X_rd_test**2,axis=1) # training data T2 index
Qi_net = np.concatenate((np.array(Qi_train),np.array(Qi_test)))
x1 = 2*np.arange(len(Qi_train))
plt.plot(x1,Qi_train)
len2 = len(Qi_test)
x2 = 2*np.arange(len2)+2+2*155
plt.plot(x2, Qi_test,'r')
ax = plt.gca()
ax.set_yscale('log')
x_ticks = [0, 130, 260, 390, 520, 650, 780]
plt.xticks(x_ticks, x_ticks)
plt.xlim([0,780]);

#compute NOR using histogram
figure, axis = plt.subplots(1, 2,figsize=(20,10))

# For T2 statisitic
axis[0].hist(np.reciprocal(T_2))
axis[0].set_title("Distribution of $\\frac{1}{T^2}$ statistic")
axis[0].set_xlabel("$\\frac{1}{T^2}$-statistic")
axis[0].set_ylabel("Frequency")

# For Q-index
axis[1].hist(Qi_net, bins=200)
axis[1].set_title("Distribution of $Q-$index")
axis[1].set_xlabel("$Q$-index")
axis[1].set_ylabel("Frequency")
```

```
plt.sca(axis[1])
x_ticks = [80]
y_ticks = [0,50,100,200,300,320, 340, 350]
plt.xticks(x_ticks, x_ticks)
plt.yticks(y_ticks, y_ticks)

# redo the plots with the NOR
figure, axis = plt.subplots(1, 2, figsize=(15,5))

#T2
axis[0].plot(x1,T2_train,label="Training data")
axis[0].plot(x2, T2_test,'r',label="Testing data")
axis[0].set_yscale('log')
plt.sca(axis[0])
x_ticks = [0, 130, 260, 390, 520, 650, 780]
plt.xticks(x_ticks, x_ticks)
plt.xlim([0,780])
plt.title("Hotelling's $T^2$-statistic")
plt.xlabel("Time[s]")
plt.fill_between(x_net,
        y1=2500, #play with this to change the NOR region
        color= "lime",
        alpha=0.3)
plt.axhline(y=2500, color='k', linestyle='--',label="NOR")
plt.legend(loc="upper left");

#Q-index
axis[1].plot(x1,Qi_train,label="Training data")
axis[1].plot(x2, Qi_test,'r',label="Testing data")
axis[1].set_yscale('log')
plt.sca(axis[1])
x_ticks = [0, 130, 260, 390, 490, 535, 650, 780]
plt.xticks(x_ticks, x_ticks)
plt.xlim([0,780])
plt.title("$Q$-index")
plt.xlabel("Time[s]")
plt.fill_between(x_net,
        y1=80, #play with this to change the NOR region
        color= "lime",
        alpha=0.3)
plt.axhline(y=80, color='k', linestyle='--',label="NOR") #change here too
plt.axvline(x=535, color='b', linestyle='--',label="Onset of crack") #
    change here too
plt.legend(loc="upper left");

#--- Part 5: Partial Decomposition Contribution ---#
pca = PCA(n_components=3)
pca.fit(Y_scaled_test)
P_n_n = pca.components_
L_half = np.diag(pca.explained_variance_**-0.5)
scalingfactor = np.dot(np.array((-1.*pca.components_.T)),L_half);
zeta= np.eye(len(scalingfactor))
nf=18; # number of features
npcs = 3;
pdc=np.zeros((120,18));
for j in range(0,110): #for fault data 261:300
```

```
    for i in range(0,nf):
      G = (zeta)-np.dot(scalingfactor[:,1:npcs],scalingfactor[:,1:npcs].T);
      I = zeta[:,i]*zeta[:,i].T;
      first_dot = np.dot(Y_scaled_test[j,:], G)
      second_dot = np.dot(first_dot, I)
      third_dot = np.dot(second_dot, Y_scaled_test[j,:].T)
      pdc[j-110,] = third_dot

pdcacclt1=pdc[0:100,:];
pdcacclt2=np.mean(pdcacclt1,axis=0)
print(pdcacclt2);
names= ['Mean','RMS','Crest Factor','Peak-to-Peak','Energy','Skewness','Kurtosis','Shape Factor','Impulse Factor','Margin Factor','Kurtosis Factor','Hybrid Kurtosis','1X','2X','3X','BP1','BP2','BP3']
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
len(names)
ax.bar(names,abs(pdcacclt2))
plt.xticks(rotation=90)
plt.show()
```

Note: For the rest of the methods, only *Part 3* was changed to observe a change in the results and hence only this is given henceforth. Additionally, we specify the import commands required for the new models.

A.3 Independent Component Analysis

```
from sklearn.decomposition import PCA, FastICA
fast_ica = FastICA(max_iter=100000, tol=1e-4, n_components=9) #needs a large number of iterations to converge as we increase n_components
fast_ica.fit(Y_scaled_train)
#plt.plot(fast_ica.mean_,'o-')
#ax = plt.gca()
#plt.axhline(y=1,linestyle='--',color='red')
#ax.set_xlabel('components')
#ax.set_ylabel('Eigenvalues')
#ax.xaxis.set_ticks(np.arange(0, 20, 3))
T_sr_train = fast_ica.transform(Y_scaled_train)
T_sr_test = fast_ica.transform(Y_scaled_test)
X_rt_train = fast_ica.inverse_transform(T_sr_train)
X_rt_test = fast_ica.inverse_transform(T_sr_test)
X_rd_train = Y_scaled_train-X_rt_train
X_rd_test = Y_scaled_test-X_rt_test
```

A.4 Sparse PCA

```
from sklearn.decomposition import PCA, SparsePCA
sparse_pca = SparsePCA(n_components=9)
sparse_pca.fit(Y_scaled_train)

T_sr_train = sparse_pca.transform(Y_scaled_train)
T_sr_test = sparse_pca.transform(Y_scaled_test)
```

```

X_rt_train = np.dot(T_sr_train, sparse_pca.components_) + sparse_pca.mean_
X_rt_test = np.dot(T_sr_test, sparse_pca.components_) + sparse_pca.mean_

X_rd_train = Y_scaled_train-X_rt_train
X_rd_test = Y_scaled_test-X_rt_test
```

A.5 Incremental PCA

```
from sklearn.decomposition import PCA, IncrementalPCA
incremental_pca = IncrementalPCA(n_components=9)
incremental_pca.fit(Y_scaled_train)

#visualize eigen values
T_sr_train = incremental_pca.transform(Y_scaled_train)
T_sr_test = incremental_pca.transform(Y_scaled_test)
X_rt_train = incremental_pca.inverse_transform(T_sr_train)
X_rt_test = incremental_pca.inverse_transform(T_sr_test)
X_rd_train = Y_scaled_train-X_rt_train
X_rd_test = Y_scaled_test-X_rt_test
```

A.6 Feature Extraction for Bearing dataset

```
#start
# -*- coding: utf-8 -*-
"""Copy of nasa-bearing-feature-extraction.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
        yl5Lwt9QTqiLNvqLvU0ed6zNZ8at8YwI
"""

#importing required libraries for for the code
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt # plotting
import numpy as np # linear algebra
import os # accessing directory structure
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import scipy
from scipy.special import entr

# Commented out IPython magic to ensure Python compatibility.
#downloading the data from NASA Archives and
# extracting them into the working directory in Google Colab
!wget 'https://ti.arc.nasa.gov/c/3/'
!7za x index.html -o/content/bearing_data/
!pwd
# %cd bearing_data
!pwd
!unrar x 1st_test.rar
!unrar x 2nd_test.rar
```

```
!unrar x 3rd_test.rar

#assigning the path of the folder w.r.t the file directory in Google Colab
dataset_path_1st = '/content/bearing_data/1st_test'
dataset_path_2nd = '/content/bearing_data/2nd_test'
dataset_path_3rd = '/content/bearing_data/4th_test/txt'

# Test for the first file
dataset = pd.read_csv(dataset_path_3rd+'/2004.03.04.09.27.46', sep='\t')
dataset.columns = ['Bearing 1','Bearing 2','Bearing 3','Bearing 4']
dataset.head()
print(dataset['Bearing 1'].abs().sum())
print((dataset['Bearing 1'].abs()**0.5)**2)
print(dataset['Bearing 1'].abs())

#testing the raw signal
dataset[['Bearing 1']].plot(figsize=(18,6));

"""# Extract Time Features
References:
http://mkalikatzarakis.eu/wp-content/uploads/2018/12/IMS_dset.html
"""

#Extraction of time domain features, total of 12 features

#function to calculate RMS
def calculate_rms(df):
    result = []
    for col in df:
        r = np.sqrt((df[col]**2).sum() / len(df[col]))
        result.append(r)
    return result

#function to calculate shape factor
def calculate_shape_factor(df):
  result=[]
  rms=calculate_rms(df)
  for i in range(len(rms)):
    result.append(rms[i]/(df[i].abs().sum()/len(df[i])))
  return result

# extract peak-to-peak features
def calculate_p2p(df):
    return np.array(df.max().abs() + df.min().abs())

#function to calculate impulse factor
def calculate_impulse_factor(df):
  result=[]
  peak=np.array(df.abs().max())
  for i in range(len(peak)):
    result.append(peak[i]/(df[i].abs().sum()/len(df[i])))
  return result

#function to calculate crest factor
def calculate_crest_factor(df):
  result=[]
```

```python
  peak=np.array(df.abs().max())
  rms=calculate_rms(df)
  for i in range(len(peak)):
    result.append(peak[i]/rms[i])
  return result

#function to calculate clearance factor
def calculate_clearance_factor(df):
  result=[]
  peak=np.array(df.abs().max())
  for i in range(len(peak)):
    result.append(peak[i]/(((df[i].abs()**0.5)**2).sum()/len(df[i])))
  return result

# extract shannon entropy (cut signals to 500 bins)
def calculate_entropy(df):
    entropy = []
    for col in df:
        entropy.append(scipy.stats.entropy(pd.cut(df[col], 500).value_counts()))
    return np.array(entropy)

#function to loop through all the data samples
def time_features(dataset_path, id_set=None):
    time_features = ['mean','std','skew','kurtosis','entropy','rms','max','p2p','shape_factor','impulse_factor','crest_factor','clearance_factor','rms*kurt']

    #different number of columns since dataset 1 has 2 sensors/bearing
    #while datasets 2 and 3 have 1 sensor/bearing
    cols1 = ['B1_a','B1_b','B2_a','B2_b','B3_a','B3_b','B4_a','B4_b']
    cols2 = ['B1','B2','B3','B4']

    # initialize
    if id_set == 1:
        columns = [c+'_'+tf for c in cols1 for tf in time_features]
        data = pd.DataFrame(columns=columns)
    else:
        columns = [c+'_'+tf for c in cols2 for tf in time_features]
        data = pd.DataFrame(columns=columns)

    for filename in os.listdir(dataset_path):
        # read dataset
        raw_data = pd.read_csv(os.path.join(dataset_path, filename), sep='\t',header=None)

        # time features
        mean_abs = np.array(raw_data.abs().mean())
        std = np.array(raw_data.std())
        skew = np.array(raw_data.skew())
        kurtosis = np.array(raw_data.kurtosis())
        entropy = calculate_entropy(raw_data)
        rms = np.array(calculate_rms(raw_data))
        max_abs = np.array(raw_data.abs().max())
        p2p = calculate_p2p(raw_data)
        shape_factor=np.array(calculate_shape_factor(raw_data))
```

```
        impulse_factor=np.array(calculate_impulse_factor(raw_data))
        crest_factor=np.array(calculate_crest_factor(raw_data))
        clearance_factor=np.array(calculate_clearance_factor(raw_data))
        rms_kurt=np.multiply(rms,kurtosis)

        print('mean_abs',mean_abs)
        print('std',std)
        print('skew',skew)
        print('kurtosis',kurtosis)
        print('entropy',entropy)
        print('rms',rms)
        print('max',max_abs)
        print('p2p',p2p)
        print('shape_factor',shape_factor)
        print('impulse_factor',impulse_factor)
        print('crest_factor',crest_factor)
        print('clearance_factor',clearance_factor)
        print('rms*kurt',rms_kurt)

        #reshaping to merge into a pandas dataframe
        if id_set == 1:
            mean_abs = pd.DataFrame(mean_abs.reshape(1,8), columns=[c+'_mean' for c in cols1])
            std = pd.DataFrame(std.reshape(1,8), columns=[c+'_std' for c in cols1])
            skew = pd.DataFrame(skew.reshape(1,8), columns=[c+'_skew' for c in cols1])
            kurtosis = pd.DataFrame(kurtosis.reshape(1,8), columns=[c+'_kurtosis' for c in cols1])
            entropy = pd.DataFrame(entropy.reshape(1,8), columns=[c+'_entropy' for c in cols1])
            rms = pd.DataFrame(rms.reshape(1,8), columns=[c+'_rms' for c in cols1])
            max_abs = pd.DataFrame(max_abs.reshape(1,8), columns=[c+'_max' for c in cols1])
            p2p = pd.DataFrame(p2p.reshape(1,8), columns=[c+'_p2p' for c in cols1])
            shape_factor=pd.DataFrame(shape_factor.reshape(1,8), columns=[c+'_shape_factor' for c in cols1])
            impulse_factor=pd.DataFrame(impulse_factor.reshape(1,8), columns=[c+'_impulse_factor' for c in cols1])
            crest_factor=pd.DataFrame(crest_factor.reshape(1,8), columns=[c+'_crest_factor' for c in cols1])
            clearance_factor=pd.DataFrame(clearance_factor.reshape(1,8), columns=[c+'_clearance_factor' for c in cols1])
            rms_kurt=pd.DataFrame(rms_kurt.reshape(1,8), columns=[c+'_rms_kurt' for c in cols1])

        else:
            mean_abs = pd.DataFrame(mean_abs.reshape(1,4), columns=[c+'_mean' for c in cols2])
            std = pd.DataFrame(std.reshape(1,4), columns=[c+'_std' for c in cols2])
            skew = pd.DataFrame(skew.reshape(1,4), columns=[c+'_skew' for c in cols2])
            kurtosis = pd.DataFrame(kurtosis.reshape(1,4), columns=[c+'
```

```
_kurtosis' for c in cols2])
                entropy = pd.DataFrame(entropy.reshape(1,4), columns=[c+'_entropy' for c in cols2])
                rms = pd.DataFrame(rms.reshape(1,4), columns=[c+'_rms' for c in cols2])
                max_abs = pd.DataFrame(max_abs.reshape(1,4), columns=[c+'_max' for c in cols2])
                p2p = pd.DataFrame(p2p.reshape(1,4), columns=[c+'_p2p' for c in cols2])
                shape_factor = pd.DataFrame(shape_factor.reshape(1,4), columns=[c+'_shape_factor' for c in cols2])
                impulse_factor = pd.DataFrame(impulse_factor.reshape(1,4), columns=[c+'_impulse_factor' for c in cols2])
                crest_factor = pd.DataFrame(crest_factor.reshape(1,4), columns=[c+'_crest_factor' for c in cols2])
                clearance_factor = pd.DataFrame(clearance_factor.reshape(1,4), columns=[c+'_clearance_factor' for c in cols2])
                rms_kurt = pd.DataFrame(rms_kurt.reshape(1,4), columns=[c+'_rms_kurt' for c in cols2])

            #adding index as date of recording
            mean_abs.index = [filename]
            std.index = [filename]
            skew.index = [filename]
            kurtosis.index = [filename]
            entropy.index = [filename]
            rms.index = [filename]
            max_abs.index = [filename]
            p2p.index = [filename]
            shape_factor.index=[filename]
            impulse_factor.index=[filename]
            crest_factor.index=[filename]
            clearance_factor.index=[filename]
            rms_kurt.index=[filename]
            # concat
            merge = pd.concat([mean_abs, std, skew, kurtosis, entropy, rms, max_abs, p2p,shape_factor,impulse_factor,crest_factor, clearance_factor,rms_kurt], axis=1)
            data = data.append(merge)

        if id_set == 1:
            cols = [c+'_'+tf for c in cols1 for tf in time_features]
            data = data[cols]
        else:
            cols = [c+'_'+tf for c in cols2 for tf in time_features]
            data = data[cols]

        data.index = pd.to_datetime(data.index, format='%Y.%m.%d.%H.%M.%S')
        data = data.sort_index()
        return data

#extracting the time features for the 3 different datasets
set1 = time_features(dataset_path_1st, id_set=1)
set2 = time_features(dataset_path_2nd, id_set=2)
set3 = time_features(dataset_path_3rd, id_set=3)
```

```
#exporting the saved data to csv format
set1.to_csv('set1_timefeatures.csv')
set2.to_csv('set2_timefeatures.csv')
set3.to_csv('set3_timefeatures.csv')
#end
```

A.7 PCA and clustering of time features of Bearing data

```
#start
# -*- coding: utf-8 -*-
"""Copy of bearing data clustering with PCA

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
        V_jGoQkukboLyjdgLh33516z_pBAOoC4
"""



#importing required libraries
import numpy as np
import pandas as pd
import os, random

import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
import seaborn as sns
sns.set_style('whitegrid')

#Getting feature data from github
!git clone https://github.com/kailashjagadeesh/bearing_data_features.git

#loading the csv data as pandas dataframe
set1 = pd.read_csv('/content/bearing_data_features/new time domain features
    /set1_timefeatures.csv')
set2 = pd.read_csv('/content/bearing_data_features/new time domain features
    /set2_timefeatures.csv')
set3 = pd.read_csv('/content/bearing_data_features/new time domain features
    /set3_timefeatures.csv')

#removing unwanted features
set2.drop('B1_rms*kurt', inplace=True, axis=1)
set2.drop('B2_rms*kurt', inplace=True, axis=1)
set2.drop('B3_rms*kurt', inplace=True, axis=1)
set2.drop('B4_rms*kurt', inplace=True, axis=1)

#removing unwanted features
set3.drop('B1_rms*kurt', inplace=True, axis=1)
set3.drop('B2_rms*kurt', inplace=True, axis=1)
set3.drop('B3_rms*kurt', inplace=True, axis=1)
set3.drop('B4_rms*kurt', inplace=True, axis=1)

#displaying the first 5 elements of dataset 1 to verify proper loading of
```

```
    data
set1.head()

#merging the x and y sensor values of dataset 1 by averaging them
#this allows us to have uniform implementation for all datasets
set1['B1_mean'] = (set1['B1_a_mean'] + set1['B1_b_mean'])/2
set1['B1_std'] = (set1['B1_a_std'] + set1['B1_b_std'])/2
set1['B1_skew'] = (set1['B1_a_skew'] + set1['B1_b_skew'])/2
set1['B1_kurtosis'] = (set1['B1_a_kurtosis'] + set1['B1_b_kurtosis'])/2
set1['B1_entropy'] = (set1['B1_a_entropy'] + set1['B1_b_entropy'])/2
set1['B1_rms'] = (set1['B1_a_rms'] + set1['B1_b_rms'])/2
set1['B1_max'] = (set1['B1_a_max'] + set1['B1_b_max'])/2
set1['B1_p2p'] = (set1['B1_a_p2p'] + set1['B1_b_p2p'])/2
set1['B1_shape_factor'] = (set1['B1_a_shape_factor'] + set1['
    B1_b_shape_factor'])/2
set1['B1_impulse_factor'] = (set1['B1_a_impulse_factor'] + set1['
    B1_b_impulse_factor'])/2
set1['B1_crest_factor'] = (set1['B1_a_crest_factor'] + set1['
    B1_b_crest_factor'])/2
set1['B1_clearance_factor'] = (set1['B1_a_clearance_factor'] + set1['
    B1_b_clearance_factor'])/2
set1['B1_rms*kurt'] = (set1['B1_a_rms*kurt'] + set1['B1_b_rms*kurt'])/2


set1['B2_mean'] = (set1['B2_a_mean'] + set1['B2_b_mean'])/2
set1['B2_std'] = (set1['B2_a_std'] + set1['B2_b_std'])/2
set1['B2_skew'] = (set1['B2_a_skew'] + set1['B2_b_skew'])/2
set1['B2_kurtosis'] = (set1['B2_a_kurtosis'] + set1['B2_b_kurtosis'])/2
set1['B2_entropy'] = (set1['B2_a_entropy'] + set1['B2_b_entropy'])/2
set1['B2_rms'] = (set1['B2_a_rms'] + set1['B2_b_rms'])/2
set1['B2_max'] = (set1['B2_a_max'] + set1['B2_b_max'])/2
set1['B2_p2p'] = (set1['B2_a_p2p'] + set1['B2_b_p2p'])/2
set1['B2_shape_factor'] = (set1['B2_a_shape_factor'] + set1['
    B2_b_shape_factor'])/2
set1['B2_impulse_factor'] = (set1['B2_a_impulse_factor'] + set1['
    B2_b_impulse_factor'])/2
set1['B2_crest_factor'] = (set1['B2_a_crest_factor'] + set1['
    B2_b_crest_factor'])/2
set1['B2_clearance_factor'] = (set1['B2_a_clearance_factor'] + set1['
    B2_b_clearance_factor'])/2
set1['B2_rms*kurt'] = (set1['B2_a_rms*kurt'] + set1['B2_b_rms*kurt'])/2

set1['B3_mean'] = (set1['B3_a_mean'] + set1['B3_b_mean'])/2
set1['B3_std'] = (set1['B3_a_std'] + set1['B3_b_std'])/2
set1['B3_skew'] = (set1['B3_a_skew'] + set1['B3_b_skew'])/2
set1['B3_kurtosis'] = (set1['B3_a_kurtosis'] + set1['B3_b_kurtosis'])/2
set1['B3_entropy'] = (set1['B3_a_entropy'] + set1['B3_b_entropy'])/2
set1['B3_rms'] = (set1['B3_a_rms'] + set1['B3_b_rms'])/2
set1['B3_max'] = (set1['B3_a_max'] + set1['B3_b_max'])/2
set1['B3_p2p'] = (set1['B3_a_p2p'] + set1['B3_b_p2p'])/2
set1['B3_shape_factor'] = (set1['B3_a_shape_factor'] + set1['
    B3_b_shape_factor'])/2
set1['B3_impulse_factor'] = (set1['B3_a_impulse_factor'] + set1['
    B3_b_impulse_factor'])/2
set1['B3_crest_factor'] = (set1['B3_a_crest_factor'] + set1['
    B3_b_crest_factor'])/2
```

```
set1['B3_clearance_factor'] = (set1['B3_a_clearance_factor'] + set1['B3_b_clearance_factor'])/2
set1['B3_rms*kurt'] = (set1['B3_a_rms*kurt'] + set1['B3_b_rms*kurt'])/2

set1['B4_mean'] = (set1['B4_a_mean'] + set1['B4_b_mean'])/2
set1['B4_std'] = (set1['B4_a_std'] + set1['B4_b_std'])/2
set1['B4_skew'] = (set1['B4_a_skew'] + set1['B4_b_skew'])/2
set1['B4_kurtosis'] = (set1['B4_a_kurtosis'] + set1['B4_b_kurtosis'])/2
set1['B4_entropy'] = (set1['B4_a_entropy'] + set1['B4_b_entropy'])/2
set1['B4_rms'] = (set1['B4_a_rms'] + set1['B4_b_rms'])/2
set1['B4_max'] = (set1['B4_a_max'] + set1['B4_b_max'])/2
set1['B4_p2p'] = (set1['B4_a_p2p'] + set1['B4_b_p2p'])/2
set1['B4_shape_factor'] = (set1['B4_a_shape_factor'] + set1['B4_b_shape_factor'])/2
set1['B4_impulse_factor'] = (set1['B4_a_impulse_factor'] + set1['B4_b_impulse_factor'])/2
set1['B4_crest_factor'] = (set1['B4_a_crest_factor'] + set1['B4_b_crest_factor'])/2
set1['B4_clearance_factor'] = (set1['B4_a_clearance_factor'] + set1['B4_b_clearance_factor'])/2
set1['B4_rms*kurt'] = (set1['B4_a_rms*kurt'] + set1['B4_b_rms*kurt'])/2

#selecting the required feature columns
set1 = set1[['B1_mean','B1_std','B1_skew','B1_kurtosis','B1_entropy','B1_rms','B1_max','B1_p2p','B1_shape_factor','B1_impulse_factor','B1_crest_factor','B1_clearance_factor','B1_rms*kurt',
             'B2_mean','B2_std','B2_skew','B2_kurtosis','B2_entropy','B2_rms','B2_max','B2_p2p','B2_shape_factor','B2_impulse_factor','B2_crest_factor','B2_clearance_factor','B2_rms*kurt',
             'B3_mean','B3_std','B3_skew','B3_kurtosis','B3_entropy','B3_rms','B3_max','B3_p2p','B3_shape_factor','B3_impulse_factor','B3_crest_factor','B3_clearance_factor','B3_rms*kurt',
             'B4_mean','B4_std','B4_skew','B4_kurtosis','B4_entropy','B4_rms','B4_max','B4_p2p','B4_shape_factor','B4_impulse_factor','B4_crest_factor','B4_clearance_factor','B4_rms*kurt']]

set2= set2[['B1_mean', 'B1_std', 'B1_skew', 'B1_kurtosis','B1_entropy', 'B1_rms', 'B1_max', 'B1_p2p', 'B1_shape_factor','B1_impulse_factor', 'B1_crest_factor', 'B1_clearance_factor','B2_mean', 'B2_std', 'B2_skew', 'B2_kurtosis', 'B2_entropy', 'B2_rms',
     'B2_max', 'B2_p2p', 'B2_shape_factor', 'B2_impulse_factor','B2_crest_factor', 'B2_clearance_factor', 'B3_mean', 'B3_std','B3_skew', 'B3_kurtosis', 'B3_entropy', 'B3_rms', 'B3_max', 'B3_p2p',
     'B3_shape_factor', 'B3_impulse_factor', 'B3_crest_factor','B3_clearance_factor', 'B4_mean', 'B4_std', 'B4_skew', 'B4_kurtosis','B4_entropy', 'B4_rms', 'B4_max', 'B4_p2p', 'B4_shape_factor',
     'B4_impulse_factor', 'B4_crest_factor', 'B4_clearance_factor']]

set3= set3[['B1_mean', 'B1_std', 'B1_skew', 'B1_kurtosis','B1_entropy', 'B1_rms', 'B1_max', 'B1_p2p', 'B1_shape_factor','B1_impulse_factor', 'B1_crest_factor', 'B1_clearance_factor','B2_mean', 'B2_std', 'B2_skew',
```

```
    'B2_kurtosis', 'B2_entropy', 'B2_rms',
        'B2_max', 'B2_p2p', 'B2_shape_factor', 'B2_impulse_factor','B2_crest_factor', 'B2_clearance_factor', 'B3_mean', 'B3_std','B3_skew', 'B3_kurtosis', 'B3_entropy', 'B3_rms', 'B3_max', 'B3_p2p',
        'B3_shape_factor', 'B3_impulse_factor', 'B3_crest_factor','B3_clearance_factor', 'B4_mean', 'B4_std', 'B4_skew', 'B4_kurtosis','B4_entropy', 'B4_rms', 'B4_max', 'B4_p2p', 'B4_shape_factor',
        'B4_impulse_factor', 'B4_crest_factor', 'B4_clearance_factor']]


set1.drop('B1_rms*kurt', inplace=True, axis=1)
set1.drop('B2_rms*kurt', inplace=True, axis=1)
set1.drop('B3_rms*kurt', inplace=True, axis=1)
set1.drop('B4_rms*kurt', inplace=True, axis=1)

set1.head()

#function to visualise the dataset
def plot_features(df):
    fig, axes = plt.subplots(4, 1, figsize=(15, 5*4))

    axes[0].plot(df['B1_mean'])
    axes[0].plot(df['B2_mean'])
    axes[0].plot(df['B3_mean'])
    axes[0].plot(df['B4_mean'])
    axes[0].legend(['B1','B2','B3','B4'])
    axes[0].set_title('Mean')

    axes[1].plot(df['B1_rms'])
    axes[1].plot(df['B2_rms'])
    axes[1].plot(df['B3_rms'])
    axes[1].plot(df['B4_rms'])
    axes[1].legend(['B1','B2','B3','B4'])
    axes[1].set_title('RMS')

    axes[2].plot(df['B1_skew'])
    axes[2].plot(df['B2_skew'])
    axes[2].plot(df['B3_skew'])
    axes[2].plot(df['B4_skew'])
    axes[2].legend(['B1','B2','B3','B4'])
    axes[2].set_title('Skewness')

    axes[3].plot(df['B1_kurtosis'])
    axes[3].plot(df['B2_kurtosis'])
    axes[3].plot(df['B3_kurtosis'])
    axes[3].plot(df['B4_kurtosis'])
    axes[3].legend(['B1','B2','B3','B4'])
    axes[3].set_title('Kurtosis')
    fig1, axes1 = plt.subplots(4, 1, figsize=(15, 5*4))

    axes1[0].plot(df['B1_entropy'])
    axes1[0].plot(df['B2_entropy'])
    axes1[0].plot(df['B3_entropy'])
    axes1[0].plot(df['B4_entropy'])
```

```
    axes1[0].legend(['B1','B2','B3','B4'])
    axes1[0].set_title('entropy')

    axes1[1].plot(df['B1_rms'])
    axes1[1].plot(df['B2_rms'])
    axes1[1].plot(df['B3_rms'])
    axes1[1].plot(df['B4_rms'])
    axes1[1].legend(['B1','B2','B3','B4'])
    axes1[1].set_title('rms')

    axes1[2].plot(df['B1_max'])
    axes1[2].plot(df['B2_max'])
    axes1[2].plot(df['B3_max'])
    axes1[2].plot(df['B4_max'])
    axes1[2].legend(['B1','B2','B3','B4'])
    axes1[2].set_title('max')

    axes1[3].plot(df['B1_p2p'])
    axes1[3].plot(df['B2_p2p'])
    axes1[3].plot(df['B3_p2p'])
    axes1[3].plot(df['B4_p2p'])
    axes1[3].legend(['B1','B2','B3','B4'])
    axes1[3].set_title('B4_p2p')

    fig2, axes2 = plt.subplots(4, 1, figsize=(15, 5*4))

    axes2[0].plot(df['B1_shape_factor'])
    axes2[0].plot(df['B2_shape_factor'])
    axes2[0].plot(df['B3_shape_factor'])
    axes2[0].plot(df['B4_shape_factor'])
    axes2[0].legend(['B1','B2','B3','B4'])
    axes2[0].set_title('shape_factor')

    axes2[1].plot(df['B1_impulse_factor'])
    axes2[1].plot(df['B2_impulse_factor'])
    axes2[1].plot(df['B3_impulse_factor'])
    axes2[1].plot(df['B4_impulse_factor'])
    axes2[1].legend(['B1','B2','B3','B4'])
    axes2[1].set_title('impulse_factor')

    axes2[2].plot(df['B1_crest_factor'])
    axes2[2].plot(df['B2_crest_factor'])
    axes2[2].plot(df['B3_crest_factor'])
    axes2[2].plot(df['B4_crest_factor'])
    axes2[2].legend(['B1','B2','B3','B4'])
    axes2[2].set_title('crest_factor')

    axes2[3].plot(df['B1_clearance_factor'])
    axes2[3].plot(df['B2_clearance_factor'])
    axes2[3].plot(df['B3_clearance_factor'])
    axes2[3].plot(df['B4_clearance_factor'])
    axes2[3].legend(['B1','B2','B3','B4'])
    axes2[3].set_title('clearance_factor')

#visualisation of dataset 1
plot_features(set1)
```

```
set1.shape

cols=set2.columns

#taking only first 12 columns coss they correspond to bearing 1, next 12 correspond to bearing 2 and so on.
set1_bearing1=set1[cols[:12]].to_numpy()
print(set1_bearing1.shape)

#70:30 split of training and testing
Y_train=set1_bearing1[0:1510,:]
Y_test=set1_bearing1[1510:-1,:]

# PRINCIPAL COMPONENT ANALYSIS

#import stuff
from sklearn.decomposition import PCA
from sklearn import preprocessing

#preprocess by standardizing
scaler = preprocessing.StandardScaler().fit(Y_train)
Y_scaled_train = scaler.transform(Y_train)
Y_scaled_test = scaler.transform(Y_test)

#perform PCA
pca = PCA(n_components=12)
pca.fit(Y_scaled_train)

#visualize variance explained
components = np.linspace(0,len(pca.explained_variance_ratio_)-1,len(pca.explained_variance_ratio_))
ax = plt.gca()
ax2 = ax.twinx()
ax2.plot(components, np.cumsum(pca.explained_variance_ratio_),color="r")
ax.bar(components, pca.explained_variance_ratio_)
ax.set_xlabel('number of components')
ax.xaxis.set_ticks(np.arange(0, 20, 1))
ax.set_ylabel('explained variance')
ax2.set_ylabel('Variance cumulative sum')
ax2.spines['right'].set_color('red')
ax2.tick_params(colors='red')
ax2.yaxis.label.set_color('red')

#Obtaining Eigen Values
plt.plot(pca.explained_variance_,"-o")
ax = plt.gca()
plt.axhline(y=1,linestyle='--',color='red')
ax.set_xlabel('components')
ax.xaxis.set_ticks(np.arange(0, 20, 3))
ax.set_ylabel('Eigenvalues')

#transforming to lower dimension feature space
pca = PCA(n_components=2)
pca.fit(Y_scaled_train)
train_transform=pca.transform(Y_scaled_train)
test_transform=pca.transform(Y_scaled_test)
```

```
print(train_transform.shape)

#merging training and testing data after transformation
print((train_transform))
import numpy as np
print(test_transform)
t=np.vstack([train_transform ,test_transform])
print(t.shape)

#visualisation of the 2D data
data=t
plt.scatter(data[:,0],data[:,1])
plt.show()

#installation of library
!pip install -U scikit-learn

#automation to choose the proper number of clusters
import sklearn.metrics as metrics
import sklearn
#import sklearn.model_selection.ParameterGrid as ParameterGrid
parameters = [2, 3, 4, 5, 10, 15, 20, 25, 30, 35, 40]
# instantiating ParameterGrid, pass number of clusters as input
parameter_grid = sklearn.model_selection.ParameterGrid({'n_clusters':
    parameters})
best_score = -1
kmeans_model = KMeans()     # instantiating KMeans model
silhouette_scores = []
# evaluation based on silhouette_score
for p in parameter_grid:
    kmeans_model.set_params(**p)    # set current hyper parameter
    kmeans_model.fit(data)          # fit model on wine dataset, this will
        find clusters based on parameter p
    ss = metrics.silhouette_score(data, kmeans_model.labels_)   # calculate
         silhouette_score
    silhouette_scores += [ss]       # store all the scores
    print('Parameter:', p, 'Score', ss)
    # check p which has the best score
    if ss > best_score:
        best_score = ss
        best_grid = p
# plotting silhouette score
plt.bar(range(len(silhouette_scores)), list(silhouette_scores), align='
    center', color='#722f59', width=0.5)
plt.xticks(range(len(silhouette_scores)), list(parameters))
plt.title('Silhouette Score', fontweight='bold')
plt.xlabel('Number of Clusters')
plt.show()

#Import required module
from sklearn.cluster import KMeans

#Initialize the class object for K means with optimum number of clusters
kmeans = KMeans(n_clusters= 3)

#predict the labels of clusters.
```

```python
label = kmeans.fit_predict(t)

print(label)

#visualisation of the clustered data along with the clusters
#Getting the Centroids
centroids = kmeans.cluster_centers_
u_labels = np.unique(label)

#plotting the results:

for i in u_labels:
    plt.scatter(t[label == i , 0] , t[label == i , 1] , label = i)
plt.scatter(centroids[:,0] , centroids[:,1] , s = 80, color = 'k')
plt.legend()
plt.show()
#end
```

REFERENCES

[1] Carlos F Alcala and S Joe Qin. "Unified analysis of diagnosis methods for process monitoring". In: *IFAC Proceedings Volumes* 42.8 (2009), pp. 1007–1012.

[2] Nicolò Bachschmid, Paolo Pennacchi, and Ezio Tanzi. *Cracked rotors: a survey on static and dynamic behaviour including modelling and diagnosis*. Springer Science & Business Media, 2010.

[3] Daniel Balageas, Claus-Peter Fritzen, and Alfredo Güemes. *Structural health monitoring*. Vol. 90. John Wiley & Sons, 2010.

[4] Sandaram Buchaiah and Piyush Shakya. "Bearing fault diagnosis and prognosis using data fusion based feature extraction and feature selection". In: *Measurement* 188 (2022), p. 110506.

[5] E Peter Carden and Paul Fanning. "Vibration based condition monitoring: a review". In: *Structural health monitoring* 3.4 (2004), pp. 355–377.

[6] Anurag Choudhary, Tauheed Mian, and Shahab Fatima. "Convolutional neural network based bearing fault diagnosis of rotating machine using thermal images". In: *Measurement* 176 (2021), p. 109196.

[7] A Choudhury and N Tandon. "A theoretical model to predict vibration response of rolling bearings to distributed defects under radial load". In: (1998).

[8] Christian Cremona and Joao Santos. "Structural health monitoring as a big-data problem". In: *Structural Engineering International* 28.3 (2018), pp. 243–254.

[9] Ning Ding et al. "Journal bearing seizure degradation assessment and remaining useful life prediction based on long short-term memory neural network". In: *Measurement* 166 (2020), p. 108215.

[10] William Gousseau et al. "Analysis of the Rolling Element Bearing data set of the Center for Intelligent Maintenance Systems of the University of Cincinnati". In: *CM2016*. 2016.

[11] Pankaj Gupta and MK Pradhan. "Fault detection analysis in rolling element bearing: A review". In: *Materials Today: Proceedings* 4.2 (2017), pp. 2085–2094.

[12] Gangjin Huang, Yuanliang Zhang, and Jiayu Ou. "Transfer remaining useful life estimation of bearing using depth-wise separable convolution recurrent network". In: *Measurement* 176 (2021), p. 109090.

[13] Wenxiu Lu and Fulei Chu. "Shaft crack identification based on vibration and AE signals". In: *Shock and Vibration* 18.1-2 (2011), pp. 115–126.

[14] NASA. *A collection of prognostic datasets from all around the world*. `https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/`.

[15] Rodrigo Nicoletti, Aldemir A Cavalini, and Valder Steffen. "Detection of cracks in rotating shafts by using the combination resonances approach and the approximated entropy algorithm". In: *Shock and Vibration* 2018 (2018).

[16] Guangxing Niu et al. “An optimized adaptive PReLU-DBN for rolling element bearing fault diagnosis”. In: *Neurocomputing* 445 (2021), pp. 26–34.

[17] Sagi Rathna Prasad and AS Sekhar. “Detection and localization of fatigue-induced transverse crack in a rotor shaft using principal component analysis”. In: *Structural Health Monitoring* 20.2 (2021), pp. 513–531.

[18] Akthem Rehab et al. “Bearings Fault Detection Using Hidden Markov Models and Principal Component Analysis Enhanced Features”. In: *arXiv preprint arXiv:2104.10519* (2021).

[19] AS Sekhar. “Crack identification in a rotor system: a model-based approach”. In: *Journal of sound and vibration* 270.4-5 (2004), pp. 887–902.

[20] CS Sunnersjö. “Varying compliance vibrations of rolling bearings”. In: *Journal of sound and vibration* 58.3 (1978), pp. 363–373.

[21] TE Tallian and OG Gustafsson. “Progress in rolling bearing vibration research and control”. In: *ASLE transactions* 8.3 (1965), pp. 195–207.

[22] Naresh Tandon and Achintya Choudhury. “A review of vibration and acoustic measurement methods for the detection of defects in rolling element bearings”. In: *Tribology international* 32.8 (1999), pp. 469–480.

[23] Zhenya Wang, Ligang Yao, and Yongwu Cai. “Rolling bearing fault diagnosis using generalized refined composite multiscale sample entropy and optimized support vector machine”. In: *Measurement* 156 (2020), p. 107574.

[24] FP Wardle and SY Poon. “Rolling bearing noise-cause and cure”. In: *Chartered mechanical engineer* 30 (1983), pp. 36–40.