Capstone 3: Predicting Shaft Unbalance using Vibration Data

Table of contents

Abstract

The use of machinery vibration and the technological advances that have been developed over the years, that make it possible to not only detect when a machine is developing a problem, but to identify the specific nature of the problem for scheduled correction.
Fault detection at rotating machinery with the help of vibration sensors offers the possibility to detect damage to machines at an early stage and to prevent production down-times by taking appropriate measures.
The following Kaggle dataset consists of pairs of training and test data for different levels of vibration imbalances created by varying amounts of unbalance masses (unbalance load) on the shaft. The output of vibration sensors corresponding to the various unbalances are provided.

Problem Statement:

How can vibration data be used to predict possible damage to rotating shaft and machinery?

About the Data

A shaft attached to a motor was setup to rotate in a bearing. The other end of the shaft was connected to a load platform. The bearing had three PCB Synotech GmbH, type PCB-M607A11 / M001AC vibration sensors sending readings to a DAQ.

Vibration data for unbalances of different sizes was recorded. The vibration data was recorded at a sampling rate of 4096 values per second. By varying the level of unbalance, different levels of difficulty can be achieved, since smaller unbalances obviously influence the signals at the vibration sensors to a lesser extent.

In total, datasets for 4 different unbalance strengths were recorded as well as one dataset with the unbalance holder without additional weight (i.e. without unbalance). The rotation speed was varied between approx. 630 and 2330 RPM in the development datasets and between approx. 1060 and 1900 RPM in the evaluation datasets. Each dataset is provided as a csv-file with five columns:

1. . V_in : The input voltage to the motor controller V_in (in V)
2. . Measured_RPM : The rotation speed of the motor (in RPM; computed from speed measurements using the DT9837)
3. . Vibration_1 : The signal from the first vibration sensor
4. . Vibration_2 : The signal from the second vibration sensor
5. . Vibration_3 : The signal from the third vibration sensor

Overview of the dataset components:
ID Radius [mm], Mass [g]
0D/ 0E - -
1D/ 1E 14 ± 0.1, 3.281 ± 0.003
2D/ 2E 18.5 ± 0.1, 3.281 ± 0.003
3D/ 3E 23 ± 0.1, 3.281 ± 0.003
4D/ 4E 23 ± 0.1, 6.614 ± 0.007

In order to enable a comparable division into a development dataset and an evaluation dataset, separate measurements were taken for each unbalance strength, respectively. This separation can be recognized in the names of the csv-files, which are of the form “1D.csv”: The digit describes the

unbalance strength (“0” = no unbalance, “4” = strong unbalance), and the letter describes the intended use of the dataset (“D” = development or training, “E” = evaluation).

Data Source: https://www.kaggle.com/datasets/jishnukoliyadan/vibration-analysis-on-rotating-shaft/data

Summary and Conclusions

- Imported the development and evaluation data from the source
- The data was analyzed and modified to make it suitable for predicting load
- Different models were explored and compared to find the best model for predicting load
- A custom metric was developed to evaluate the best model
- The best model was found to be a **Logistic Regression model**

Steps

1. Download the development and evaluation data from the provided link.
2. Read the development data into a pyspark dataframe
3. Explore the pyspark dataframe and observe discrepancies (negative RPM and vibrations, etc..)
4. Group the pyspark dataframe by voltage and load level
5. Convert the grouped pyspark dataframe to pandas dataframe
6. Repeat these steps for the evaluation data
7. Plot the vibration vs. RPM for various load levels for both training and evaluation data and shuffle if necessary to obtain suitable data
8. Define model metric for evaluation
9. Split the data into train and test
10. Develop data pre-processing and modeling pipeline and Grid SearchCV to tune hyperparameters
11. Fit the search to the training data
12. Select the model with the best hyperparameters and evaluate on the test data
13. Plot the confusion matrix and determine the model metric for different models
14. Choose the model with the highest metric value

Dependencies and Initial Definitions

```
import math as mth
import pandas as pd
import numpy as np
from pyspark import SparkContext
from pyspark.sql import SparkSession,DataFrame,functions,types
import pyspark.sql.functions as spark_fn
import pyarrow
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline,make_pipeline
from sklearn.linear_model import LinearRegression,LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier,RandomForestRegressor
from sklearn.metrics import r2_score,mean_squared_error, mean_absolute_error, mean_absolute_
classification_report, roc_curve,auc,confusion_matrix, ConfusionMatrixDisplay
from sklearn.preprocessing import MinMaxScaler, StandardScaler, Normalizer,FunctionTransform
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split,TimeSeriesSplit,GridSearchCV
from xgboost import XGBClassifier,XGBRegressor
import tensorflow as tf
from keras import callbacks
from keras.models import Sequential
from keras.layers import InputLayer,Dense, Flatten,LSTM
# from keras.optimizers import Adam
import scipy
from scipy.stats import linregress
from scipy.stats import pearsonr
import statsmodels
from statsmodels.tsa.stattools import acf
from datetime import datetime, timedelta
import random
import sys
import os
from io import StringIO
import json # library to handle JSON files
from matplotlib import cm,colors
from matplotlib.colors import Normalize
from geopy.geocoders import Nominatim # convert an place into latitude and longitude values
import folium # map rendering library
from bs4 import BeautifulSoup
import camelot
import requests
from IPython import get_ipython


# Create a spark sql session
spark=SparkSession.builder.getOrCreate()
sc=spark.sparkContext
```

```
import warnings

# Customize how warnings are displayed
warnings.filterwarnings('always', module='.*')
warnings.formatwarning = lambda message, category, filename, lineno, line=None: f'{category.
```

General Classes and Functions

```
# Class to create Plots grid

from itertools import product,cycle,combinations
from scipy.stats import pearsonr
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

class Plotsgrid:

    """ Creates a nxn grid of plots for an input df. Shows blank for grid values exceeding r


    def __init__(self,df):
        self.df=df

    def figure_params(self,size_fac1,size_fac2):
        df=self.df
        n_data_cols=len(df.columns)
        self.n_data_cols=n_data_cols

        n_cols=int(n_data_cols**.5)
        n_rows=0
        while n_rows*n_cols<n_data_cols:
            n_rows+=1
        # Create the figure and axes grid
        fig, axs = plt.subplots(nrows=n_rows, ncols=n_cols,figsize=(int(size_fac1*n_data_col

        # Flatten the axs array for easier iteration
        if n_rows == 1 and n_cols == 1:  # Only one subplot
            self.axs = [axs]
        else:
            self.axs = axs.flatten()

    # Loop through the axes and Hist plot
    def histplots(self,bins=-1,density=False):
        # Initialize figure
        self.figure_params(1,2)
        # Create a cycler to iterate over the DataFrame columns
        cycler = cycle(self.df.columns)
        for i,ax in enumerate(self.axs):
```

```
            if i==self.n_data_cols:
                ax.set_visible(False)  # Hide any extra subplots if there are more subplots
                break
            col = next(cycler)  # Get the next column name
            if bins==-1:
                if density:
                    sns.histplot(data=self.df, x=col, ax=ax,stat='density')  # Plot the hist
                else:
                    sns.histplot(data=self.df, x=col, ax=ax)  # Plot the histogram on the cu
            else:
                if density:
                    sns.histplot(data=self.df, x=col, ax=ax,bins=bins,stat='density')  # Plo
                else:
                    sns.histplot(data=self.df, x=col, ax=ax,bins=bins)  # Plot the histogram
        plt.tight_layout()
        plt.show()

    # Loop through the axes and Box plot
    def boxplots(self):

        # Coerce df to numeric:
        df=self.df.apply(pd.to_numeric,errors='coerce').dropna(how='all',axis=1)

        # Initialize figure
        self.figure_params(1,2)
        # Create a cycler to iterate over the DataFrame columns
        cycler = cycle(df.columns)
        for i,ax in enumerate(self.axs):
            try:
                if i==self.n_data_cols:
                    ax.set_visible(False)  # Hide any extra subplots if there are more subpl
                    break
                col = next(cycler)  # Get the next column name
                sns.boxplot(data=df, y=col, ax=ax)  # Plot the boxplot on the current axis
            except Exception as E:
                print(E)
        plt.tight_layout()
        plt.show()

    # Loop through the axes and Line plot
    def lineplots(self):
        # Initialize figure
        self.figure_params(5,5)
        # Create a cycler to iterate over the DataFrame columns
        cycler = cycle(self.df.columns)
        for i,ax in enumerate(self.axs):
            if i==self.n_data_cols:
                ax.set_visible(False)  # Hide any extra subplots if there are more subplots
                break
            col = next(cycler)  # Get the next column name
            print(col)
```

```
            sns.lineplot(data=self.df,x=self.df.index,y=col, ax=ax)  # Plot the Line on the
        plt.tight_layout()
        plt.show()

    def scatterplots(self):

        # Coerce df to numeric:
        df=self.df.apply(pd.to_numeric,errors='coerce').dropna(how='all',axis=1)

        # Get all numeric combinations
        col_combinations = list(combinations(df.columns, 2))
        num_combs = len(col_combinations)

        # Calculate grid size based on the number of combinations
        n_cols = int(np.ceil(np.sqrt(num_combs)))
        n_rows = int(np.ceil(num_combs / n_cols))

        # Create the figure and axes grid
        fig, axs = plt.subplots(nrows=n_rows, ncols=n_cols, figsize=(int(2*n_cols),int(2.5*r

        plt.suptitle('Pairwise Scatter with higher correlations shown brighter')
        axs = axs.flatten()  # Flatten the axes array for easier iteration

        # Iterate through each combination and plot scatterplots
        for i, (col_x, col_y) in enumerate(col_combinations):
            ax = axs[i]

            df_x_y=df[[col_x,col_y]].dropna()

            x = df_x_y[col_x]
            y = df_x_y[col_y]

            # Calculate Pearson correlation and p-value
            if len(x) > 1 and len(y) > 1:  # Ensure there are enough data points
                corr, p_value = pearsonr(x, y)
                corr_array=np.array([corr]*x.shape[0])
                # Scatter plot with colored points
                sns.scatterplot(x=x, y=y, ax=ax,hue=corr_array, palette='coolwarm',hue_norm=

                # Add correlation coefficient and p-value as the legend
                ax.text(0.05, 0.95, f"r = {corr:.2f}\np = {p_value:.2e}",
                        transform=ax.transAxes, fontsize=7, verticalalignment='top',
                        bbox=dict(boxstyle="round,pad=0.3", facecolor="lightgray", edgecolor

                # Set plot labels
                ax.set_xlabel(col_x)
                ax.set_ylabel(col_y)

        # Hide any extra subplots
        for j in range(i + 1, len(axs)):
            axs[j].set_visible(False)
```

```
            plt.tight_layout()
            plt.show()


    def mape(y_true, y_pred):
        """Compute the mean absolute percentage error (MAPE)."""
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100


    def basic_eda(df):
        # Check if 'df' is a DataFrame
        if isinstance(df, pd.DataFrame):
            pass
        else:
            df=pd.DataFrame(df)
        cols=df.columns.tolist()
        print('\nCOLUMNS LIST: ')
        print(cols)
        print('\nFIRST FEW: ')
        print(df.head(3))
        print('\nLAST FEW: ')
        print(df.tail(3))
        print('\nINFO: ')
        print(df.info())
        print('\nDESCRIPTION: ')
        print(df.describe())
        print('\nUNIQUE VALUES: ')
        [print(col+' Unique: ',len(df[col].value_counts())) for col in df.columns];
        return cols


    def basic_eda_spark(df):

        # Check if 'df' is a Spark DataFrame
        if isinstance(df, DataFrame):
            pass
        else:
            raise TypeError("The input is not a Spark DataFrame")
        # Create a table view
        df.createOrReplaceTempView("eda_table")
        # Get the column list
        cols = df.columns
        print('\nCOLUMNS LIST: ')
        print(cols)

        # Show first few rows (head)
        print('\nFIRST FEW: ')
        df.show(3)

        # Show last few rows (tail equivalent in PySpark by ordering)
        #print('\nLAST FEW: ')
```

```
        #df.orderBy(df.columns[0], ascending=False).show(3)

        # Show schema (info equivalent)
        print('\nINFO (Schema): ')
        df.printSchema()

        # Show summary statistics (describe equivalent)
        print('\nDESCRIPTION: ')
        df.describe().show()

        # Count of unique values:
        print('\nCOUNT UNIQUE: ')
        [print(col+': ',df.select(col).distinct().count()) for col in df.columns];

        return cols



    def plot_line_from_spark(df):
        df_plot=df.limit(100000).toPandas()
        for col in df.columns:
            df_plot[col].plot(kind='line');
            plt.legend([col])
            plt.show();



    # Read local csv
    def read_local_csv(filenames):
        '''Read a file or list of files into a dataframe'''
        if type(filenames)==list:
            df=pd.DataFrame()
            for file in filenames:
                df_file=pd.read_csv(data_dir+'\\'+file)
                load_num=[i for i in file if  i.isdigit()][0]
                df_file['load']=load_num
                df=pd.concat([df,df_file],axis=0)
        else:
            df=pd.read_csv(data_dir+'\\'+filenames)
            load_num=[i for i in file if  i.isdigit()][0]
            df['load']=load_num
        return df

    def read_local_csv_object(filenames):
        print(filenames)
        ''' Read a csv or list of csvs into a spark object'''
        if type(filenames)==list:
            file=filenames[0]
            spark_df=spark.read.csv(data_dir+'\\'+file,header=True,inferSchema=True)
            load_num=[int(i) for i in file if  i.isdigit()][0]
            spark_df=spark_df.withColumn('load',spark_fn.lit(load_num))
            for file in filenames[1:]:
```

```
            spark_df_file=spark.read.csv(data_dir+'\\'+file,header=True,inferSchema=True)
            load_num=[int(i) for i in file if  i.isdigit()][0]
            spark_df_file=spark_df_file.withColumn('load',spark_fn.lit(load_num))
            spark_df=spark_df.union(spark_df_file)
    else:
        spark_df= spark.read.csv(data_dir+'\\'+filenames,header=True,inferSchema=True)
    return spark_df
```

Read data

```
current_directory=os.getcwd()

# List all files in the data directory
data_dir=current_directory+'\kaggle\working'
files = os.listdir(data_dir);

filenames=[file for file in files if '.csv' in file]
print(filenames)
```

```
['0D_trunc.csv', '0E_trunc.csv', '1D_trunc.csv', '1E_trunc.csv', '2D_trunc.csv', '2E_tru
SyntaxWarning: invalid escape sequence '\k'
SyntaxWarning: invalid escape sequence '\k'
SyntaxWarning: invalid escape sequence '\k'
```

```
# Development and evaluation files
filenames_D=[filename for filename in filenames if 'D' in filename]
filenames_E=[filename for filename in filenames if 'E' in filename]


# Read the development files
df_D_spark=read_local_csv_object(filenames_D)
```

```
['0D_trunc.csv', '1D_trunc.csv', '2D_trunc.csv', '3D_trunc.csv', '4D_trunc.csv']
```

Exploratory Data Analysis (EDA)

```
# Perform basic EDA
basic_eda_spark(df_D_spark)
```

```
COLUMNS LIST:
['V_in', 'Measured_RPM', 'Vibration_1', 'Vibration_2', 'Vibration_3', 'load']

FIRST FEW:
+----+------------+-----------+-----------+-----------+----+
|V_in|Measured_RPM|Vibration_1|Vibration_2|Vibration_3|load|
```

```
+----+------------+-----------+-----------+-----------+----+
| 0.0|   28.610235|        0.0|        0.0|        0.0|   0|
| 0.0|   28.610235|        0.0|        0.0|        0.0|   0|
| 0.0|   28.610235|        0.0|        0.0|        0.0|   0|
+----+------------+-----------+-----------+-----------+----+
only showing top 3 rows


INFO (Schema):
root
 |-- V_in: double (nullable = true)
 |-- Measured_RPM: double (nullable = true)
 |-- Vibration_1: double (nullable = true)
 |-- Vibration_2: double (nullable = true)
 |-- Vibration_3: double (nullable = true)
 |-- load: integer (nullable = false)


DESCRIPTION:
+-------+-----------------+-------------------+--------------------+--------------------
|summary|             V_in|       Measured_RPM|         Vibration_1|         Vibration_2
+-------+-----------------+-------------------+--------------------+--------------------
|  count|         13201553|           13201553|            13201553|            1320155
|   mean|5.995143904660508|-35698.335014651355|0.001651138868151...|0.002609488510923...
| stddev|2.327930769161795|  2986857.101551288| 0.05552156238148596|  0.084574367461515
|    min|              0.0|             -2.4E8|         -0.12001276|         -0.2157604
|    max|             10.0|           4091.723|           7.8491378|           8.798155
+-------+-----------------+-------------------+--------------------+--------------------


COUNT UNIQUE:
V_in:  162
Measured_RPM:  27804
Vibration_1:  122577
Vibration_2:  164840
Vibration_3:  62060
load:  5
['V_in', 'Measured_RPM', 'Vibration_1', 'Vibration_2', 'Vibration_3', 'load']
```

```
# Explore occurence of negative RPM values
df_D_spark.filter((df_D_spark.Measured_RPM<0)).describe().show()
```

```
+-------+----+------------+------------------+-------------------+------------------+---
|summary|V_in|Measured_RPM|       Vibration_1|         Vibration_2|       Vibration_3|
+-------+----+------------+------------------+-------------------+------------------+---
|  count|2045|        2045|              2045|               2045|              2045|
|   mean| 0.0|      -2.4E8|1.1340587862948655| 2.7391967250366744|1.4354542156332515|
| stddev| 0.0|         0.0|0.8078785936476082|0.49377066863850905|0.8008889551017733|1.4
|    min| 0.0|      -2.4E8|       0.010514259|          1.8436027|        0.27963758|
|    max| 0.0|      -2.4E8|         2.4504042|          3.8436234|          2.748549|
+-------+----+------------+------------------+-------------------+------------------+---
```

- There are values of RPM < 0. This is clearly some kind of measurement fault as an RPM of -2.4E8 is unrealistic. Hence we can disregard values lower than 0
- There are also negative vibration values; we can turn them positive since we are interested in the magnitude of the vibration, not the direction.

```
# Remove RPM <0
df_D_spark=df_D_spark.filter((df_D_spark.Measured_RPM>=0))
print(df_D_spark)
```

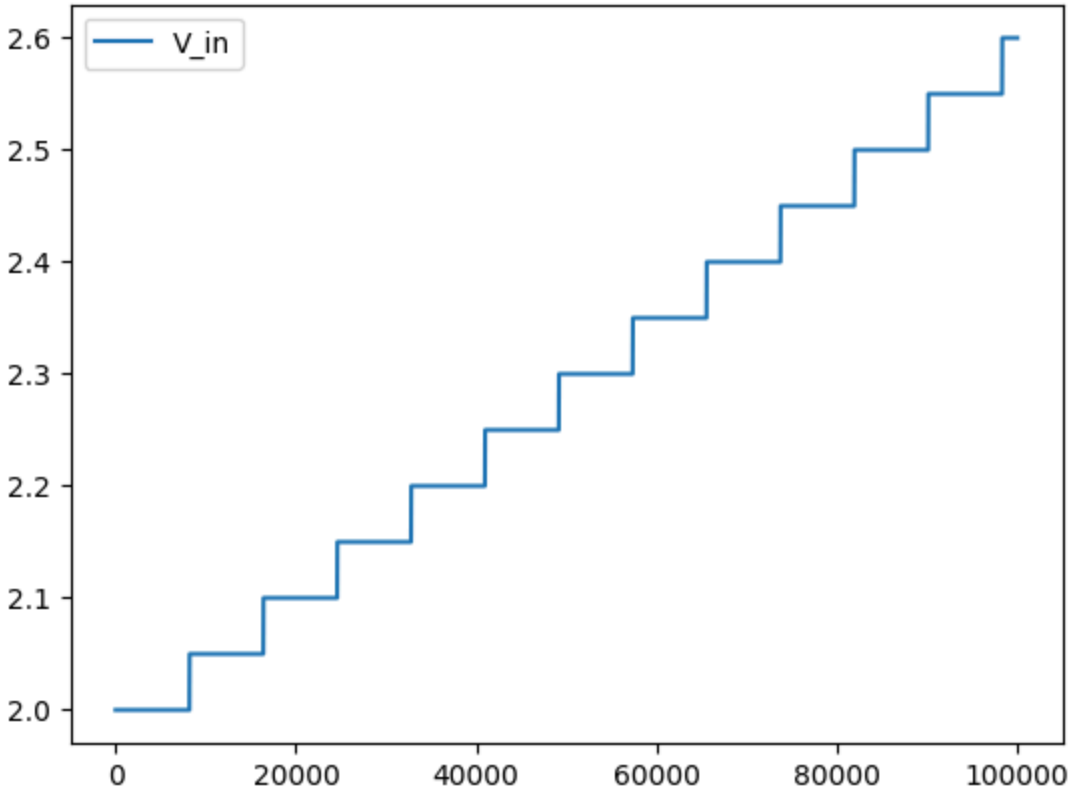
```
DataFrame[V_in: double, Measured_RPM: double, Vibration_1: double, Vibration_2: double,
```

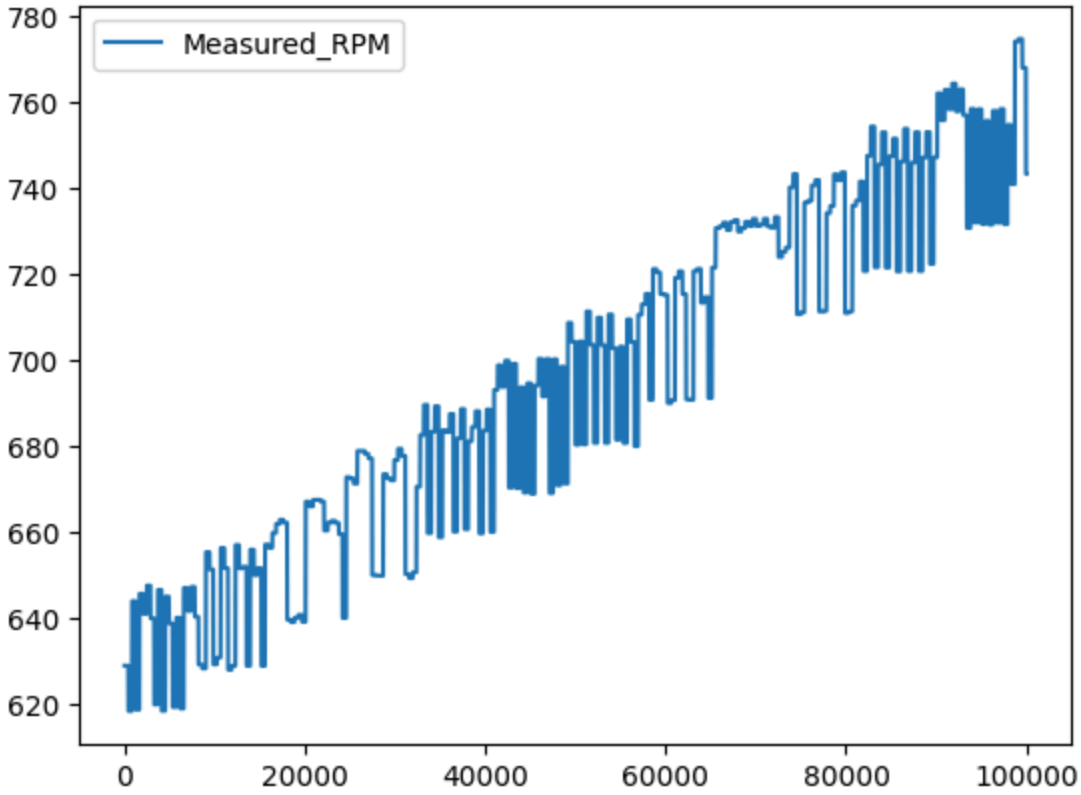
```
# Turn vibration values to absolute
for col in df_D_spark.columns:
    if 'Vib' in col:
        df_D_spark=df_D_spark.withColumn(col, spark_fn.abs(spark_fn.col(col)))
df_D_spark.describe().show()
```

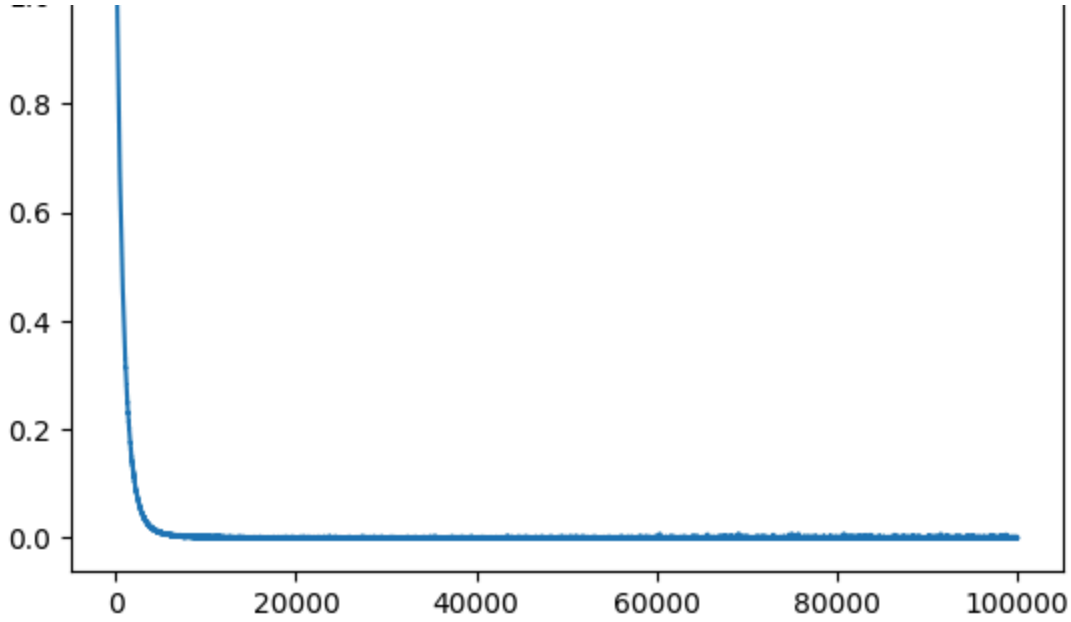
```
+-------+-----------------+-----------------+--------------------+--------------------+-
|summary|             V_in|     Measured_RPM|         Vibration_1|         Vibration_2|
+-------+-----------------+-----------------+--------------------+--------------------+-
|  count|         13199508|         13199508|            13199508|            13199508|
|   mean|5.996072732408105|1479.338342939554|0.006926123949303975|0.008946409263340052|0
| stddev| 2.32691468542015|494.5346613461752| 0.05232164670326984| 0.07668427695074911|
|    min|              0.0|        28.610235|                 0.0|                 0.0|
|    max|             10.0|         4091.723|           7.8491378|           8.7981558|
+-------+-----------------+-----------------+--------------------+--------------------+-
```

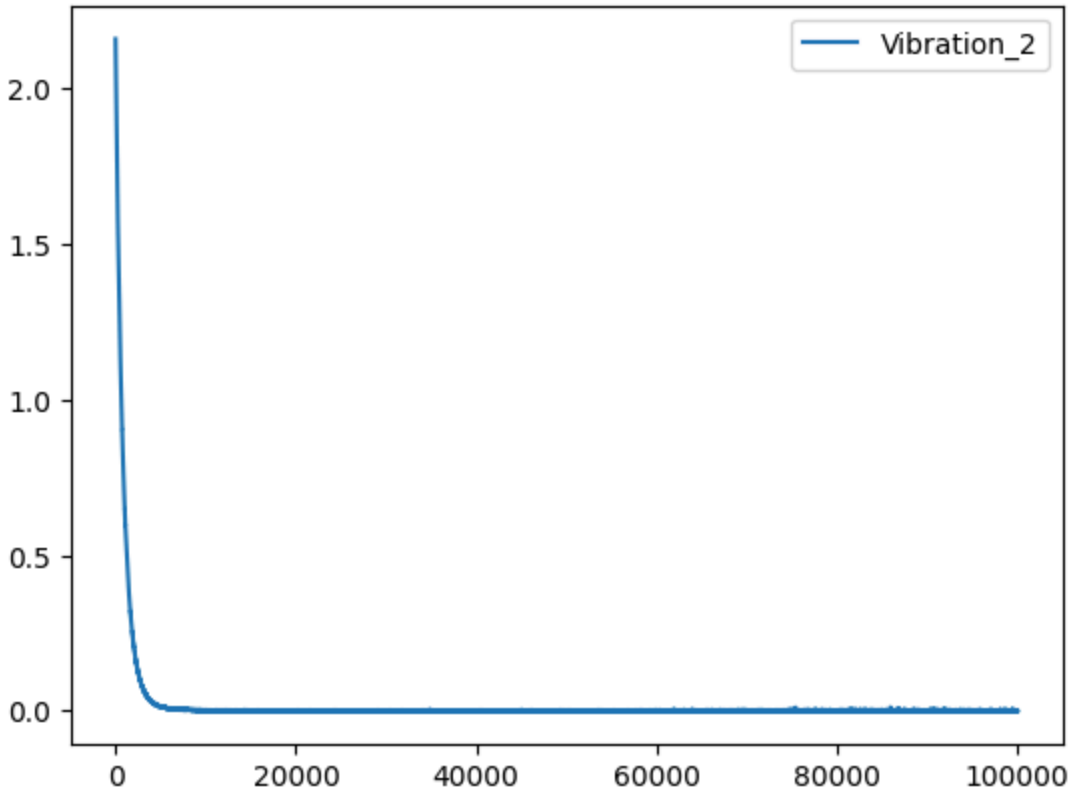
```
plot_line_from_spark(df_D_spark)
```

```
DeprecationWarning: distutils Version classes are deprecated. Use packaging.version ir
DeprecationWarning: distutils Version classes are deprecated. Use packaging.version ir
ResourceWarning: unclosed <socket.socket fd=7140, family=2, type=1, proto=0, laddr=('1
```

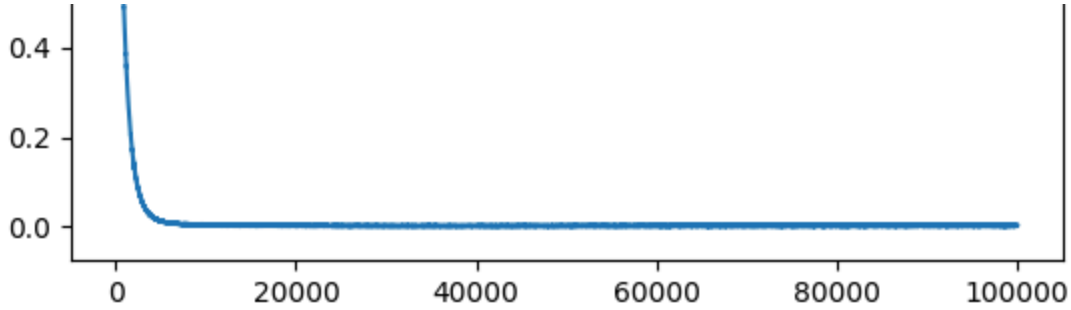


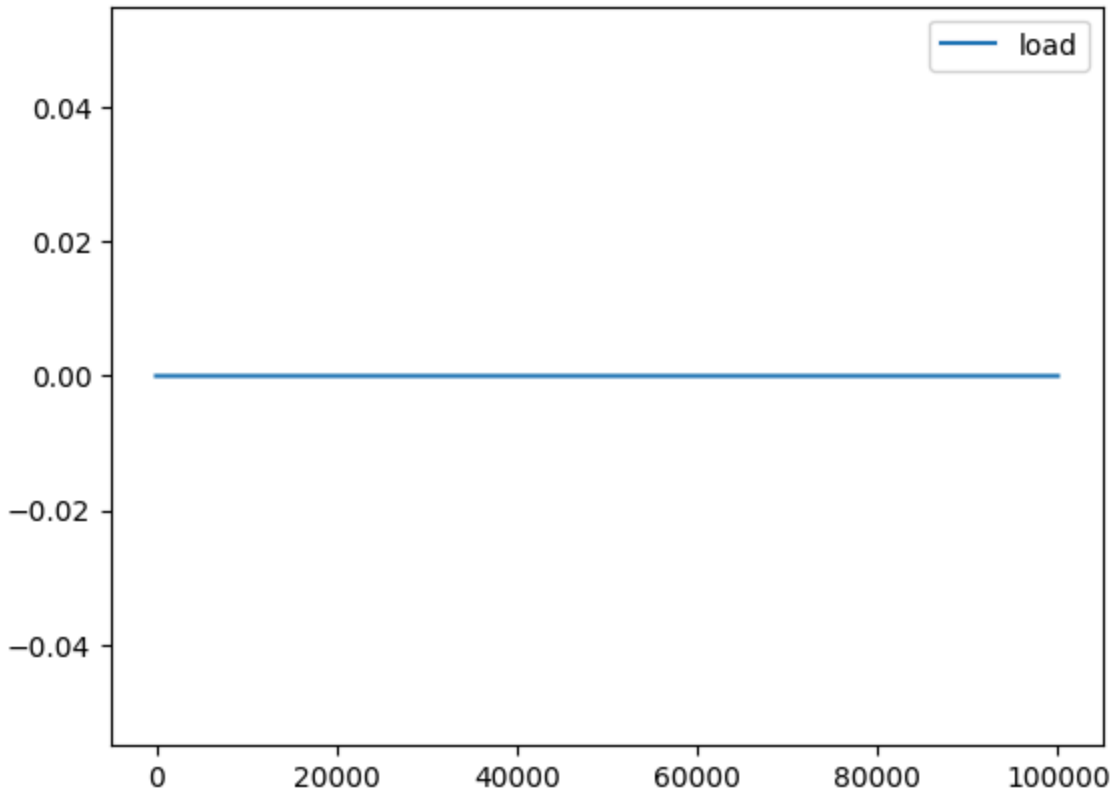



0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
100000


Vibration_2
2.0
1.5
1.0
0.5
0.0
0
20000
40000
60000
80000
100000


Vibration_3
1.4
1.2
1.0
0.8
0.6


0.4
0.2
0.0
0
20000
40000
60000
80000
100000


load
0.04
0.02
0.00
−0.02
−0.04
0
20000
40000
60000
80000
100000

- We can filter the data further, to say RPM > 600. This will filter out the spike in vibration signal as well
- Based on the voltage and RPM patterns, it looks like the purpose of the experiment was to maintain the motor at specific RPMs.
- We can take the mean of the parameters for each voltage level to understand how the speed and vibration vary with voltage, while also drastically reducing data size.

```
df_D_spark=df_D_spark.filter(df_D_spark.Measured_RPM>600)
```

```
df_D_spark_grouped=df_D_spark.groupBy(['V_in','load']).mean()
```

'Load' was included in the grouping to prevent getting the mean values of load instead of the actual load values in the df

```
df_D_spark_grouped.describe().show()
```

```
+-------+-----------------+------------------+------------------+------------------+----
|summary|             V_in|              load|         avg(V_in)| avg(Measured_RPM)|
+-------+-----------------+------------------+------------------+------------------+----
|  count|              805|               805|               805|               805|
|   mean|5.999999999999997|               2.0| 5.999999999999992|1480.2197952421488|0.00
| stddev|2.325234701682919|1.4150927751172553|2.3252347016829047|492.42333510183096|0.00
|    min|              2.0|                 0|               2.0| 636.9171586791892|5.10
|    max|             10.0|                 4|              10.0|2332.1845971923567| 0.0
+-------+-----------------+------------------+------------------+------------------+----
```

Convert to Pandas

```
# Let's first convert to pandas
df_D_grouped=df_D_spark_grouped.toPandas()
```

```
DeprecationWarning: distutils Version classes are deprecated. Use packaging.version inst
DeprecationWarning: distutils Version classes are deprecated. Use packaging.version inst
ResourceWarning: unclosed <socket.socket fd=6128, family=2, type=1, proto=0, laddr=('127
```

```
df_D_grouped.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 805 entries, 0 to 804
Data columns (total 8 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   V_in               805 non-null    float64
 1   load               805 non-null    int32
 2   avg(V_in)          805 non-null    float64
```

```
 3   avg(Measured_RPM)  805 non-null    float64
 4   avg(Vibration_1)   805 non-null    float64
 5   avg(Vibration_2)   805 non-null    float64
 6   avg(Vibration_3)   805 non-null    float64
 7   avg(load)          805 non-null    float64
dtypes: float64(7), int32(1)
memory usage: 47.3 KB
```

```
# We can drop the avg V_in and avg load columns and remove 'avg' from the column names
df_D_grouped.columns=[col.replace('avg','').replace('(','').replace(')','') for col in df_D_
# Remove duplicate columns
df_D_grouped=df_D_grouped.loc[:, ~df_D_grouped.columns.duplicated()]
```

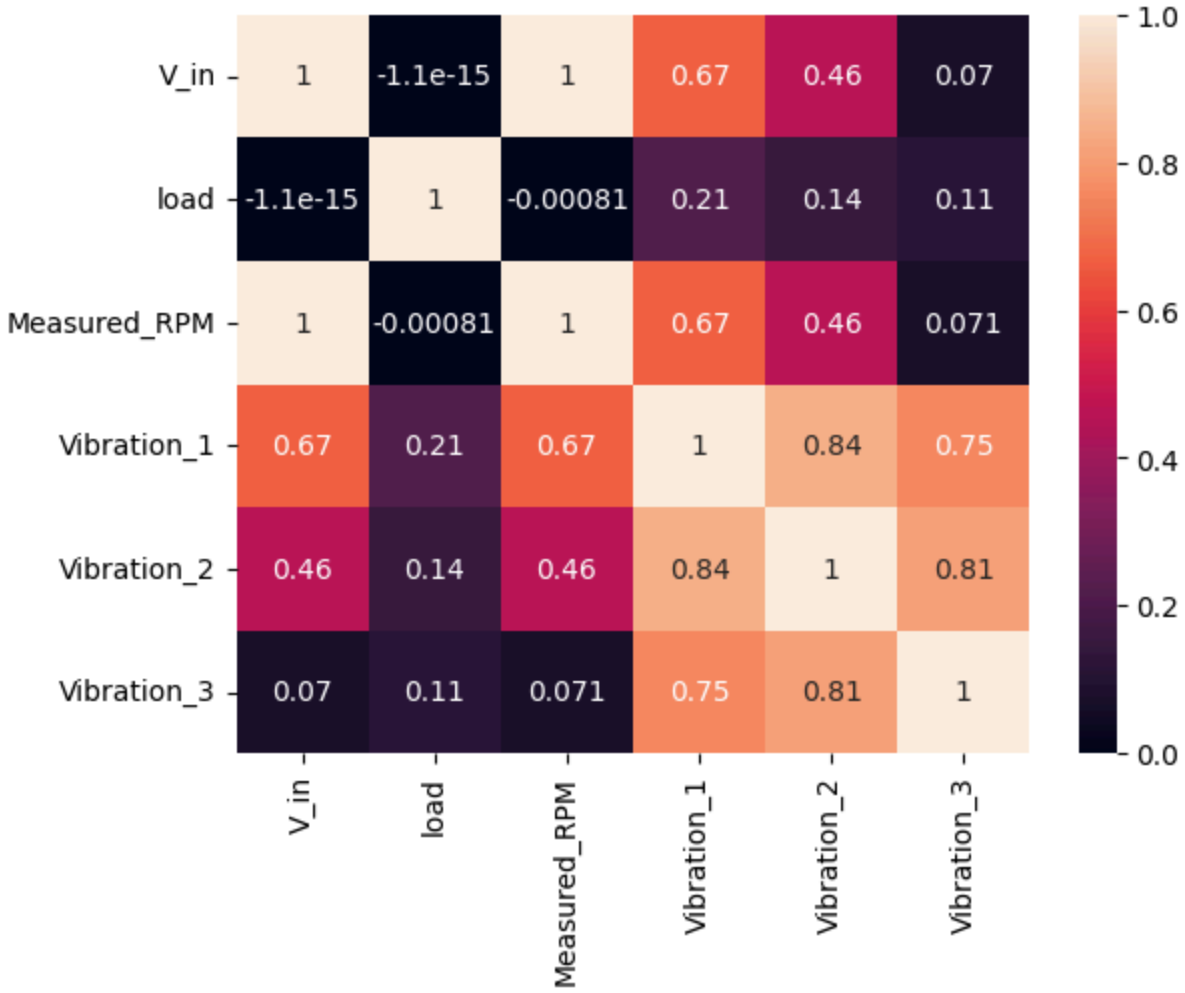
```
df_D_grouped.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 805 entries, 0 to 804
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   V_in          805 non-null    float64
 1   load          805 non-null    int32
 2   Measured_RPM  805 non-null    float64
 3   Vibration_1   805 non-null    float64
 4   Vibration_2   805 non-null    float64
 5   Vibration_3   805 non-null    float64
dtypes: float64(5), int32(1)
memory usage: 34.7 KB
```

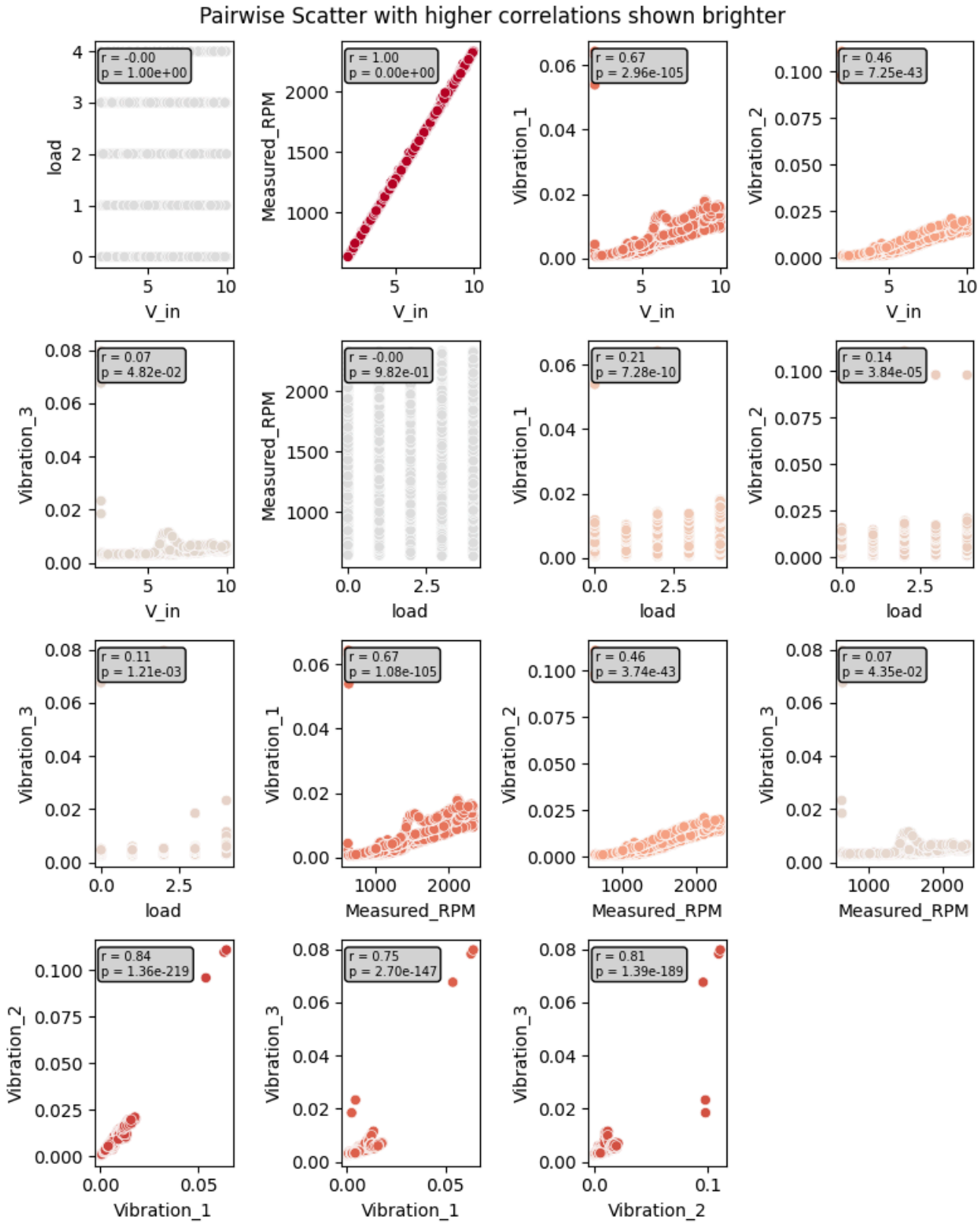
We can see the correlations by plotting a heatmap

```
sns.heatmap(df_D_grouped.corr(),annot=True)
```

```
<Axes: >
```



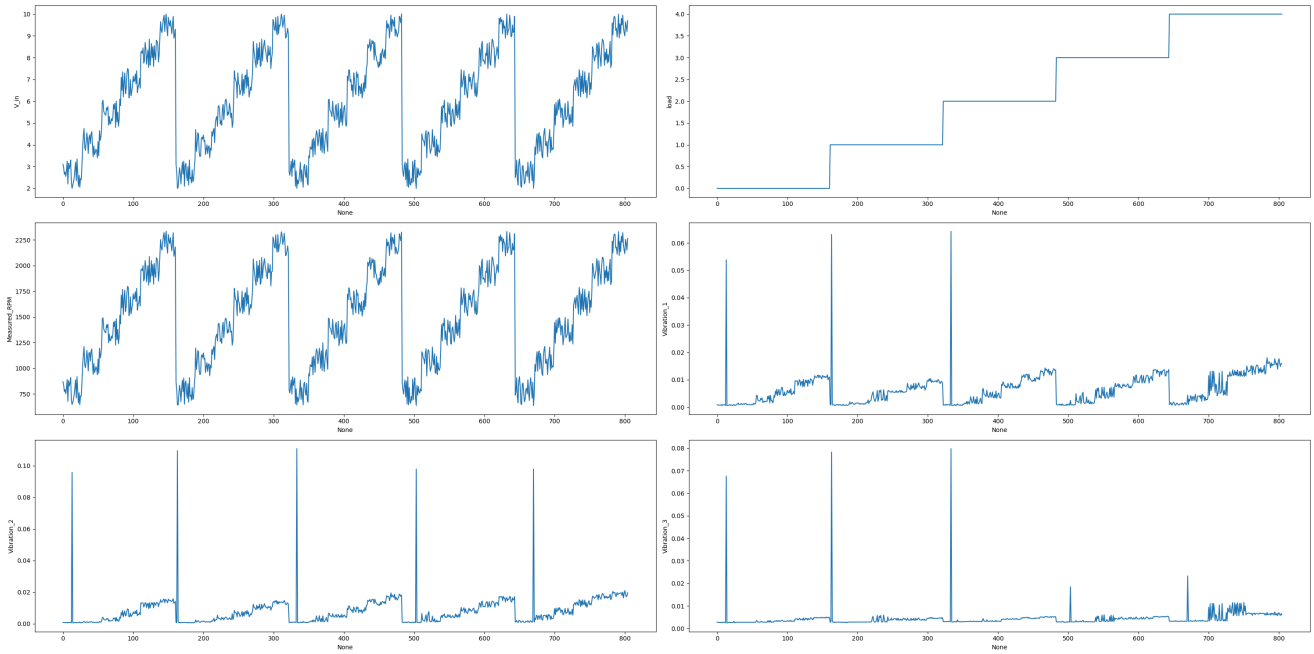

```
Plotsgrid(df_D_grouped).scatterplots()
```


Pairwise Scatter with higher correlations shown brighter
r = -0.00
p = 1.00e+00
r = 1.00
p = 0.00e+00
r = 0.67
p = 2.96e-105
r = 0.46
p = 7.25e-43
r = 0.07
p = 4.82e-02
r = -0.00
p = 9.82e-01
r = 0.21
p = 7.28e-10
r = 0.14
p = 3.84e-05
r = 0.11
p = 1.21e-03
r = 0.67
p = 1.08e-105
r = 0.46
p = 3.74e-43
r = 0.07
p = 4.35e-02
r = 0.84
p = 1.36e-219
r = 0.75
p = 2.70e-147
r = 0.81
p = 1.39e-189
load
Measured_RPM
Vibration_1
Vibration_2
Vibration_3
V_in

- The vibration components are correlated weakly to load

```
Plotsgrid(df_D_grouped).lineplots()
```

```
V_in
load
Measured_RPM
Vibration_1
Vibration_2
Vibration_3
```



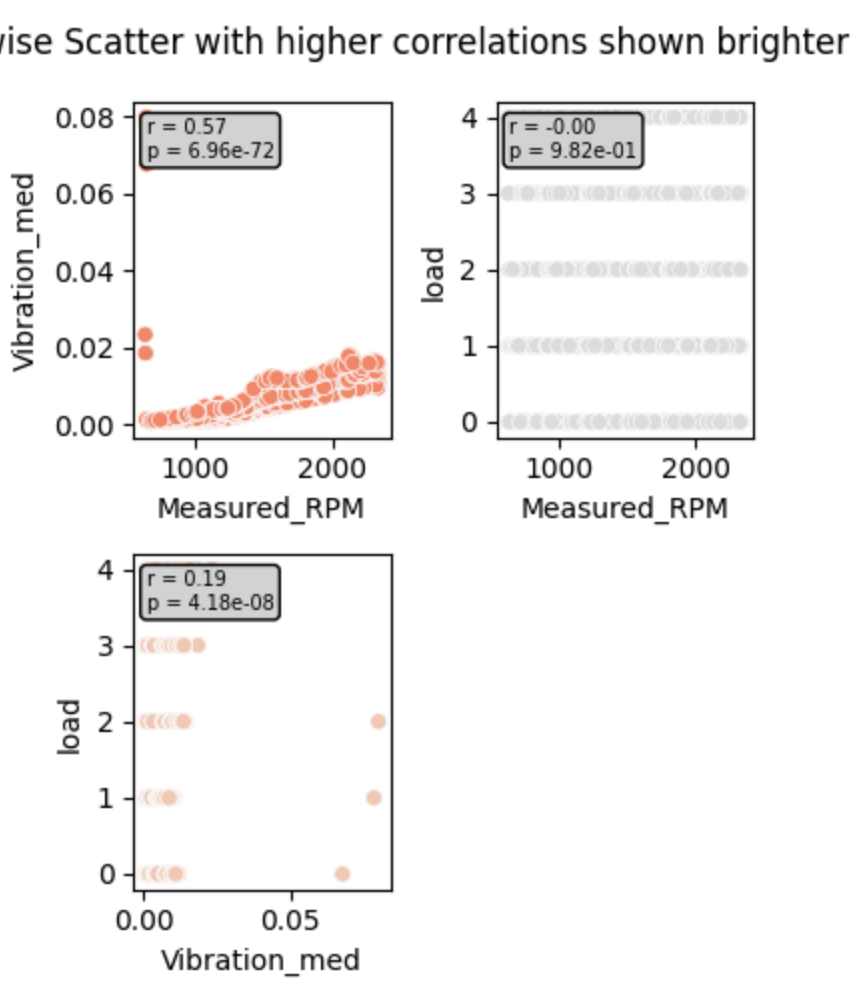
- As the load (unbalance) progresses, the vibrations for the same cycle generally increases with load. This is a small correlation between vibration and load
- We also see the vibrations being strongly correlated. This is expected for a rigid shaft. Therefore, we can consider the median value of the vibrations for our analysis

```
df_D_grouped['Vibration_med']=df_D_grouped[['Vibration_1','Vibration_2','Vibration_3']].medi
```

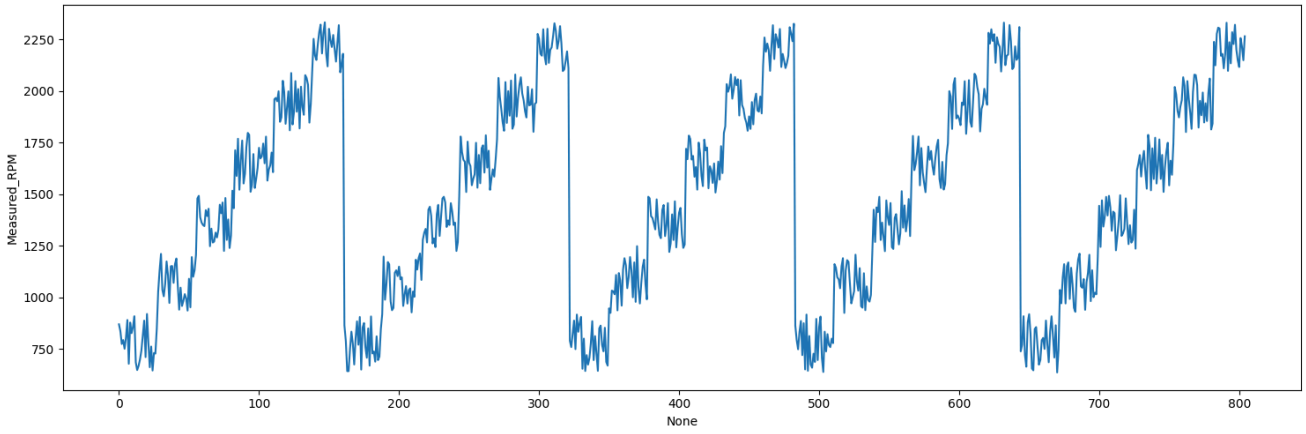
Make plots for: RPM, median vibration and load

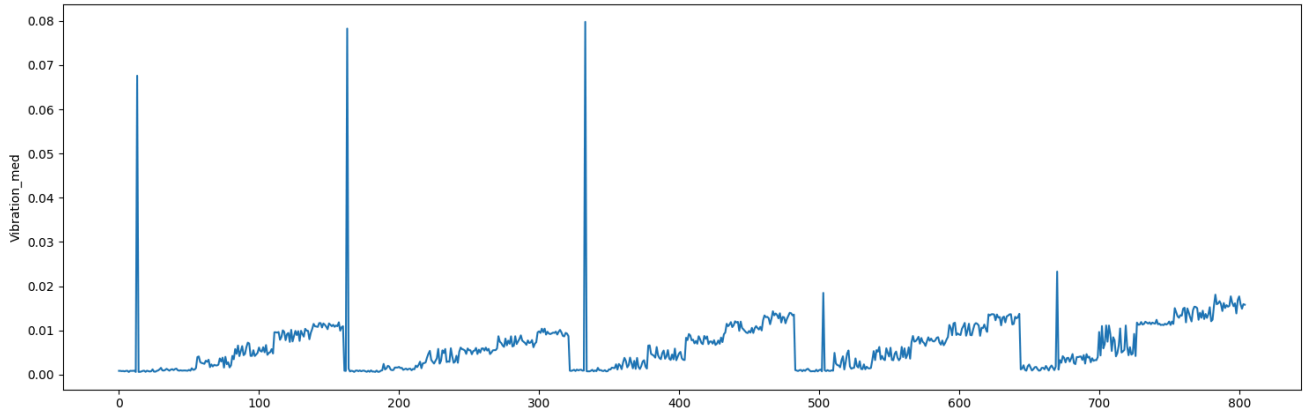
```
df_D_grouped_featured=df_D_grouped[['Measured_RPM','Vibration_med','load']]
```

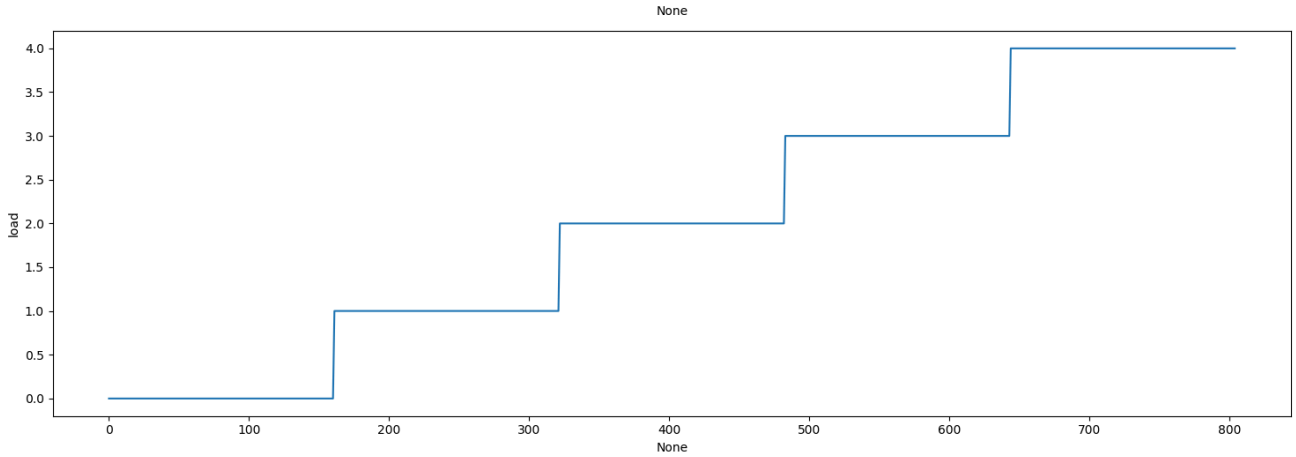
```
Plotsgrid(df_D_grouped_featured).scatterplots()
Plotsgrid(df_D_grouped_featured).lineplots()
```




```
Measured_RPM
Vibration_med
load
```



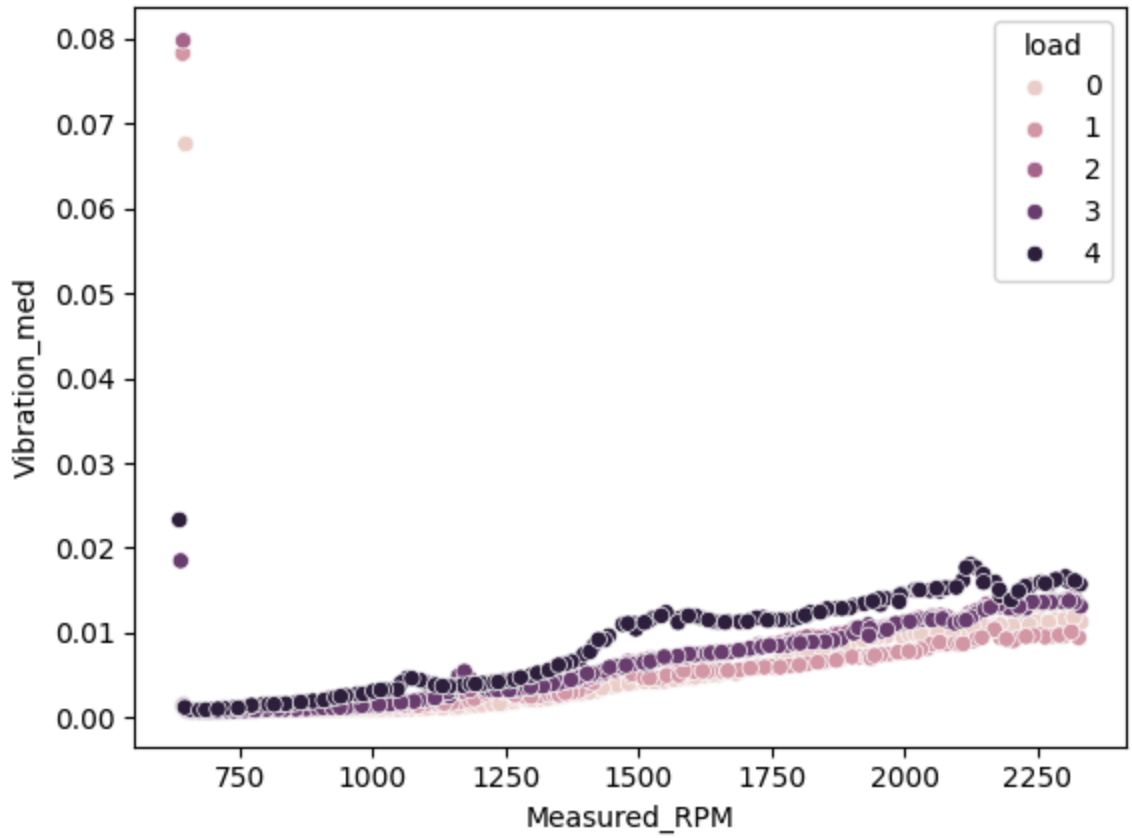



None
load
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
100
200
300
400
500
600
700
800
None

Let's explore also: Vibration vs. RPM for different load levels

```
sns.scatterplot(data=df_D_grouped_featured,x='Measured_RPM',y='Vibration_med',hue='load')
```

```
<Axes: xlabel='Measured_RPM', ylabel='Vibration_med'>
```



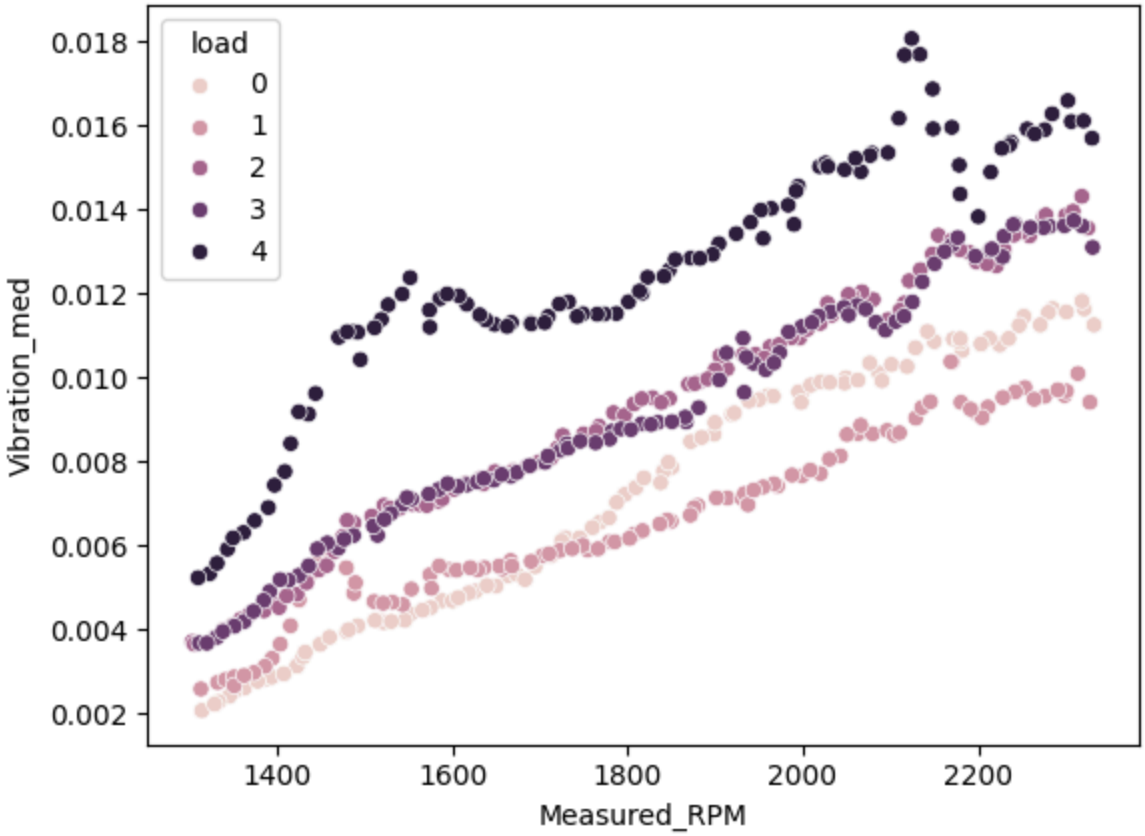
- The vibrations do indeed rise as load increases, for a given RPM.
- The effect is more pronounced between loads 0,1,2 and loads 3,4.
- Finally we can filter out RPMs below 1300 as the trend is much stronger above this value. We also get rid of the outlier vibration points at low RPM, likely electrical issues

Therefore we can create a classification model to predict the vibration level using the RPM and median vibrations

```
df_D_grouped_featured=df_D_grouped_featured[df_D_grouped_featured.Measured_RPM>1300]
```

```
sns.scatterplot(data=df_D_grouped_featured,x='Measured_RPM',y='Vibration_med',hue='load')
```

```
<Axes: xlabel='Measured_RPM', ylabel='Vibration_med'>
```



- There is barely any resolution between load levels 2 and 3. So one would expect the number of true positives here to be fewer
- Load level 0 shows *more* vibration than load level 1 at higher speeds- this is interesting , as one might expect the opposite
- Increasing the weight of the load (load 4) has a drastic effect on the vibration characteristics

Data Processing Method Summary

1. Add a 'load' column corresponding to level of unbalance
2. Filter for RPM > 1300
3. Take absolute values of vibrations
4. Group by 'V_in' and 'load' and remove the additional columns
5. Calculate median of vibrations 1,2,3
6. select 'Measured_RPM','Vibration_med' and 'load' columns

```
def process_files(filename):
    ''' Function to read in and process a file to make it compatible for modeling'''
    #print(filename)
    #  Read the file
    spark_df=read_local_csv_object(filenames_E)
```

```
    # Remove RPM <1300
    spark_df=spark_df.filter((spark_df.Measured_RPM>=1300))
    # Absolute values for vib
    for col in spark_df.columns:
        if 'Vib' in col:
            spark_df=spark_df.withColumn(col, spark_fn.abs(spark_fn.col(col)))
    # Group by V_in,load
    spark_df=spark_df.groupBy(['V_in','load']).mean()
    # Convert to pandas
    df=spark_df.toPandas()
    # We can drop the avg V_in and avg load columns and remove 'avg' from the column names
    df.columns=[col.replace('avg','').replace('(','').replace(')','') for col in df.columns]
    # Remove duplicate columns
    df=df.loc[:, ~df.columns.duplicated()]
    # Calcuate median of vibrations
    df['Vibration_med']=df[['Vibration_1','Vibration_2','Vibration_3']].median(axis=1)
    # Select RPM, median vibration and load
    df=df[['Measured_RPM','Vibration_med','load']]

    return df



# Create the evaluation df similar to the 'development' df
df_E_grouped_featured=process_files(filenames_E)
```

```
['0E_trunc.csv', '1E_trunc.csv', '2E_trunc.csv', '3E_trunc.csv', '4E_trunc.csv']
DeprecationWarning: distutils Version classes are deprecated. Use packaging.version inst
DeprecationWarning: distutils Version classes are deprecated. Use packaging.version inst
ResourceWarning: unclosed <socket.socket fd=7688, family=2, type=1, proto=0, laddr=('127
```

```
# Explore the evaluation df
basic_eda(df_E_grouped_featured)
```

```
COLUMNS LIST:
['Measured_RPM', 'Vibration_med', 'load']

FIRST FEW:
   Measured_RPM  Vibration_med  load
0   1353.860217       0.002436     0
1   1346.793309       0.002245     0
2   1324.485053       0.002017     0

LAST FEW:
     Measured_RPM  Vibration_med  load
152   1932.797451       0.014683     4
153   1882.744901       0.015137     4
154   1874.222343       0.014631     4

INFO:
<class 'pandas.core.frame.DataFrame'>
```