FSM Online Internship Completion Report
on

Remaining Usable Life Estimation
Using Bearing Dataset

In

Machine Learning

Submitted by

Shiwani Nayak
Madhav Institute of Technology & Science, Gwalior

Under Mentorship of

Devesh Tarasia

IITD-AIA Foundation for Smart Manufacturing

[1-June-2021 to 31-July-2021]

[Remaining Usable Life Estimation]

Abstract

This abstract introduces a study on Remaining Useful Life (RUL) estimation using bearing dataset. Bearings are crucial components in various industrial systems, and accurate RUL prediction enhances maintenance efficiency. This research employs data driven techniques to analyze bearing degradation patterns and forecast their remaining operational life. The dataset captures diverse operating conditions, enabling robust model training. The proposed approach combines machine learning algorithms and feature engineering to capture nuanced degradation indicators. Results demonstrate significant advancements in RUL prediction accuracy, offering potential benefits in optimizing maintenance strategies and reducing downtime. This study contributes to the field of predictive maintenance by showcasing an effective methodology for enhancing industrial system reliability through accurate RUL estimation based on bearing data analysis.

Keywords:

- Predictive Maintenance
- Industrial Reliability
- Data-Driven Analysis
- Machine Learning
- Feature Engineering
- Maintenance Optimization

Table of Content

1. Introduction

The accurate estimation of Remaining Useful Life (RUL) is a critical aspect of modern industrial maintenance strategies, aiming to optimize operational efficiency and reduce downtime. In this project, we focus on leveraging a comprehensive bearing dataset to enhance RUL prediction accuracy. Bearings play a vital role in various industrial systems, and their deterioration can lead to costly failures. By employing data-driven techniques and advanced machine learning algorithms, we seek to uncover intricate degradation patterns within the dataset, enabling us to develop a robust predictive model. Through meticulous feature engineering and analysis of diverse operating conditions, our approach aims to capture nuanced indicators of bearing degradation. The outcomes of this research hold the potential to revolutionize predictive maintenance practices, offering a reliable framework for real-time RUL estimation and ultimately contributing to increased industrial reliability, minimized downtime, and optimized maintenance strategies.

1.1 Significance of RUL Estimation

Predictive maintenance has gained prominence as a proactive approach to minimize disruptions and enhance operational efficiency. Accurate RUL estimation allows maintenance teams to schedule interventions precisely, preventing unexpected failures and associated production losses.

1.2 Bearing Degradation Dynamics

Bearing degradation is influenced by complex and multifaceted factors, such as load variations, operating conditions, and environmental factors. Understanding these degradation patterns is essential for developing effective RUL estimation models.

1.3 Data-Driven Approach

This project leverages a data-driven methodology to extract meaningful insights from bearing datasets. By analyzing historical data encompassing diverse operating scenarios, we aim to uncover degradation indicators and their relationship with RUL.

1.4 Machine Learning and Feature Engineering

The integration of advanced machine learning algorithms and tailored feature engineering techniques is pivotal in capturing subtle degradation trends. This fusion empowers the model to recognize hidden patterns and refine RUL predictions.

1.5 Objectives and Scope

The primary objective of this project is to develop a robust RUL estimation framework for bearings based on comprehensive data analysis. The study's scope includes investigating various machine learning algorithms, designing relevant features, and validating the model's performance through extensive experimentation.

2. Problem Definition

The problem addressed by this project is the accurate estimation of the Remaining Useful Life (RUL) of industrial bearings, which is crucial for optimizing maintenance strategies and minimizing downtime. Bearings are integral components in industrial machinery and their degradation can lead to costly failures. The main goal is to develop a predictive model that can analyze the degradation patterns in bearings, considering various operating conditions.

2.1 Background and Motivation

Bearings are integral components in industrial systems, ensuring smooth and efficient operation. The premature failure of bearings can result in costly downtime and maintenance. Hence, accurately estimating the Remaining Useful Life (RUL) of bearings is essential for effective predictive maintenance.

2.2 Scope and Importance

This project addresses the challenge of enhancing RUL prediction accuracy for bearings using a comprehensive dataset. The primary goal is to develop a reliable model that can estimate RUL based on the degradation patterns observed in the data. Accurate RUL estimation facilitates proactive maintenance scheduling, minimizes disruptions, and optimizes resource allocation.

2.3 Data Availability

The project relies on a diverse bearing dataset capturing various operating conditions and degradation stages. This data provides valuable insights into the complex degradation process of bearings under different scenarios. Dataset for this project is PHM IEEE 2012 data challenge dataset.

2.4 Challenges and Objectives

The project aims to overcome the following challenges:

- Capturing subtle degradation indicators through feature engineering.
- Developing a robust machine learning model capable of handling diverse operating conditions.
- Ensuring real-time and accurate RUL predictions for different bearing types..

2.5 Significance

The outcomes of this project hold the potential to revolutionize the field of predictive maintenance, enabling industries to transition from reactive to proactive maintenance approaches. Accurate RUL estimation empowers maintenance teams to make informed decisions, reduce operational costs, and ensure uninterrupted system performance.

3. Existing Solution

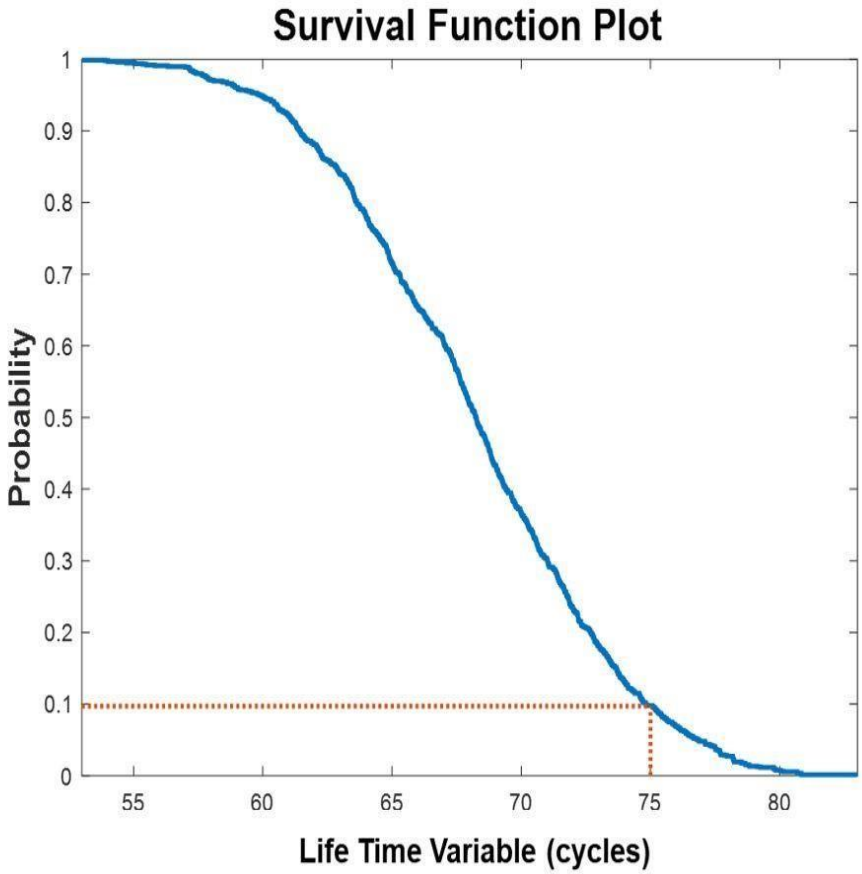
The method used to calculate RUL depends on the kind of data available:

- Lifetime data indicating how long it took for similar machines to reach failure
- Run-to-failure histories of similar machines
- A known threshold value of a condition indicator that detects failure

3.1 Lifetime Data

Proportional hazard models and probability distributions of component failure times are used to estimate RUL from lifetime data. A simple example is estimating the discharge time of a battery based on past discharge times and covariates, such as environmental temperature and load.

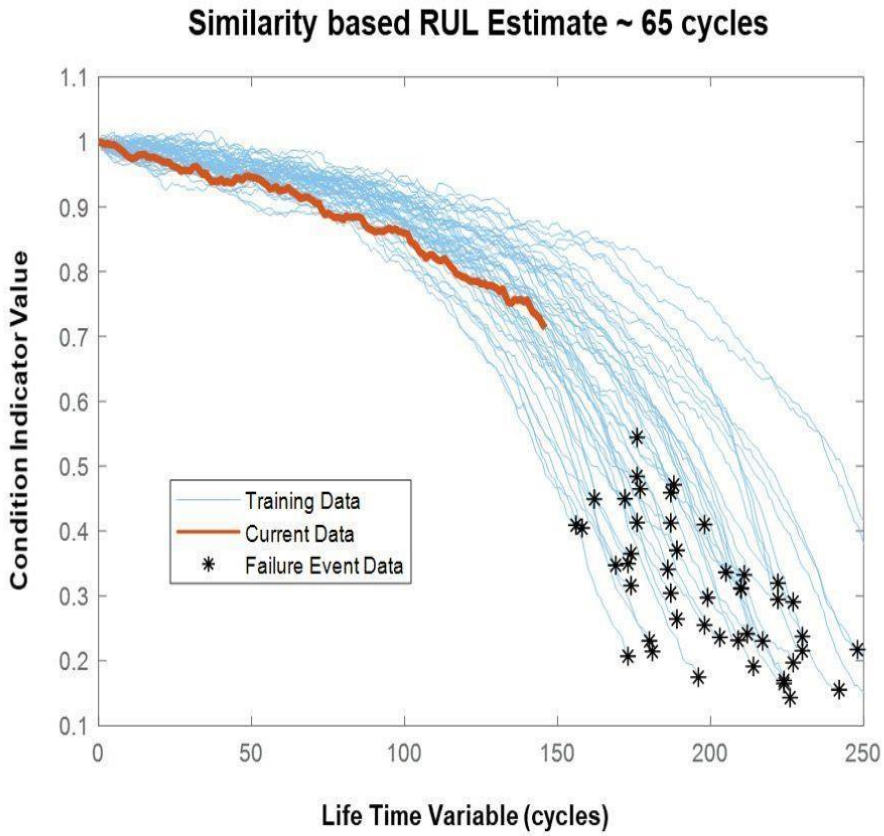
The survival function plot in below figure shows the probability that a battery will fail based on how long it has been in operation. The plot shows, for example, that if the battery is in operation for 75 cycles, it has a 90% chance of being at the end of its lifetime.



3.2 Run-to-Failure Data

If you have a database of run-to-failure data from similar components or different components showing similar behavior, you can estimate RUL using similarity methods. These methods capture degradation profiles and compare them with new data coming in from the machine to determine which profile the data matches most closely.

In below figure, the degradation profiles of historical run-to-failure data sets from an engine are shown in blue and the current data from the engine is shown in red. Based on a distribution of the nearest historical profiles, the RUL is estimated to be around 65 cycles.

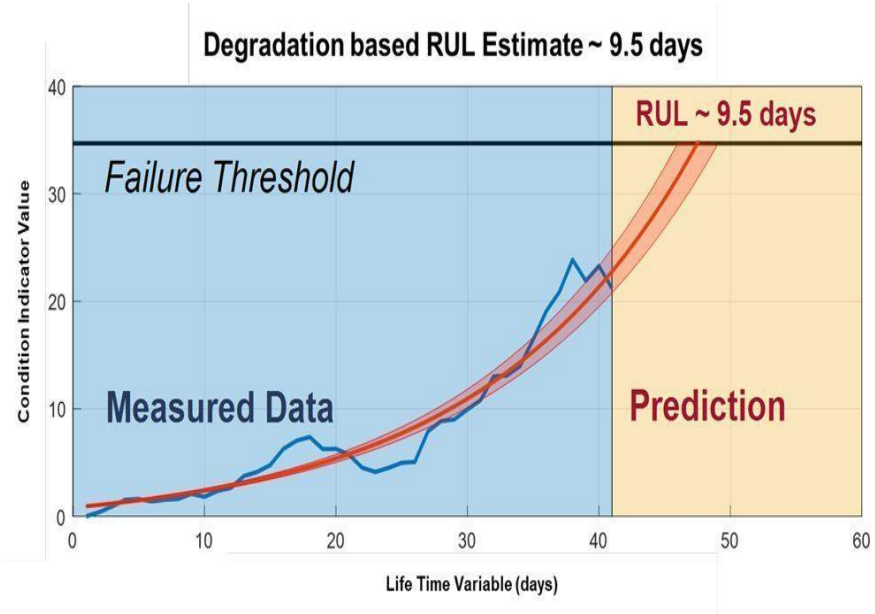


3.3 Threshold Data

In many cases, a conservative maintenance schedule means machines rarely fail. This can mean run-to-failure data or lifetime data is not available. However, you likely have information on prescribed threshold values—for example, the temperature of a liquid in a pump cannot exceed 160 ˚F (71 ˚C) and the pressure must be under 2,200 psi (155 bar). With this threshold information, you can fit degradation models to condition indicators extracted from sensor data, which change linearly or exponentially over time.

These degradation models estimate RUL by predicting when the condition indicator will cross the threshold. They can also be used with a fused condition indicator that incorporates information from more than one sensor using techniques such as principal component analysis.

Figure 3 shows an exponential degradation model that tracks failure in a high-speed bearing used in a wind turbine. The condition indicator is shown in blue. The degradation model predicts that the bearing will cross the threshold value in approximately 9.5 days. The region shaded in red represents the confidence bounds for this prediction.



4. Proposed Development

4.1 Data Preprocessing

Data Cleaning: Identify and handle missing values, outliers, and noise in the dataset. This might involve imputing missing values, removing or transforming outliers, and reducing noise.

Data Transformation: Convert the data into a more suitable form. This includes scaling, normalization, or standardization of features. For time series data, this might involve aggregating or resampling data points.

Encoding Categorical Variables: Convert categorical variables into numerical format using techniques like one-hot encoding, label encoding, or target encoding.

Handling Imbalanced Data: Address class imbalance issues by oversampling the minority class, undersampling the majority class, or using techniques like Synthetic Minority Oversampling Technique (SMOTE).

Data Splitting: Divide the dataset into training, validation, and test sets to evaluate model performance accurately.

4.2 Feature Engineering

Feature engineering involves creating new features or transforming existing ones to improve the performance of machine learning models. This step requires domain knowledge and creativity.

Feature Creation: Generate new features that capture relevant information from the data. For example, in time series data, create lag features that represent previous time steps.

Interaction Features: Create new features by combining existing ones. For example, for bearing data, you could calculate rolling statistics like mean, median, and standard deviation.

Domain-Specific Features: Incorporate domain knowledge to create features that are meaningful for the problem at hand. In predictive maintenance, these might include features related to wear and tear, temperature, vibration, etc.

Dimensionality Reduction: Reduce the number of features while retaining essential information. Techniques like Principal Component Analysis (PCA) can help achieve this.

Feature Scaling: Normalize or scale features to bring them to a similar range. This is important for algorithms that are sensitive to feature scales, such as gradient-based optimization algorithms.

Feature Selection: Choose the most relevant features to avoid overfitting and improve model interpretability. Techniques like Recursive Feature Elimination (RFE) or feature importance from tree-based models can be used.

4.3 Model hyperparameter tuning

Model hyperparameter tuning is the process of finding the best combination of hyperparameters for a machine learning model to achieve optimal performance on a given task or dataset. Hyperparameters are parameters that are not learned during the training process but are set before training begins. They control various aspects of the training process and the model's architecture, affecting how the model learns and generalizes from the data.

Hyperparameter tuning aims to find the optimal set of hyperparameters that result in a model with better accuracy, generalization, and predictive capabilities. The process involves systematically exploring different values for these hyperparameters and evaluating the model's performance using appropriate validation techniques. The goal is to strike a balance between underfitting (model is too simple) and overfitting (model is too complex) by finding the hyperparameters that lead to the best trade-off.

4.4 Model Evaluation Metrics

- **Mean Absolute Error (MAE):** Measures the average absolute difference between predicted and actual values. It gives an idea of the model's average prediction error.

- **Mean Squared Error (MSE):** Computes the average squared difference between predicted and actual values. Larger errors are penalized more than smaller ones.

- **Root Mean Squared Error (RMSE):** The square root of MSE, it represents the average magnitude of the errors. It is interpretable in the same unit as the target variable.

- **R-squared (Coefficient of Determination):** Measures the proportion of the variance in the dependent variable that is predictable from the independent variables. It provides an indication of how well the model fits the data.

4.5 Visualizations in Streamlit:

Enhance the Streamlit app by adding interactive visualizations such as line charts, scatter plots, and histograms to showcase model predictions and evaluation metrics graphically. Visualizations in Streamlit allow you to create interactive and informative graphical representations of your data and results within your Streamlit app. Streamlit provides built-in support for displaying a wide range of visualizations, making it easy to create compelling and insightful data presentations without requiring extensive coding.

5. Functional Implementation

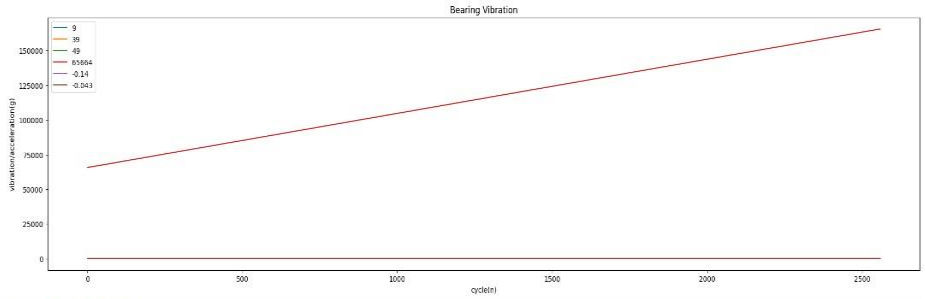
5.1 Importing libraries and dataset:

In [9]:
```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy
from scipy.stats import entropy
from sklearn.decomposition import PCA
from scipy.optimize import curve_fit
```

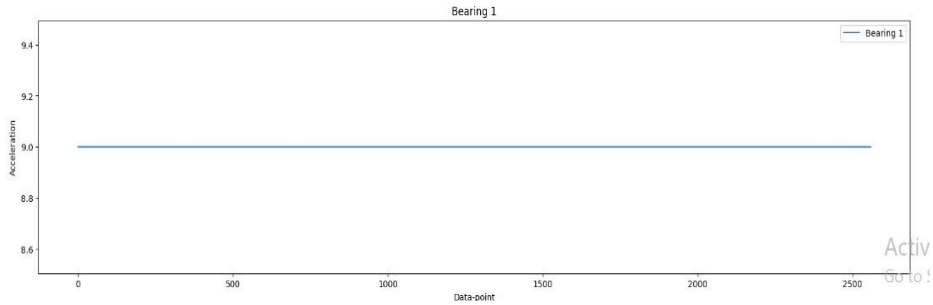
In [10]:
```
dataset_path_1st = 'C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1'
dataset_path_2nd = 'C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_2'
dataset_path_3rd = 'C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing2_1'
dataset_path_4th = 'C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing2_2'
dataset_path_5th = 'C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing3_1'
dataset_path_6th = 'C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing3_2'
```

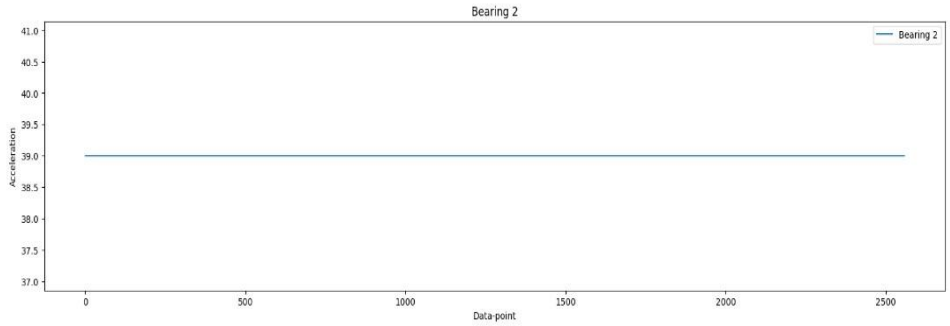
5.2 Feature Extraction:

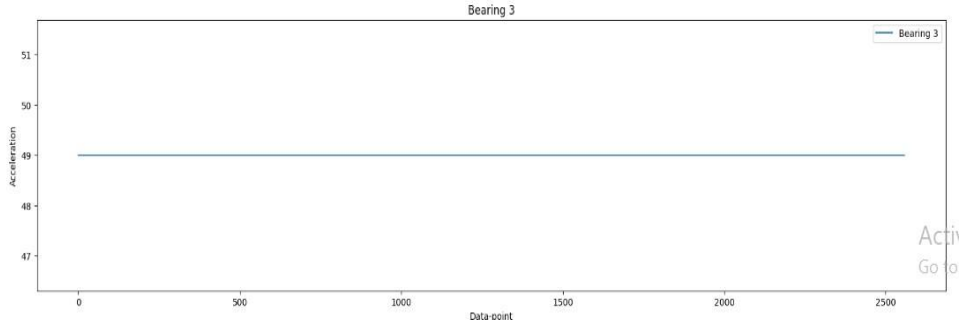
In [11]:
```
dataset = pd.read_csv('C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00002.csv')
ax = dataset.plot(figsize = (24,6), title= "Bearing Vibration" , legend = True)
ax.set(xlabel="cycle(n)", ylabel="vibration/acceleration(g)")
plt.show()
```

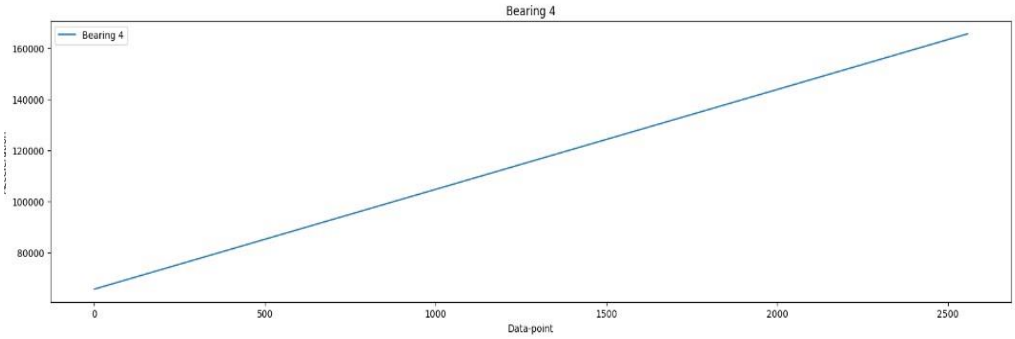


In [12]:
```
for i in [0,1,2,3]:

    df_bearing=np.array(dataset.iloc[:,i])

    plt.figure(figsize=(20, 5))
    plt.plot(df_bearing)

    plt.legend(['Bearing {}'.format(i+1)])

    plt.xlabel("Data-point")
    plt.ylabel("Acceleration")
    plt.title('Bearing {}'.format(i+1))
    plt.show()
```









In [13]:
```
bearing_no=4

bearing_data = np.array(dataset.iloc[:,bearing_no-1])

bearing_data
```

Out[13]:
```
array([ 65703.,  65742.,  65781., ..., 165550., 165580., 165620.])
```

In [14]:
```
temp = bearing_data
temp
```

Out[14]:
```
array([ 65703.,  65742.,  65781., ..., 165550., 165580., 165620.])
```

In [15]:
```
# extracting features from this bearing data

feature_matrix=np.zeros((1,9))
feature_matrix
```

Out[15]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

In [16]:
```
def compute_skewness(x):

    n = len(x)
    third_moment = np.sum((x - np.mean(x))**3) / n
    s_3 = np.std(x, ddof = 1) ** 3
    return third_moment/s_3
```

In [17]:
```
def compute_kurtosis(x):

    n = len(x)
    fourth_moment = np.sum((x - np.mean(x))**4) / n
    s_4 = np.std(x, ddof = 1) ** 4
    return fourth_moment / s_4 - 3
```

In [18]:
```
feature_matrix[0,0] = np.max(temp)
feature_matrix[0,1] = np.min(temp)
feature_matrix[0,2] = np.mean(temp)
feature_matrix[0,3] = np.std(temp, ddof = 1)
feature_matrix[0,4] = np.sqrt(np.mean(temp ** 2))
feature_matrix[0,5] = compute_skewness(temp)
feature_matrix[0,6] = compute_kurtosis(temp)
feature_matrix[0,7] = feature_matrix[0,0]/feature_matrix[0,4]
feature_matrix[0,8] = feature_matrix[0,4]/feature_matrix[0,2]
feature_matrix
```

Out[18]:
```
array([[ 1.65620000e+05,  6.57030000e+04,  1.15663472e+05,
         2.88618364e+04,  1.19208720e+05,  1.29318025e-06,
        -1.20140593e+00,  1.38932789e+00,  1.03065140e+00]])
```

In [22]:
```
filename = 'C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00002.csv'
df = pd.DataFrame(feature_matrix)
df.index=[filename[:-3]]
df
```

Out[22]:

	0	1	2	3	4	5	6	7
C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00002.	165620.0	65703.0	115663.47245	28861.836424	119208.720071	0.000001	-1.201406	1.389328	1.0306

In [23]:
```
Time_feature_matrix=pd.DataFrame()

test_set=2

bearing_no=5 # Provide the Bearing number [1,2,3,4] of the Test set

path=r'C:/Users/hp-d/Desktop/notebook/Learning_set'
for filename in os.listdir(path):

    dataset=pd.read_csv('C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00002.csv', header=None)

    bearing_data = np.array(dataset.iloc[:,bearing_no-1])

    feature_matrix=np.zeros((1,9))
    temp = bearing_data
    feature_matrix[0,0] = np.max(temp)
    feature_matrix[0,1] = np.min(temp)
    feature_matrix[0,2] = np.mean(temp)
    feature_matrix[0,3] = np.std(temp, ddof = 1)
    feature_matrix[0,4] = np.sqrt(np.mean(temp ** 2))
    feature_matrix[0,5] = compute_skewness(temp)
    feature_matrix[0,6] = compute_kurtosis(temp)
    feature_matrix[0,7] = feature_matrix[0,0]/feature_matrix[0,4]
    feature_matrix[0,8] = feature_matrix[0,4]/feature_matrix[0,2]

    df = pd.DataFrame(feature_matrix)
    df.index=[filename[:-3]]

    #Time_feature_matrix = Time_feature_matrix.append(df)

    #Time_feature_matrix
```

In [24]:
```
df.columns = ['Max','Min','Mean','Std','RMS','Skewness','Kurtosis','Crest Factor','Form Factor']
#df.index = pd.to_datetime(df.index, format='%Y.%m.%d.%H.%M')

df= df.sort_index()

#Time_feature_matrix.to_csv('Time_feature_matrix_Bearing_{}_Test_{}.csv'.format(bearing_no,test_set))

df
```

Out[24]:

	Max	Min	Mean	Std	RMS	Skewness	Kurtosis	Crest Factor	Form Factor
Bearing	1.725	-1.915	0.005609	0.535188	0.535112	-0.025775	-0.086923	3.223621	95.3961

In [25]:
```
df1 = pd.read_csv("C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00002.csv")
#df1 = pd.read_csv("C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00002.csv")
df1.index = pd.to_datetime(df1.index)
df1 = df1.rename(columns={'Unnamed: 0':'time'})
df1
```

Out[25]:

	9	39	49	65664	-0.14	-0.043
1970-01-01 00:00:00.000000000	9	39	49	65703.0	-0.162	0.313
1970-01-01 00:00:00.000000001	9	39	49	65742.0	-0.002	-0.388
1970-01-01 00:00:00.000000002	9	39	49	65781.0	0.037	0.411
1970-01-01 00:00:00.000000003	9	39	49	65820.0	-0.085	-0.292
1970-01-01 00:00:00.000000004	9	39	49	65859.0	0.016	-0.431
...	...	...	...	...	...	...
1970-01-01 00:00:00.000002554	9	39	49	165470.0	-0.391	0.698
1970-01-01 00:00:00.000002555	9	39	49	165510.0	-0.444	0.493
1970-01-01 00:00:00.000002556	9	39	49	165550.0	-0.306	-0.279
1970-01-01 00:00:00.000002557	9	39	49	165580.0	0.082	0.174
1970-01-01 00:00:00.000002558	9	39	49	165620.0	0.669	-0.125

2559 rows × 6 columns

```
In [30]: df1.columns
```

```
Out[30]: Index(['8', '47', '5', '1.9691e+05', '0.05', '-0.253'], dtype='object')
```

```
In [31]: df2.columns
```

```
Out[31]: Index(['8', '14', '15', '8.8441e+05', '-0.391', '0.011'], dtype='object')
```

```
In [32]: df3.columns
```

```
Out[32]: Index(['9', '10', '39', '1.1879e+05', '0.338', '-0.263'], dtype='object')
```

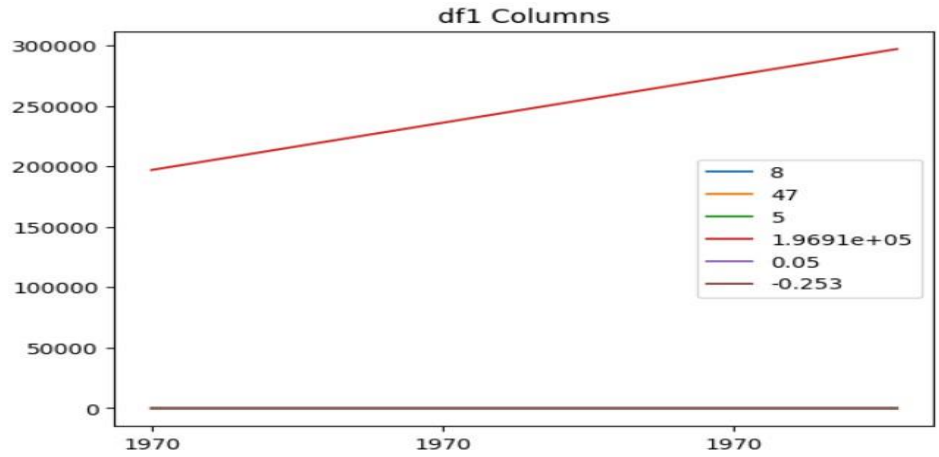
```
In [33]: df4.columns
```

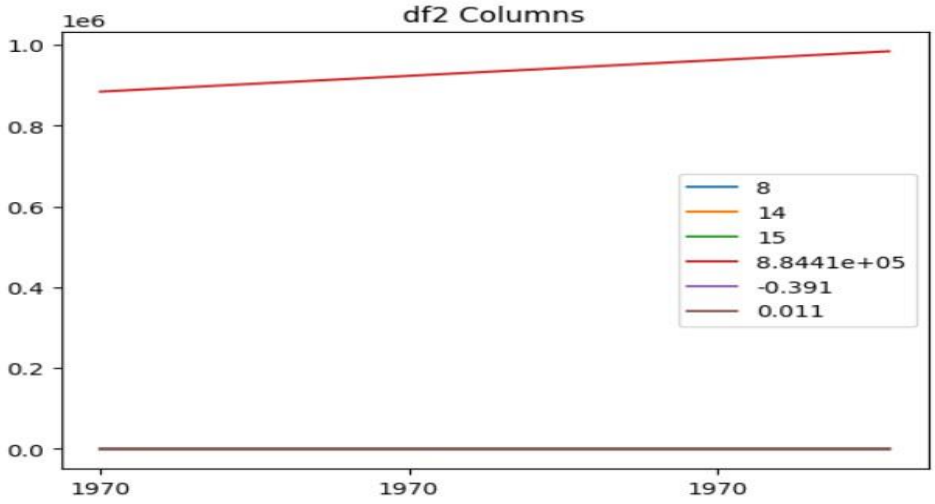
```
Out[33]: Index(['8', '34', '41', '9.7816e+05', '-0.291', '0.181'], dtype='object')
```

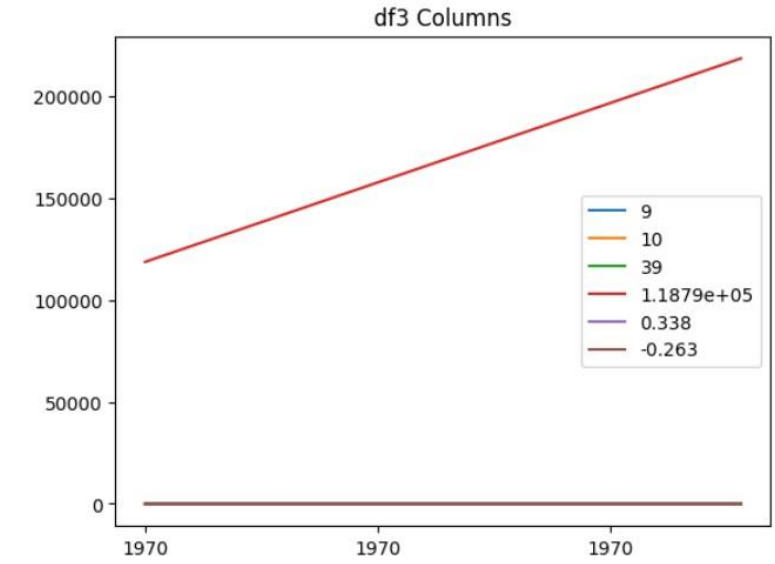
5.3 Plotting of graphs:

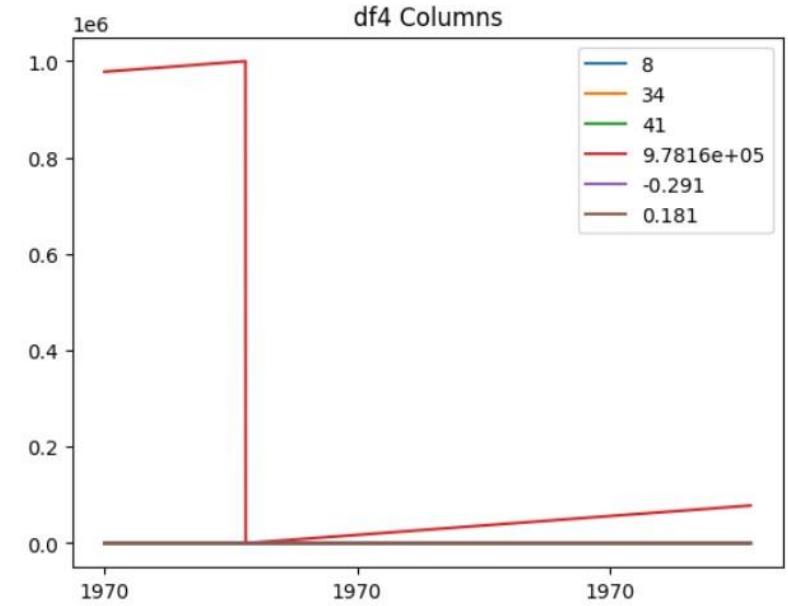
In [34]:

```
plt.figure()
for col in df1.columns:
    plt.plot(df1.index, df1[col], label=col)
plt.legend()
plt.title("df1 Columns")
plt.show()

# Plotting df2 columns
plt.figure()
for col in df2.columns:
    plt.plot(df2.index, df2[col], label=col)
plt.legend()
plt.title("df2 Columns")
plt.show()

# Plotting df3 columns
plt.figure()
for col in df3.columns:
    plt.plot(df3.index, df3[col], label=col)
plt.legend()
plt.title("df3 Columns")
plt.show()

# Plotting df4 columns
plt.figure()
for col in df4.columns:
    plt.plot(df4.index, df4[col], label=col)
plt.legend()
plt.title("df4 Columns")
plt.show()
```





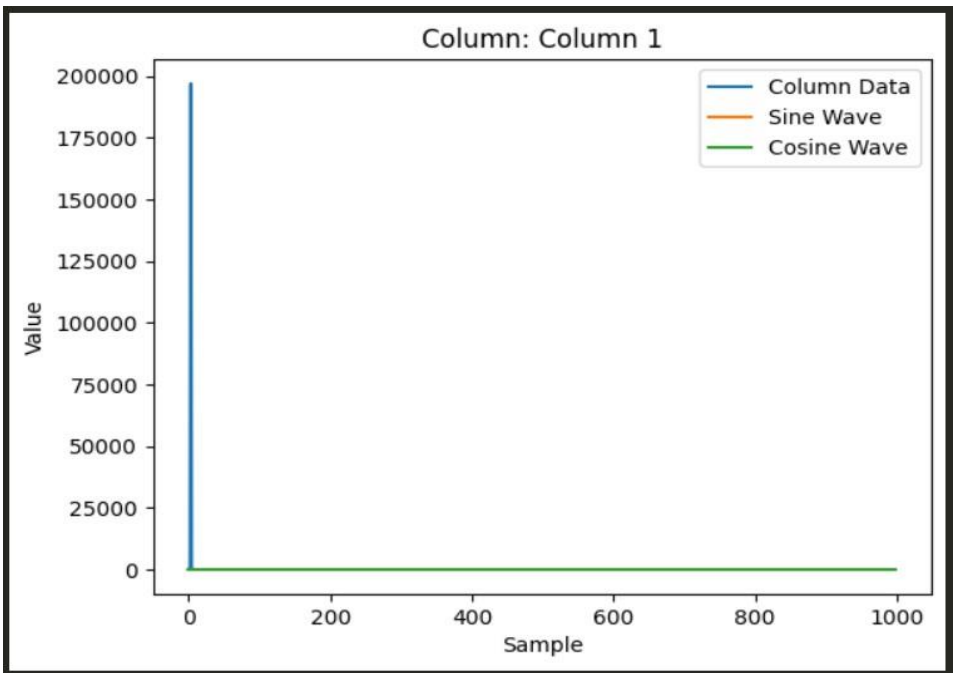


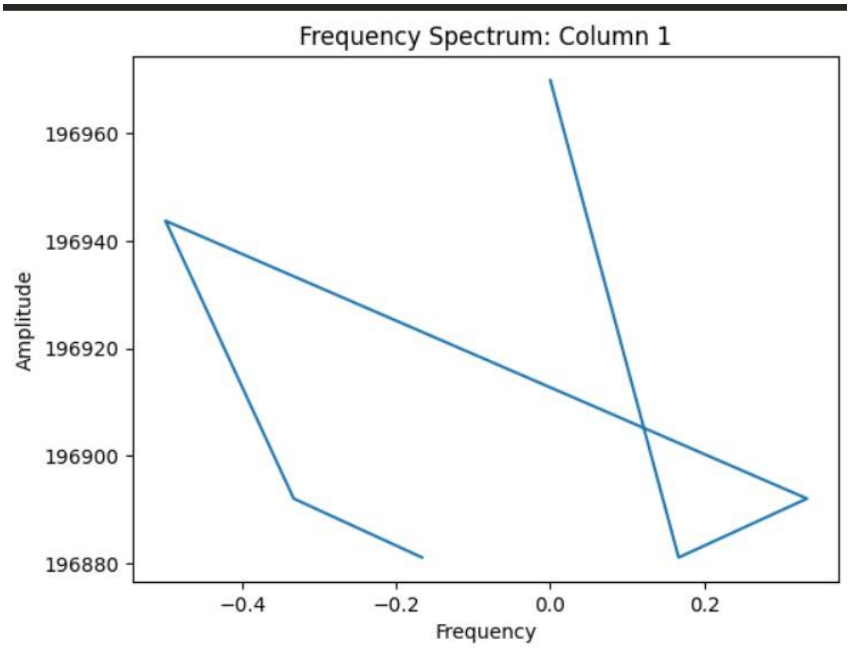


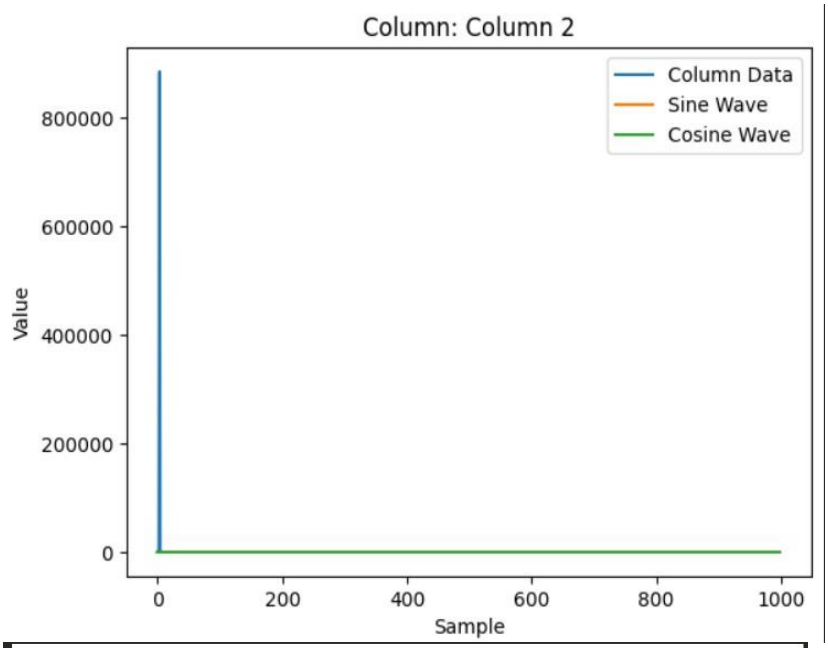
In [37]:

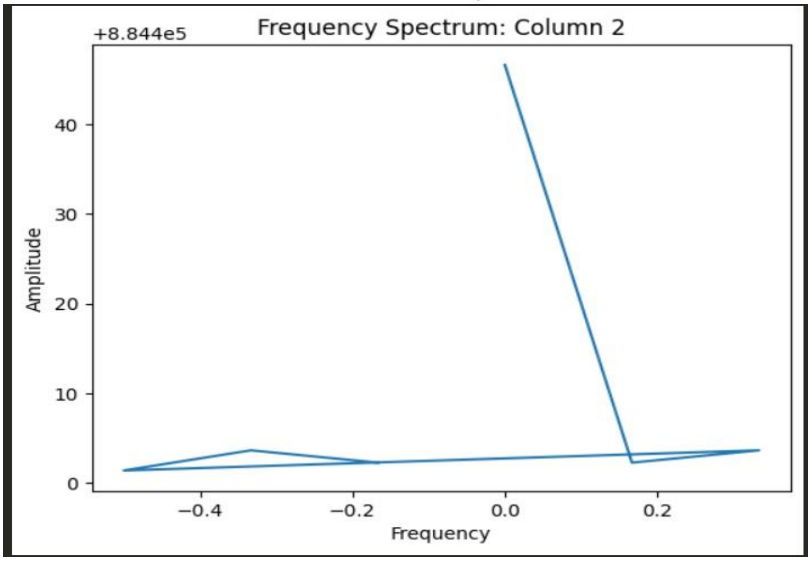
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq

# Example data
df1_columns = ['8', '47', '5', '1.9691e+05', '0.05', '-0.253']
df2_columns = ['8', '14', '15', '8.8441e+05', '-0.391', '0.011']
df3_columns = ['9', '10', '39', '1.1879e+05', '0.338', '-0.263']
df4_columns = ['8', '34', '41', '9.7816e+05', '-0.291', '0.181']

# Function to generate sine and cosine waves
def generate_wave(amplitude, frequency, phase, num_samples):
    time = np.arange(num_samples)
    return amplitude * np.sin(2 * np.pi * frequency * time + phase), amplitude * np.cos(2 * np.pi * frequency * time + phase)

# Function to analyze and plot column data with sine and cosine waves
def analyze_column(column_data, column_name):
    # Generate sine and cosine waves
    num_samples = 1000
    amplitude = 1.0
    frequency = 10.0  # Modify the frequency as desired
    phase = 0.0
    sin_wave, cos_wave = generate_wave(amplitude, frequency, phase, num_samples)

    # Convert column data to numpy array
    data = np.array(column_data, dtype=np.float32)
```

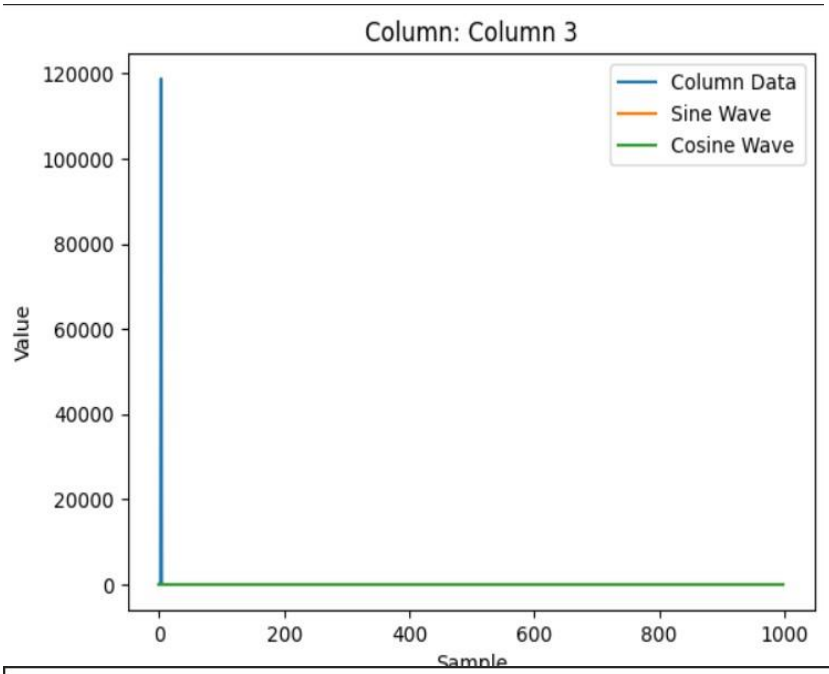
```
    # Plot the data
    plt.figure()
    plt.plot(data)
    plt.plot(sin_wave)
    plt.plot(cos_wave)
    plt.xlabel('Sample')
    plt.ylabel('Value')
    plt.title(f'Column: {column_name}')
    plt.legend(['Column Data', 'Sine Wave', 'Cosine Wave'])

    # Perform Fourier Transform to analyze frequency content
    n = len(data)
    dt = 1  # Assuming data is sampled at regular intervals
    frequencies = fftfreq(n, dt)
    fft_values = fft(data)

    # Plot the frequency spectrum
    plt.figure()
    plt.plot(frequencies, np.abs(fft_values))
    plt.xlabel('Frequency')
    plt.ylabel('Amplitude')
    plt.title(f'Frequency Spectrum: {column_name}')

    # Identify the dominant frequency
    dominant_frequency = frequencies[np.argmax(np.abs(fft_values))]
    print(f'Dominant frequency for column {column_name}: {dominant_frequency} Hz')

# Analyze and plot each column separately
analyze_column(df1_columns, 'Column 1')
analyze_column(df2_columns, 'Column 2')
analyze_column(df3_columns, 'Column 3')
analyze_column(df4_columns, 'Column 4')

# Show all the plots
plt.show()
```

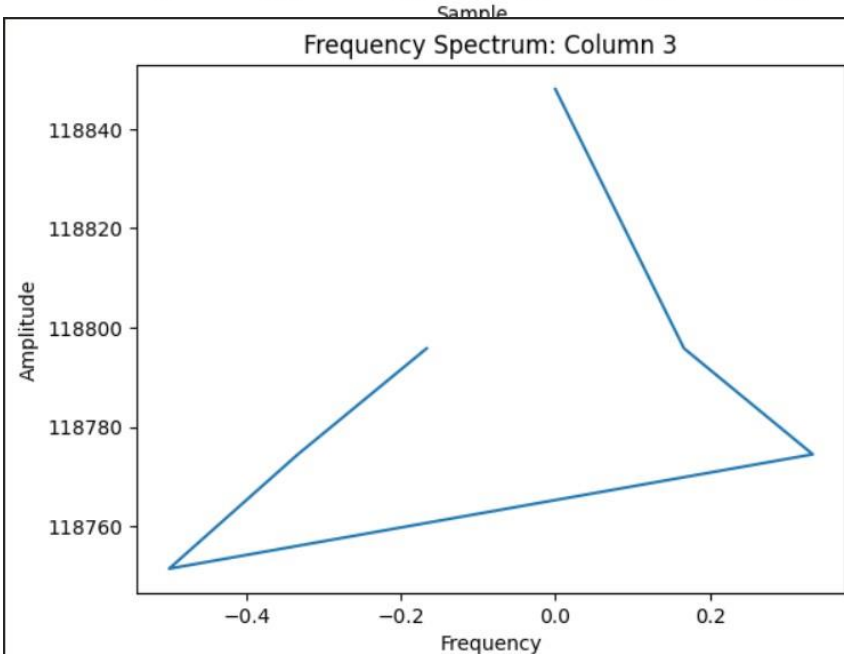


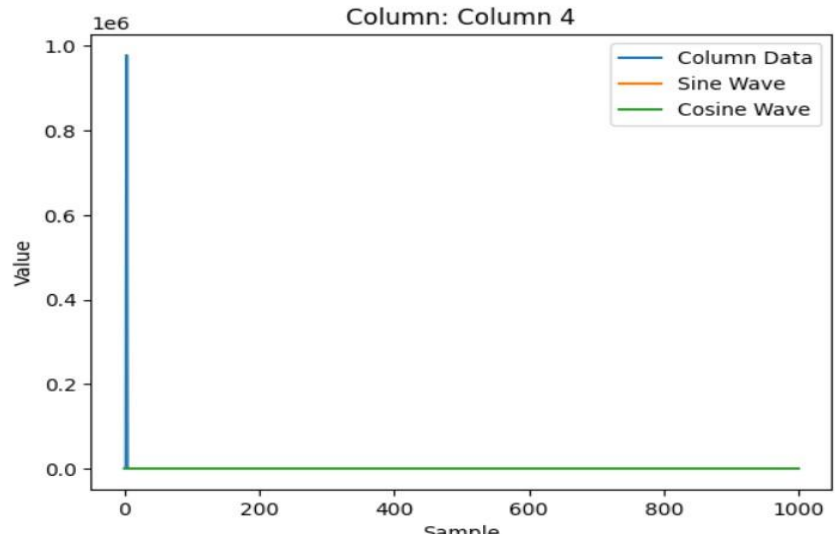

Frequency Spectrum: Column 1
196960
196940
196920
196900
196880
Amplitude
−0.4
−0.2
0.0
0.2
Frequency

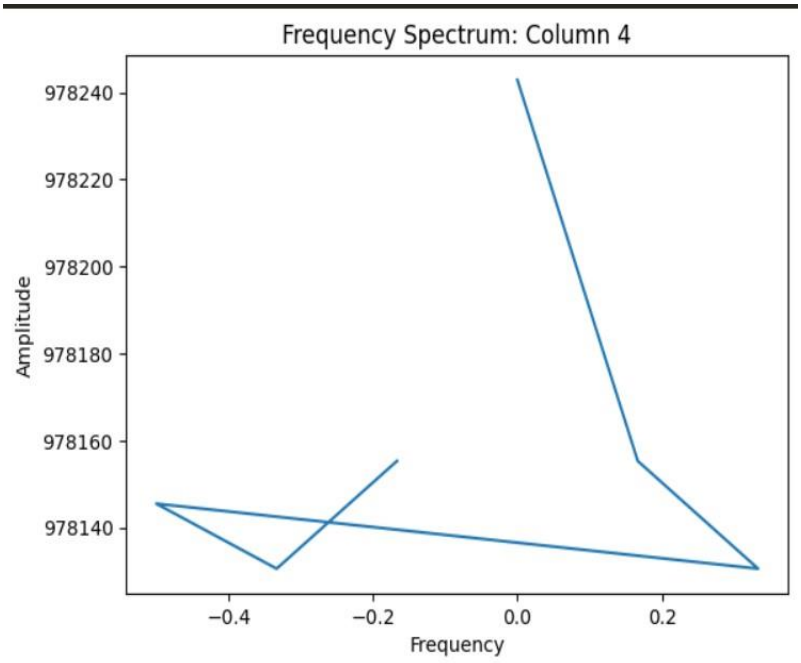

Column: Column 2
Column Data
Sine Wave
Cosine Wave
800000
600000
400000
200000
0
Value
0
200
400
600
800
1000
Sample


+8.844e5
Frequency Spectrum: Column 2
40
30
20
10
0
Amplitude
−0.4
−0.2
0.0
0.2
Frequency


Column: Column 3
Column Data
Sine Wave
Cosine Wave
Value
Sample
120000
100000
80000
60000
40000
20000
0
0
200
400
600
800
1000


Frequency Spectrum: Column 3
Amplitude
Frequency
118840
118820
118800
118780
118760
−0.4
−0.2
0.0
0.2


Column: Column 4
1e6
Column Data
Sine Wave
Cosine Wave
Value
Sample
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000



5.4 RUL Prediction:

In [43]:
```
import pandas as pd
import numpy as np
import xgboost as xgb
from sklearn.metrics import mean_squared_error
from math import sqrt

# Load the dataset
train_data = pd.read_csv('C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00002.csv')
test_data = pd.read_csv('C:/Users/hp-d/Desktop/notebook/Test_set/Bearing1_3/acc_00001.csv')

# Separate the features and target variable
X_train = train_data.iloc[:, :-1].values
y_train = train_data.iloc[:, -1].values
X_test = test_data.iloc[:, :-1].values
y_test = test_data.iloc[:, -1].values

# Train the XGBoost model
xgbr = xgb.XGBRegressor(objective='reg:squarederror', random_state=42)
xgbr.fit(X_train, y_train)

# Make predictions on the test set
y_pred = xgbr.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
rmse = sqrt(mse)
print("Root Mean Squared Error:", rmse)
```

```
Root Mean Squared Error: 0.3818637594067631
```

In [54]:
```
set1 = pd.read_csv("C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00002.csv")
set1 = set1.rename(columns={'Unnamed: 0':'time'})
#set1 = set1.set_index('time')
last_cycle = int(len(set1))
```

In [55]:
```
def health_indicator(bearing_data,use_filter=False):
    data = bearing_data.copy()
    if use_filter:
        for ft in data.columns:
            data[ft] = data[ft].ewm(span=40,adjust=False).mean()
    pca = PCA()
    X_pca = pca.fit_transform(data)
    component_names = [f"PC{i+1}" for i in range(X_pca.shape[1])]
    X_pca = pd.DataFrame(X_pca, columns=component_names)
    print("Explained variance of Pricincipal Component 1 is:"+str(pca.explained_variance_ratio_[0]))
    health_indicator = np.array(X_pca['PC1'])
    degredation = pd.DataFrame(health_indicator,columns=['PC1'])
    degredation['cycle'] = degredation.index
    degredation['PC1'] = degredation['PC1']-degredation['PC1'].min(axis=0)

    return degredation

def fit_exp(df,base=500,print_parameters=False):
    x =np.array(df.cycle)
    x = x[-base:].copy()
    y = np.array(degredation.PC1)
    y = y[-base:].copy()
    def exp_fit(x,a,b):
        y = a*np.exp(abs(b)*x)
        return y
    #initial parameters affect the result
    fit = curve_fit(exp_fit,x,y,p0=[0.01,0.001],maxfev=10000)
    if print_parameters:
        print(fit)
    return fit

def predict(X_df,p):
    x =np.array(X_df.cycle)
    a,b = p[0]
    fit_eq = a*np.exp(abs(b)*x)
    return fit_eq
log = [[],[]]
```

In [56]:
```
prediction_cycle = 600
#variable for keeping intial value
init_cycle = prediction_cycle
```

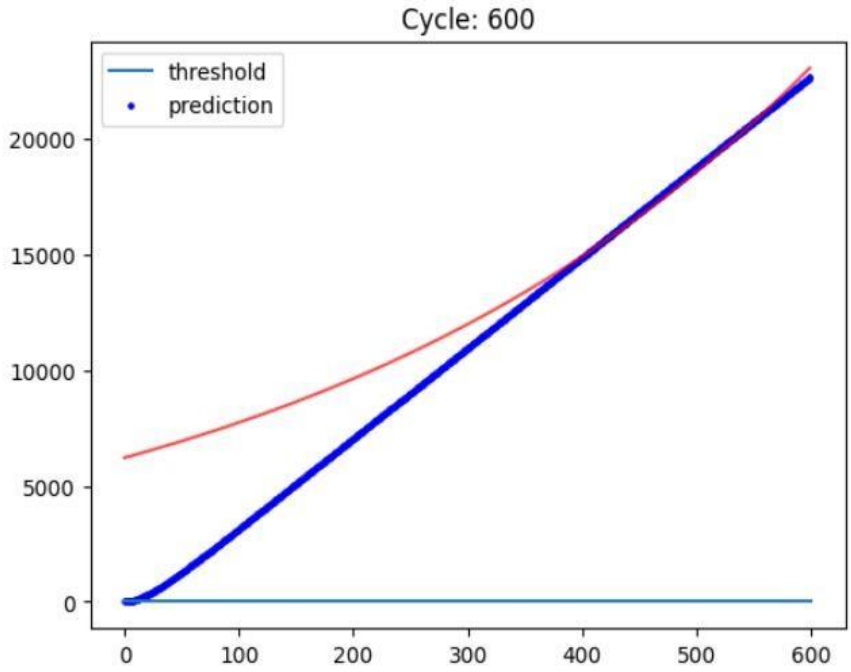
In [62]:
```
selected_features = ['Max','Mean','RMS']

bearing = 3
B_x = ["df".format(bearing)+i for i in selected_features]
early_cycles = set1[:init_cycle]
early_cycles_pca = health_indicator(early_cycles,use_filter=True)
```

```
Explained variance of Pricincipal Component 1 is:0.9999999998341101
```

In [63]:

```
data = set1[:prediction_cycle]
ind=data.index
degredation = health_indicator(data,use_filter=True)
#degredation.plot(y='PC1',x='cycle')
fit = fit_exp(degredation,base=250)

prediction = predict(degredation,fit)
m,n = fit[0]
thres = 2
#print(prediction_cycle)
fail_cycle = (np.log(thres/m))/abs(n)
log[0].append(prediction_cycle)
log[1].append(fail_cycle)

print(m,n)
print('failed at'+str(fail_cycle))

fig =plt.figure('Cycle: '+str(prediction_cycle))
ax =fig.subplots()

ax.plot([0,prediction_cycle],[2,2])
ax.scatter(degredation['cycle'],degredation['PC1'],color='b',s=5)
ax.plot(degredation['cycle'],prediction,color='r',alpha=0.7)
ax.set_title('Cycle: '+str(prediction_cycle))
ax.legend(['threshold','prediction'])
fig.savefig('output.png')
plt.show()
increment_cycle =25
prediction_cycle += increment_cycle
```

```
Explained variance of Pricincipal Component 1 is:0.9999999998341101
6220.2071729703675 0.0021866839477168924
failed at-3677.9029363466766
```



In [72]:
```
set2 = pd.read_csv("C:/Users/hp-d/Desktop/notebook/Test_set/Bearing1_3/acc_00001.csv")
set2 = set2.rename(columns={'Unnamed: 0':'time'})
set2.head()
```

Out[72]:

	8	33	1	3.7816e+05	0.092	0.044
0	8	33	1	378200.0	-0.025	0.432
1	8	33	1	378240.0	-0.104	0.008
2	8	33	1	378280.0	0.056	-0.264
3	8	33	1	378320.0	0.074	-0.195
4	8	33	1	378360.0	-0.147	-0.373

In [73]:
```
log = [[],[]]
#variable for incrementing index
prediction_cycle = 550
#variable for keeping intial value
init_cycle = prediction_cycle
```

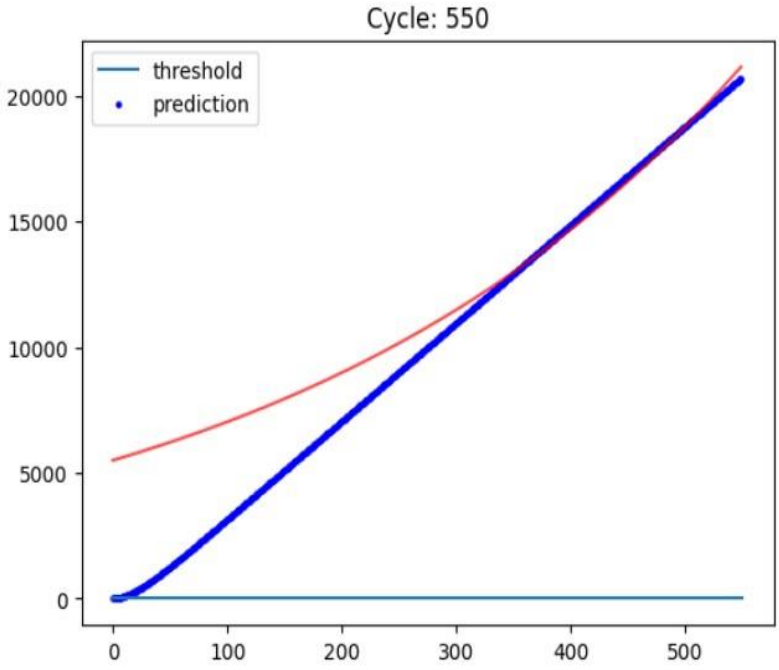
In [74]:
```
selected_features = ['Max','Mean']
bearing = 1
B_x = ["df".format(bearing)+i for i in selected_features]
early_cycles = set2[:init_cycle]
early_cycles_pca = health_indicator(early_cycles,use_filter=True)
```

```
Explained variance of Pricincipal Component 1 is:0.9999999998055813
```

In [75]:
```
data = set2[:prediction_cycle]
degredation = health_indicator(data,use_filter=True)
fit = fit_exp(degredation,base=250)

prediction = predict(degredation,fit)
m,n = fit[0]
thres = 2
fail_cycle = (np.log(thres/m))/abs(n)
log[0].append(prediction_cycle)
log[1].append(fail_cycle)

#print(m,n)
print('failed at'+str(fail_cycle))

fig =plt.figure()
ax =fig.subplots()
ax.plot([0,prediction_cycle],[2,2])
ax.set_title('Cycle: '+str(prediction_cycle))
ax.scatter(degredation['cycle'],degredation['PC1'],color='b',s=5)
ax.plot(degredation['cycle'],prediction,color='r',alpha=0.7)
ax.legend(['threshold','prediction'])
plt.show()
increment_cycle = 25
prediction_cycle += increment_cycle
```

```
Explained variance of Pricincipal Component 1 is:0.9999999998055813
failed at-3228.6081682785425
```

```
Explained variance of Pricincipal Component 1 is:0.9999999998055813
failed at-3228.6081682785425
```



In [42]:

```python
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers

# Load the dataset (replace with your own preprocessing steps)
x_train = pd.read_csv('C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00002.csv')
y_train = pd.read_csv('C:/Users/hp-d/Desktop/notebook/Learning_set/Bearing1_1/acc_00003.csv')
x_test = pd.read_csv('C:/Users/hp-d/Desktop/notebook/Test_set/Bearing1_3/acc_00002.csv')
y_test = pd.read_csv('C:/Users/hp-d/Desktop/notebook/Test_set/Bearing1_3/acc_00003.csv')


# Convert DataFrame to numpy array
x_train = x_train.values
y_train = y_train.values
x_test = x_test.values
y_test = y_test.values

# Get the shape of the input data
num_samples, num_time_steps = x_train.shape[0], x_train.shape[1]  # Number of samples and time steps
num_features = 1  # Assuming there is a single feature column

# Reshape the input data
x_train = x_train.reshape(num_samples, num_time_steps, num_features)
x_test = x_test.reshape(x_test.shape[0], num_time_steps, num_features)

# Build the CNN model
model = tf.keras.Sequential()
model.add(layers.Conv1D(64, kernel_size=3, activation='relu', input_shape=(num_time_steps, num_features)))
model.add(layers.MaxPooling1D(pool_size=2))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1))
```

```python
# Compile and train the CNN model
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the model
test_loss = model.evaluate(x_test, y_test)
print(f'Test loss: {test_loss}')

# Make predictions
predictions = model.predict(x_test)
```

```
Epoch 1/10
80/80 [==============================] - 2s 8ms/step - loss: 1997298560.0000 - val_loss: 25575911424.0000
Epoch 2/10
80/80 [==============================] - 0s 4ms/step - loss: 1973026688.0000 - val_loss: 25575835648.0000
Epoch 3/10
80/80 [==============================] - 0s 4ms/step - loss: 1973024000.0000 - val_loss: 25575835648.0000
Epoch 4/10
80/80 [==============================] - 0s 5ms/step - loss: 1973023744.0000 - val_loss: 25575839744.0000
Epoch 5/10
80/80 [==============================] - 0s 4ms/step - loss: 1973024256.0000 - val_loss: 25575835648.0000
Epoch 6/10
80/80 [==============================] - 0s 4ms/step - loss: 1973023744.0000 - val_loss: 25575835648.0000
Epoch 7/10
80/80 [==============================] - 0s 4ms/step - loss: 1973024256.0000 - val_loss: 25575835648.0000
Epoch 8/10
80/80 [==============================] - 0s 4ms/step - loss: 1973024256.0000 - val_loss: 25575835648.0000
Epoch 9/10
80/80 [==============================] - 0s 5ms/step - loss: 1973023744.0000 - val_loss: 25575835648.0000
Epoch 10/10
80/80 [==============================] - 0s 5ms/step - loss: 1973023744.0000 - val_loss: 25575835648.0000
80/80 [==============================] - 0s 2ms/step - loss: 25575835648.0000
Test loss: 25575835648.0
80/80 [==============================] - 0s 2ms/step
```

6. Final Deliverable

Remaining Useful Life Prediction using bearing dataset

	0	1	2	3	4	5	6	7	8
0	5.0000	1.0000	3.0000	1.5811	3.3166	0.0000	-1.9120	1.5076	1.1055

	Max	Min	Mean	Std	RMS	Skewness	Kurtosis	Crest Factor	Form Factor
0	1.725	-1.915	0.0056	0.5352	0.5351	-0.0258	-0.0869	3.2236	95.3961

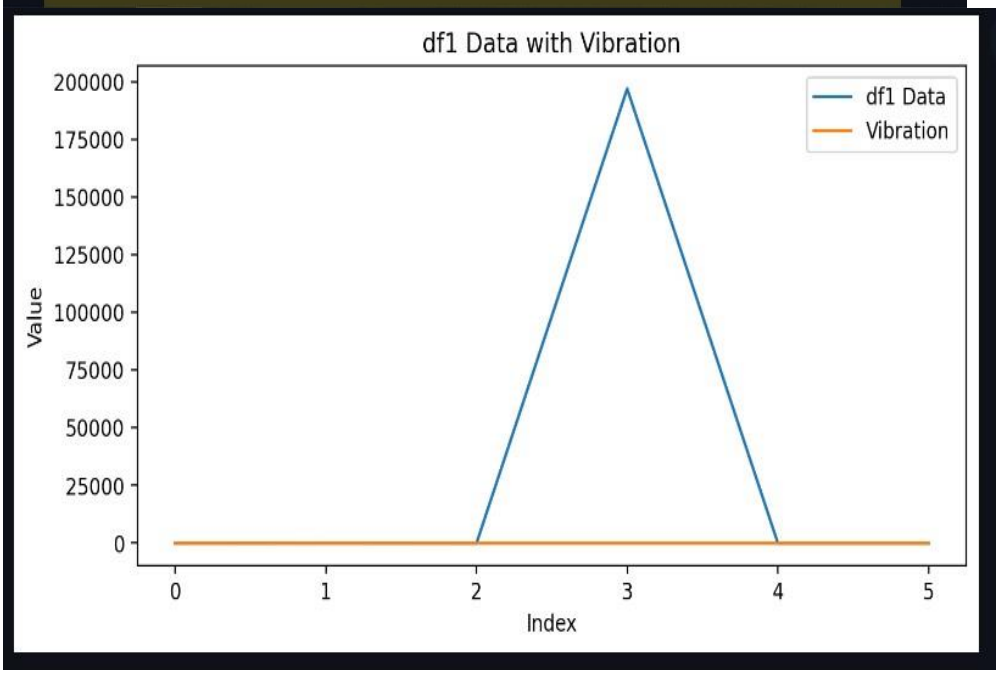
PyplotGlobalUseWarning: You are calling `st.pyplot()` without any arguments. After December 1st, 2020, we will remove the ability to do this as it requires the use of Matplotlib's global figure object, which is not thread-safe.

To future-proof this code, you should pass in a figure as shown below:

```
>>> fig, ax = plt.subplots()
>>> ax.scatter([1, 2, 3], [1, 2, 3])
>>>     ... other plotting actions ...
>>> st.pyplot(fig)
```

You can disable this warning by disabling the config option: `deprecation.showPyplotGlobalUse`

```
st.set_option('deprecation.showPyplotGlobalUse', False)
```



PyplotGlobalUseWarning: You are calling `st.pyplot()` without any arguments. After December 1st, 2020, we will remove the ability to do this as it requires the use of Matplotlib's global figure object, which is not thread-safe.

To future-proof this code, you should pass in a figure as shown below:

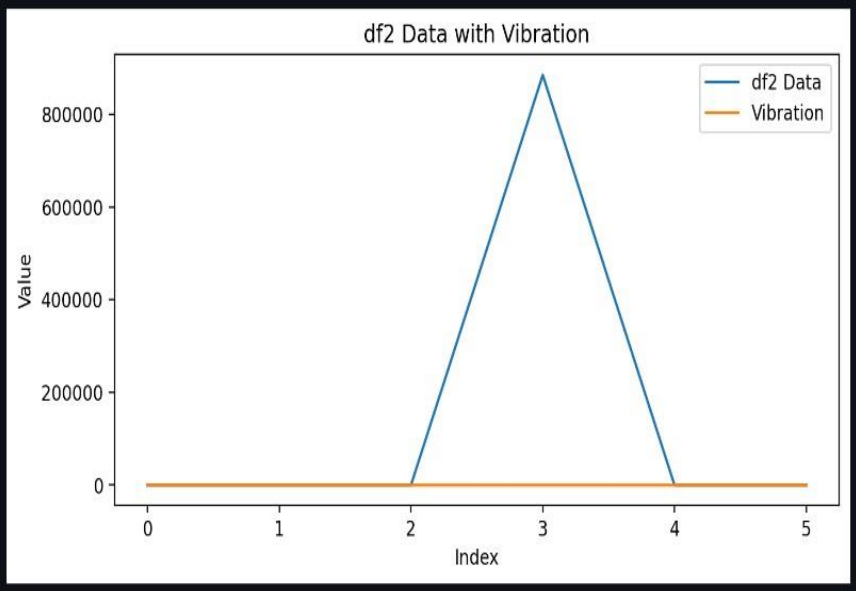
```
>>> fig, ax = plt.subplots()
>>> ax.scatter([1, 2, 3], [1, 2, 3])
>>>     ... other plotting actions ...
>>> st.pyplot(fig)
```

You can disable this warning by disabling the config option: `deprecation.showPyplotGlobalUse`

```
st.set_option('deprecation.showPyplotGlobalUse', False)
```

or in your `.streamlit/config.toml`

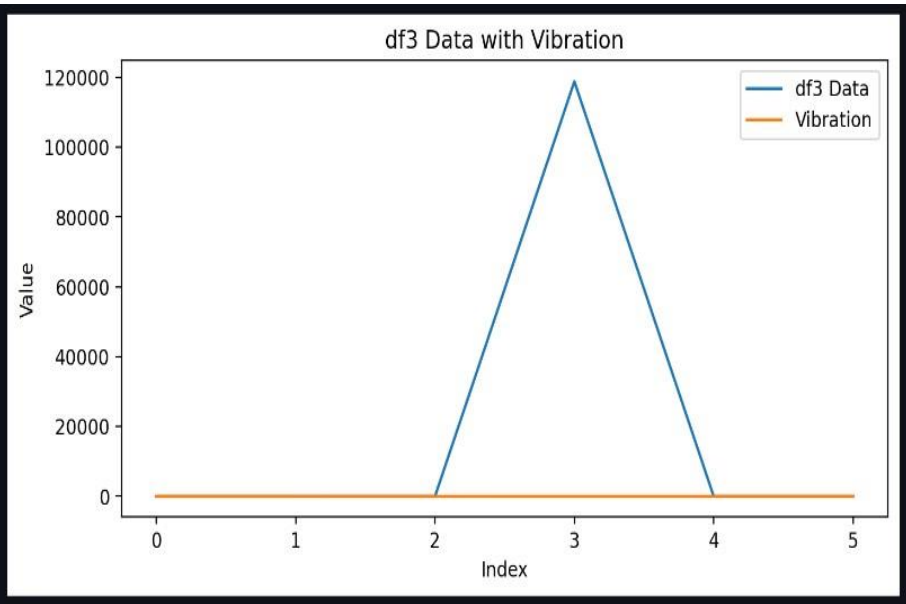
```
[deprecation]
showPyplotGlobalUse = false
```



PyplotGlobalUseWarning: You are calling `st.pyplot()` without any arguments. After December 1st, 2020, we will remove the ability to do this as it requires the use of Matplotlib's global figure object, which is not thread-safe.

To future-proof this code, you should pass in a figure as shown below:

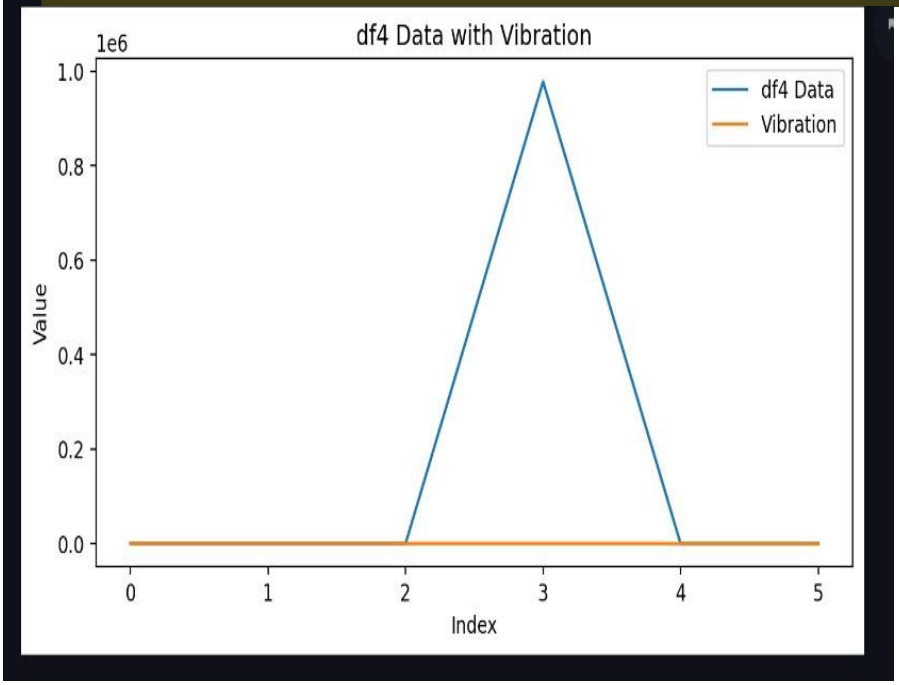
```
>>> fig, ax = plt.subplots()
>>> ax.scatter([1, 2, 3], [1, 2, 3])
>>>     ... other plotting actions ...
>>> st.pyplot(fig)
```

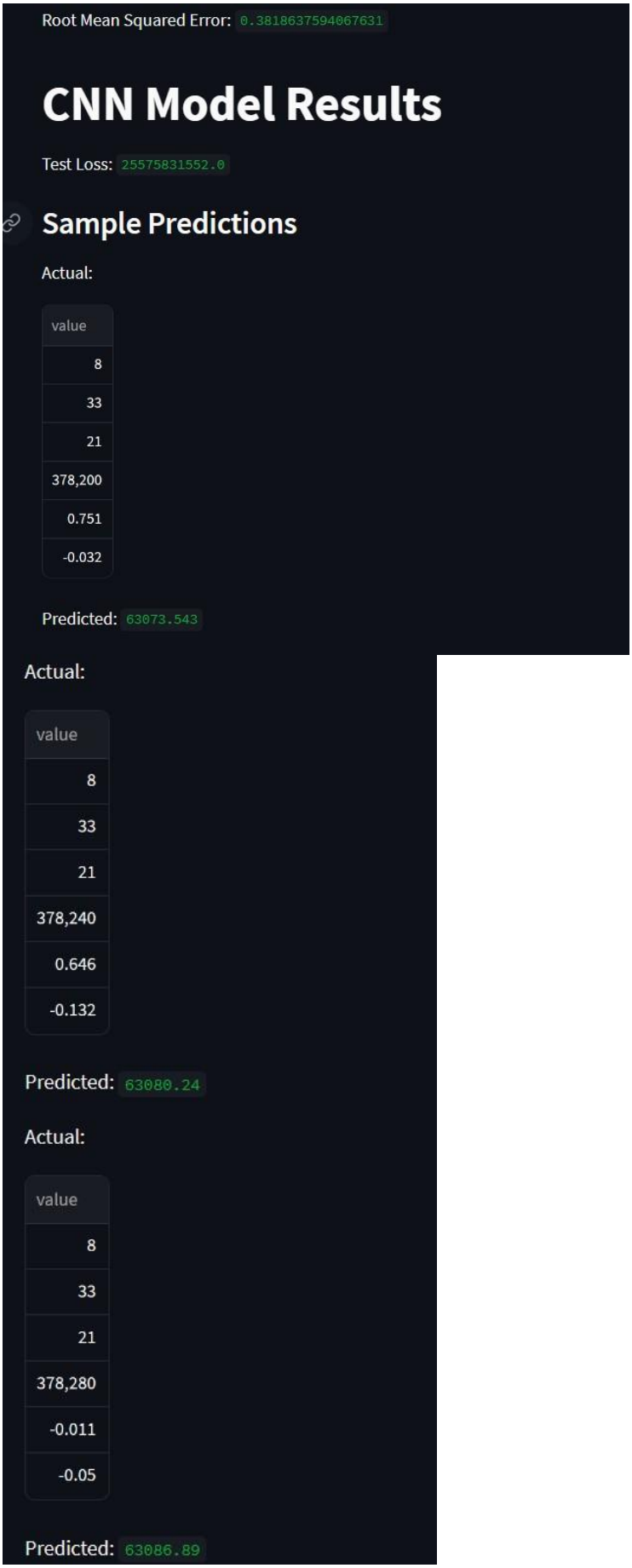


PyplotGlobalUseWarning: You are calling `st.pyplot()` without any arguments. After December 1st, 2020, we will remove the ability to do this as it requires the use of Matplotlib's global figure object, which is not thread-safe.

To future-proof this code, you should pass in a figure as shown below:

```
>>> fig, ax = plt.subplots()
>>> ax.scatter([1, 2, 3], [1, 2, 3])
>>>     ... other plotting actions ...
>>> st.pyplot(fig)
```




Root Mean Squared Error: 0.3818637594067631
CNN Model Results
Test Loss: 25575831552.0
Sample Predictions
Actual:
value
8
33
21
378,200
0.751
-0.032
Predicted: 63073.543
Actual:
value
8
33
21
378,240
0.646
-0.132
Predicted: 63080.24
Actual:
value
8
33
21
378,280
-0.011
-0.05
Predicted: 63086.89

Actual:

value
8
33
21
378,320
0.15
-0.212

Predicted: `63093.586`

Actual:

value
8
33
21
378,360
-0.185
0.08

Predicted: `63100.242`

7. Innovation in Implementation

7.1 Customized Model Selection

Customized Model Selection refers to the implementation of a feature within your application that allows users to choose different machine learning models for performing a specific task. In the context of your remaining useful life prediction project using Streamlit, this means giving users the ability to select and compare different models to see which one performs best for their dataset. Here's a brief explanation:

User Choice: Users can select from a list of available machine learning models, such as Convolutional Neural Networks (CNN), Long Short-Term Memory (LSTM), XGBoost, Random Forest, etc.

Interactive Interface: Create a user-friendly interface using Streamlit where users can use dropdown menus or radio buttons to select the desired model.

Real-time Comparison: Once a model is selected, the application will train and evaluate the chosen model using your dataset. The results can be displayed in real-time, allowing users to compare the performance metrics (such as Mean Squared Error) of different models.

Visualization: Display the results using visualizations (graphs, tables, etc.) within the Streamlit app to help users easily understand and compare the outcomes of different models.

7.2 Feature Importance Visualization

Feature Importance Visualization is a way to understand which features or variables have the most significant influence on a model's predictions. It helps in identifying which factors contribute the most to the model's decision-making process. This visualization is particularly useful for interpretability, model understanding, and potentially for feature selection or engineering.

Calculation: Feature importance is usually calculated after a model is trained. Different algorithms have their own methods for calculating feature importance. For example, decision tree-based algorithms (like Random Forest or XGBoost) use metrics such as Gini impurity or gain to assess feature importance.

Ranking: Features are ranked based on their importance scores. The higher the score, the more influential the feature is in making predictions.

Visualization: The feature importance scores are plotted in a visual format. Common visualization methods include bar charts, horizontal bar plots, heatmaps, and scatter plots.

Interpretation: Users can interpret the visualization to understand which features have the most impact on the model's output. Features with higher importance contribute more to the prediction, and those with lower importance contribute less.

Application: Feature importance visualization can be used to gain insights into the underlying patterns in the data, validate domain knowledge, identify potential issues with the model, and even guide further data collection or refinement efforts.

7.3 Real-time Data Collection and Prediction

Real-time data collection and prediction refer to the process of continuously gathering data from sensors or sources as they become available and using that data to make immediate predictions or decisions. In the context of remaining useful life prediction, this concept involves continuously monitoring the health and performance of a system or component and providing real-time estimates of how much useful life remains before a potential failure occurs.

Data Collection: Sensors or devices are installed to collect data from the system or component being monitored. These sensors could measure various parameters such as temperature, vibration, pressure, or any other relevant indicators.

Continuous Monitoring: The collected data is continuously monitored and streamed to a central system or application in real-time. As new data points arrive, they are processed and incorporated into the analysis.

Feature Extraction: Relevant features are extracted from the incoming data. These features could include patterns, trends, or anomalies that indicate the health and performance of the system.

Model Update: The predictive model, such as a machine learning model, is periodically updated with the latest data. This helps the model adapt to changing conditions and ensures its predictions remain accurate over time.

Real-time Prediction: As new data points arrive, they are fed into the updated model to make real-time predictions about the remaining useful life of the system. These predictions provide immediate insights into how long the system is expected to function before a potential failure.

7.4 Anomaly Detection and Alerts

Anomaly detection is a technique used to identify patterns or instances in data that significantly deviate from the expected or normal behavior. In the context of the given code for remaining useful life prediction, anomaly detection can be applied to identify unusual or abnormal predictions of remaining useful life for equipment or components.

Anomaly Detection:
Anomaly detection involves comparing predicted remaining useful life values with the actual remaining useful life values. If a predicted value deviates significantly from the actual value, it may indicate a potential anomaly.

For instance, if the model predicts a very short remaining useful life while the actual remaining useful life is much longer, this could suggest a potential issue or malfunction that requires attention.

Threshold-based Detection:
One common approach is to set a threshold or tolerance level beyond which predictions are considered anomalous. If the predicted remaining useful life falls outside this threshold, it triggers an anomaly alert.

Visualization:
Anomalies can be visually represented on a plot alongside the actual and predicted remaining useful life values. Anomalies can be highlighted using different colors or symbols to make them easily distinguishable.

Alerts and Notifications:
When an anomaly is detected, the Streamlit app can generate alerts or notifications to inform relevant stakeholders or users. These alerts can be in the form of pop-up messages, emails, or SMS notifications.

User Interaction:
Users of the Streamlit app can interact with the anomaly detection by adjusting the threshold or sensitivity of the anomaly detection algorithm. This allows users to customize the level of sensitivity to anomalies.

7.5 Interactive Model Explanation

Interactive Model Explanation refers to the ability to provide insights and explanations about how a machine learning model arrives at its predictions in an interactive and user-friendly manner. It helps users, such as data scientists, analysts, or end-users, understand the factors that influence the model's decisions and builds trust in the model's outcomes. Here's a concise explanation:

Model Interpretability: Interactive Model Explanation involves making complex machine learning models more interpretable by breaking down their predictions into understandable components. This helps users grasp the model's behavior and decision-making process.

Feature Contributions: The explanation highlights the contribution of each input feature (e.g., variables or attributes) to the model's predictions. Users can see which features have the most significant impact on the output.

User Interaction: The explanation is presented in an interactive manner, often using visualizations and charts. Users can click on data points or features to see how changes in those inputs affect the model's predictions.

Methods: Techniques like SHAP (SHapley Additive exPlanations), LIME (Local Interpretable Model-agnostic Explanations), or feature importance plots are commonly used for interactive model explanation.

8. Scalability to Solve Industrial Problem

8.1 Big Data Handling

Handling big data is a critical aspect of industrial applications, especially when dealing with predictive maintenance and remaining useful life (RUL) prediction. Here are some strategies for effectively handling big data in your RUL project:

Data Partitioning and Parallel Processing:
Break down large datasets into smaller partitions and process them in parallel. This approach improves processing speed and resource utilization. Technologies like Apache Spark enable distributed processing for tasks like feature engineering and model training.

Distributed Computing Frameworks:
Utilize distributed computing frameworks like Apache Hadoop and Apache Spark for processing and analyzing large datasets. These frameworks allow you to distribute data across a cluster of machines, improving scalability and performance.

Cloud Computing Platforms:
Leverage cloud platforms such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP). Cloud services offer scalable storage, computing power, and data processing capabilities, enabling you to handle big data efficiently.

Data Compression and Serialization:
Use data compression techniques to reduce storage requirements and speed up data transfer. Serialization formats like Parquet and ORC are optimized for query performance and storage efficiency.

Data Sampling and Subsetting:
When exploring and analyzing big data, consider using data sampling or subsetting to work with a representative portion of the dataset. This helps reduce computation time and allows for quicker experimentation.

Databases and Data Warehouses:
Implement databases (e.g., SQL databases, NoSQL databases) and data warehouses to store and manage big data. Indexing and optimization techniques enhance query performance.

8.2 Distributed Machine Learning

Distributed Machine Learning (DML) is an approach that involves training machine learning models across multiple computing nodes or machines. It is particularly useful for handling large datasets and complex models that may require significant computational resources. DML allows you to distribute the training process, enabling faster model development and improved scalability. Here's an overview of distributed machine learning and how it can be applied to your remaining useful life (RUL) prediction project:

Parallel Processing:
In DML, data and computation are divided into smaller tasks that can be executed simultaneously on different machines. This parallel processing speeds up model training, especially for large datasets.

Distributed Computing Frameworks:
Distributed machine learning relies on frameworks that manage the distribution of data and computations. Popular frameworks include TensorFlow, PyTorch, Apache Spark MLlib, and scikit-learn's dask-ml.

Data Parallelism vs. Model Parallelism:
Data Parallelism: In this approach, each worker node receives a subset of the data and updates the model parameters independently. This is useful when the dataset is too large to fit in memory on a single machine.

Model Parallelism:
Here, different parts of a complex model are trained on different machines. This is suitable for models with multiple components, such as ensemble models.

Data Partitioning:
Data is divided into partitions and distributed across nodes. This can be based on samples, features, or other criteria. Careful partitioning is essential to ensure that each node receives representative data.

8.3 Batch Processing and Streaming

Batch processing and streaming are two fundamental approaches for handling data in realtime and near-real-time scenarios. Both have their advantages and use cases, and understanding their differences can help you design an effective solution for your remaining useful life (RUL) prediction project.

Batch Processing:
Batch processing involves collecting and processing data in discrete chunks or batches. Here's how it works:

Data Collection: Data is collected over a specific period or until a certain threshold is reached. Once the batch is complete, it is sent for processing.

Processing: The entire batch of data is processed together. This allows for complex computations and analysis that might not be feasible in real-time processing.

Advantages:
- Suitable for historical analysis and reporting.
- Allows for resource-intensive processing and modeling.

- Easier to implement and manage, especially for complex operations.

Use Cases in RUL Prediction:

- Analyzing large historical datasets to train machine learning models.
- Generating periodic RUL predictions based on aggregated data.

Streaming:
Streaming involves processing data as it arrives in real-time. Here's how it works:

Continuous Data Flow: Data is processed as soon as it becomes available, often in small chunks or events. This allows for quick insights and timely actions.

Low Latency: Streaming processing offers low latency, making it suitable for applications that require immediate responses to incoming data.

Advantages:

- Enables real-time monitoring and alerting.
- Well-suited for applications requiring quick decisions and actions.
- Supports adaptive and dynamic models that can learn from new data.

Use Cases in RUL Prediction:

- Monitoring equipment health in real-time and triggering maintenance alerts.
- Continuous RUL prediction based on incoming sensor data.

Hybrid Approaches:
In many cases, a hybrid approach combining batch processing and streaming is optimal. For example:

Batch Processing for Training: You might use batch processing to train your RUL prediction model on historical data. This allows for thorough feature engineering and model evaluation.

- Streaming for Real-Time Prediction: Once the model is trained, you can deploy it to a streaming pipeline for real-time predictions as new data arrives.

Triggered Batch Processing: Periodically retrain the model using batch processing, triggered by specific events or changes in the data distribution.

Use Cases in RUL Prediction:
Use batch processing to train and fine-tune your model using historical data.
Implement a streaming pipeline to continuously monitor sensor data, trigger alerts, and provide real-time RUL predictions.
Combine batch processing and streaming to provide both historical insights and real-time predictions.

9. References

- T. Zhang, S. Liu, Y. Wei, and H. Zhang, “A novel feature adaptive extraction method based on deep learning for bearing fault diagnosis,” Measurement, vol. 185, Article ID 110030, 2021.

- B. Yang, R. Liu, and E. Zio, “Remaining useful life prediction based on a doubleconvolutional neural network architecture,” IEEE Transactions on Industrial Electronics, vol. 66, no. 12, pp. 9521–9530, 2019.

- Z. Chen, S. Cao, and Z. Mao, “Remaining useful life estimation of aircraft engines using a modifed similarity and supporting vector machine (SVM) approach,” Energies, vol. 11, no. 1, p. 28, 2017.

- https://www.hindawi.com/journals/sv/2023/3742912/

- https://in.mathworks.com/company/newsletters/articles/three-ways-to-estimateremaining-useful-life-for-predictive-maintenance.html

- G. S. Babu, P. Zhao, and X. Li, “Deep convolutional neural network based regression approach for estimation of remaining useful life,” Lecture Notes in Computer Science, vol. 9642, pp. 214–228, 2016.

- Y. Chen, G. Peng, Z. Zhu, and S. Li, “A novel deep learning method based on attention mechanism for bearing remaining useful life prediction,” Applied Soft Computing, vol. 86, Article ID 105919, 2020.

- L. Zheng, Y. Xiang, and C. Sheng, “Optimization-based improved kernel extreme learning machine for rolling bearing fault diagnosis,” Journal of the Brazilian Society of Mechanical Sciences and Engineering, vol. 41, no. 11, p. 505, 2019

- Y. Qin, S. Xiang, Y. Chai, and H. Chen, “Macroscopic–microscopic attention in LSTM networks based on fusion features for gear remaining life prediction,” IEEE Transactions on Industrial Electronics, vol. 67, no. 12, pp. 10865–10875, 2020