University of South-Eastern Norway

www.usn.no

IIA2017: Industrial IT

Assignment 1 – Prediction Electric Engine Status using Audio Analysis and Machine Learning

Joel Honkanen 237611

Faculty of Technology, Natural sciences and Maritime Sciences
Campus Porsgrunn

Contents

1 Introduction

This is assignment 2 in the courseIIA1420 Machine Learning and Sensor Technology. In this assignment sound data from industrial motors are analyzed and classified into three different categories. These are good, broken and heavy load. The dataset was recorded in 2017 at the Fraunhofer Institute [5]. This report describes how a data model is constructed, data preparation for machine learning and the actual training and evaluation of classifiers. Several calculations such as RMS (Root Mean Square), Crest Factor, FFT (to calculate the DFT) and Mel Spectrograms are evaluated as potential features for the Machine Learning. All calculations are performed in Python and Jupyter notebook, using the libraries Numpy, Matplotlib, Librosa, Sci-kit learn and Pandas.

1.1 Dataset

The dataset is sampled with 44100Hz, which means 44100 samples per second. The dataset is split into 3 second samples and divided into training and test datasets. This is convenient as sound data can be large and slow to compute. In this assignment only the 3 second samples are used

Table 1.1: Training data samples

Category	**Number of 3 s. samples**
Good engine	105
Broken engine	124
Heavy load engine	128

Table 1.2: Test data samples

Category	**Number of 3 s. samples**
Good engine	669
Broken engine	665
Heavy load engine	687

The test set is much larger, but the training set only contain pure recording of motor sound, while the test set is divided into several categories of noise in the recordings. In this report the given split between train and test is kept as is. But the test set was divided into two test sets, to validate the selected model once more (done due to the large amount and size of data this was feasible). The background noise added to test set is persons talking, white noise, atmospheric (low, med, high) and stress test

2 Results

In this chapter the result will be presented by showing Pyhton visualizations and calculations. The most important parts of code are shown below, for complete code refer to Appendix 1-3.

2.1 Data model

To be able to test different combinations and make the code scalable for future use, the majority of the work was aimed toward build an object oriented data model. Inspiration to the data flow was gotten from [3] even though they rely on other packages. A Jupyter notebook called assignment2.ipynb (see Appendix 1) is used as the main workflow. Instances of data objects will be instantiated here. All theoretical calculations are static methods in a python file called sounds.py (see Appendix 2).

```
# Imports
import numpy as np
import wave
import librosa, librosa.display
import pandas as pd
from scipy.fft import fft, fftfreq, fftshift
import scipy.signal
import matplotlib.pyplot as plt

# Class for single sound sample (or optional appended samples) and various processing and plotting
# There are only static methods in this class that return results without  making instances of variables

class Sound:
    # Read in raw data file(s) and make array
    @staticmethod
    def readwavfile(files, wavfolder):
        #print(files, wavfolder)
        arrlist = []
        # Read wav file(s)
        for file in files:
            with wave.open(wavfolder + file) as wav_file:
                frames = wav_file.readframes(wav_file.getnframes())
                arr = np.frombuffer(frames, dtype=np.int16)
                #print(arr)
            arrlist.append(arr)
            #self.N += len(arr)
        # Transform to single 1D numpy array
        if len(arrlist) > 1:
            return np.concatenate(arrlist)
        else:
            return arrlist[0]

    # Low pass filter
    @staticmethod
    def lowpass(data, fs, cutoff, poles):
        sos = scipy.signal.butter(poles, cutoff, 'lowpass', fs=fs, output='sos')
        return scipy.signal.sosfiltfilt(sos, data)
```

Figure 2.1: Main part of static method class in sound.py, only showing the first methods

The file dataset.py (see Appendix 3) contain no sound data, instead it keeps track of all filenames and labels. Sound files are generated when the get item is called from outside. Most important task for this class is to keep track of the metadata, such as file names and folders, and make sure the labels to be classified are tied to the correct file even when shuffling and slicing in the main workflow notebook.

```python
# Imports
import numpy as np
import wave
import librosa, librosa.display
import pandas as pd
from scipy.fft import fft, fftfreq, fftshift
import scipy.signal
import matplotlib.pyplot as plt

# The class with static methods for individual processing
from sound import Sound

# Class that structures the datasets and for a collection of sound samples
# Calls are made to the Sound() class in sound.py

class Dataset():
    def __init__(self, files, wavfolder, fs, labels):
        self.files = files
        self.wavfolder = wavfolder
        self.fs = fs
        self.labels = labels

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        file = []
        file.append(self.files[idx])
        soundarr = Sound.readwavfile(file, self.wavfolder)
        return soundarr, self.labels[idx], self.files[idx]

    def getfs(self):
        return self.fs
```

Figure 2.2: Code for class Dataset in dataset.py

The last file is called calculate.py (see Appendix 4). This code contains the individual sound file arrays that are read in in a loop in the dataset.py. All other calculations, such as RMS, FFT etc., are done here for all instantiated files, but the actual result is returned by each method, not stored in the class. It is possible to retrieve the original sound array from here with getsound() method. This is good when inspecting samples individually, but if a new copy of the FFT is needed, then the calculation is done again (it was not desirable to store all data in the class objects, enough to keep the sound array, to keep size down. The class Dataset objects are instantiated in this Calculate class constructor, and the list of sound arrays are built by the constructor.

```python
# Imports
import numpy as np
import wave
import librosa, librosa.display
import pandas as pd
from scipy.fft import fft, fftfreq, fftshift
import scipy.signal
import matplotlib.pyplot as plt

# The class with static methods for individual processing
from sound import Sound

# Class that perform the actual calculations for data in datasets input
# This class also outputs the good, broken and heavyload in a combined numpy array, including labels

class Calculate():
    def __init__(self, datasets):
        self.datasets = datasets
        self.sounds = []
        self.labels = []
        self.files = []
        # Make list of sound array(s) with labels
        for ds in self.datasets:
            for idx in range(len(ds)):
                d, l, f = ds[idx]
                self.sounds.append(d)
                self.labels.append(l)
                self.files.append(f)

    # Get copy of soundarrays as list
    def getsound(self):
        return np.array(self.sounds), np.array(self.labels), self.files

    # Calculate dft with fft for all arrays
    def calcfft(self, fpar=[44100, 2000, 3], filt=False):
        ffts = []
        for sound in self.sounds:
            if filt:
                sound = Sound.lowpass(sound, fpar[0], fpar[1], fpar[2])
            fft = Sound.soundfft(sound, self.datasets[0].getfs())
            ffts.append(fft)
        return np.array(ffts), np.array(self.labels), self.files

    # Calculate mel spec for all arrays
    def calcspec(self):
        specs = []
        for sound in self.sounds:
            spec = Sound.melspec(sound, self.datasets[0].getfs(), hop=1024, nfft=2048)
            specs.append(spec)
        return np.array(specs), np.array(self.labels), self.files
```

Figure 2.3: Partial code for class Calculate in calculate.py (not all methods are shown)

The data flow is setup in the assignment2.ipynb notebook. Before instantiating the data objects, the file names are retrieved and shuffled to get a random sequence for the training set. During dataset instantiation the labels are created as integers, good motor = 1, broken motor = 2 and heavy load motor = 3. Each label has its own instance.

```
# Paths to train data folders with sound
good_folder_train = "C:/python/IIA1420/assignment2/IDMT-ISA-ELECTRIC-ENGINE/train_cut/engine1_good/"
broken_folder_train = "C:/python/IIA1420/assignment2/IDMT-ISA-ELECTRIC-ENGINE/train_cut/engine2_broken/"
heavyload_folder_train = "C:/python/IIA1420/assignment2/IDMT-ISA-ELECTRIC-ENGINE/train_cut/engine3_heavyload/"
```

```
# Get list of files in directories
good_files_train = os.listdir(good_folder_train)
broken_files_train = os.listdir(broken_folder_train)
heavyload_files_train = os.listdir(heavyload_folder_train)

good_files_train = sorted(good_files_train, key=lambda item : int(item.split('_')[1].split('.')[0]))
broken_files_train = sorted(broken_files_train, key=lambda item : int(item.split('_')[1].split('.')[0]))
heavyload_files_train = sorted(heavyload_files_train, key=lambda item : int(item.split('_')[1].split('.')[0]))

print('Good files train: ', good_files_train[0:5])
print('Broken files train: ', broken_files_train[0:5])
print('Heavyload files train: ', heavyload_files_train[0:5])
```

```
Good files train:  ['pure_0.wav', 'pure_1.wav', 'pure_2.wav', 'pure_3.wav', 'pure_4.wav']
Broken files train:  ['pure_0.wav', 'pure_1.wav', 'pure_2.wav', 'pure_3.wav', 'pure_4.wav']
Heavyload files train:  ['pure_0.wav', 'pure_1.wav', 'pure_2.wav', 'pure_3.wav', 'pure_4.wav']
```

```
# Get random number of samples from train set
#random.seed(123)
ngtr = len(good_files_train) # Using full length to get all samples
nbtr = len(broken_files_train)
nhtr = len(heavyload_files_train)

good_files_train_samp = random.sample(good_files_train, ngtr)
broken_files_train_samp = random.sample(broken_files_train, nbtr)
heavyload_files_train_samp = random.sample(heavyload_files_train, nhtr)

print('Good files train sample: ', good_files_train_samp[0:5])
print('Broken files train sample: ', broken_files_train_samp[0:5])
print('Heavyload files train sample: ', heavyload_files_train_samp[0:5])
```

```
Good files train sample:  ['pure_1.wav', 'pure_20.wav', 'pure_4.wav', 'pure_54.wav', 'pure_35.wav']
Broken files train sample:  ['pure_59.wav', 'pure_16.wav', 'pure_57.wav', 'pure_77.wav', 'pure_46.wav']
Heavyload files train sample:  ['pure_35.wav', 'pure_37.wav', 'pure_57.wav', 'pure_90.wav', 'pure_17.wav']
```

```
# Initiate the dataset objects for training with links to files, folders and labels
# Labels are defined as numbers 1 = good, 2 = broken and 3 = heavy load
good_train = Dataset(good_files_train_samp, good_folder_train, fs, np.ones(ngtr).astype(int))
broken_train = Dataset(broken_files_train_samp, broken_folder_train, fs, 2*np.ones(nbtr).astype(int))
heavyload_train = Dataset(heavyload_files_train_samp, heavyload_folder_train, fs, 3*np.ones(nhtr).astype(int))
```

```
# Combine the training data sets and calculate the sound arrays for each file
data_train = Calculate([good_train, broken_train, heavyload_train])
```

```
# Use method to get the all sound files and corresponding labels and filenames
# These sound files will be used to generate the additional calculations
train_sound, train_label, train_file = data_train.getsound()
```

Figure 2.4: Filenames and instantiation of train data objects in assignment2.ipynb

The same method is used for test data, as it has the same predefined labels and folder structure (note that filenames are more diverse as it has the different noise cases). Another difference in the test data model is the large number of files. To deal with this the test set was split into a main test set, and an additional “rest” test dataset. This rest contain the greatest number of files and was used as extra test of best model in the end of the notebook. Note file names are shuffled before split into test and the rest dataset.

```
# TEST DATA PROCESSING
```

```
# Paths to test data folders with sound
good_folder_test = "C:/python/IIA1420/assignment2/IDMT-ISA-ELECTRIC-ENGINE/test_cut/engine1_good/"
broken_folder_test = "C:/python/IIA1420/assignment2/IDMT-ISA-ELECTRIC-ENGINE/test_cut/engine2_broken/"
heavyload_folder_test = "C:/python/IIA1420/assignment2/IDMT-ISA-ELECTRIC-ENGINE/test_cut/engine3_heavyload/"
```

```
# Get list of files in directories
good_files_test = os.listdir(good_folder_test)
broken_files_test = os.listdir(broken_folder_test)
heavyload_files_test = os.listdir(heavyload_folder_test)

good_files_test = sorted(good_files_test, key=lambda item : int(item.rsplit('_', 1)[1].split('.')[0]))
broken_files_test = sorted(broken_files_test, key=lambda item : int(item.rsplit('_', 1)[1].split('.')[0]))
heavyload_files_test = sorted(heavyload_files_test, key=lambda item : int(item.rsplit('_', 1)[1].split('.')[0]))

print('Good files test: ', good_files_test[0:5])
print('Broken files test: ', broken_files_test[0:5])
print('Heavyload files test: ', heavyload_files_test[0:5])
```

```
Good files test:  ['atmo_high_0.wav', 'atmo_low_0.wav', 'atmo_medium_0.wav', 'stresstest_0.wav', 'talking_1_0.wav']
Broken files test:  ['atmo_high_0.wav', 'atmo_low_0.wav', 'atmo_medium_0.wav', 'stresstest_0.wav', 'talking_1_0.wav']
Heavyload files test:  ['atmo_high_0.wav', 'atmo_low_0.wav', 'atmo_medium_0.wav', 'stresstest_0.wav', 'talking_1_0.wav']
```

```
# Get random number of samples from test set
#np.random.seed(123)
#random.seed(123)
all_good_files_test_samp = random.sample(good_files_test, int(len(good_files_test)))
all_broken_files_test_samp = random.sample(broken_files_test, int(len(broken_files_test)))
all_heavyload_files_test_samp = random.sample(heavyload_files_test, int(len(heavyload_files_test)))

good_files_test_samp = all_good_files_test_samp[0:int(len(good_files_test)/3)]
broken_files_test_samp = all_broken_files_test_samp[0:int(len(broken_files_test)/3)]
heavyload_files_test_samp = all_heavyload_files_test_samp[0:int(len(heavyload_files_test)/3)]

# These will be used as a final extra test at the end of notebook
good_files_test_rest = all_good_files_test_samp[int(len(good_files_test_samp)):]
broken_files_test_rest = all_broken_files_test_samp[int(len(broken_files_test_samp)):]
heavyload_files_test_rest = all_heavyload_files_test_samp[int(len(heavyload_files_test_samp)):]

ngte = len(good_files_test_samp)  # shorten the number of samples
nbte = len(broken_files_test_samp)
nhte = len(heavyload_files_test_samp)

print('Good files test sample: ', good_files_test_samp[0:5])
print('Broken files test sample: ', broken_files_test_samp[0:5])
print('Heavyload files test sample: ', heavyload_files_test_samp[0:5])
```

```
Good files test sample:  ['stresstest_2.wav', 'atmo_medium_1.wav', 'talking_3_18.wav', 'atmo_medium_60.wav', 'stresstest_38.wav']
Broken files test sample:  ['stresstest_38.wav', 'whitenoise_low_115.wav', 'stresstest_36.wav', 'talking_4_35.wav', 'whitenoise_low_9.wav']
Heavyload files test sample:  ['atmo_medium_19.wav', 'atmo_medium_46.wav', 'talking_2_28.wav', 'talking_4_68.wav', 'talking_3_69.wav']
```

```
# Check so files names split ok
print(len(good_files_test_samp), good_files_test_samp[-3:], good_files_test_samp[222])
print(len(good_files_test_rest), good_files_test_rest[:3], good_files_test_rest[-1])
print(len(all_good_files_test_samp), all_good_files_test_samp[220:225], all_good_files_test_samp[-1])
```

```
223 ['whitenoise_low_8.wav', 'talking_1_60.wav', 'talking_3_24.wav'] talking_3_24.wav
446 ['atmo_high_54.wav', 'atmo_low_64.wav', 'atmo_high_16.wav'] stresstest_28.wav
669 ['whitenoise_low_8.wav', 'talking_1_60.wav', 'talking_3_24.wav', 'atmo_high_54.wav', 'atmo_low_64.wav'] stresstest_28.wav
```

```
# Define the datasets for testing with links to files, folders and labels
# Labels are defined as numbers 1 = good, 2 = broken and 3 = heavy load
good_test = Dataset(good_files_test_samp, good_folder_test, fs, np.ones(ngte).astype(int))
broken_test = Dataset(broken_files_test_samp, broken_folder_test, fs, 2*np.ones(nbte).astype(int))
heavyload_test = Dataset(heavyload_files_test_samp, heavyload_folder_test, fs, 3*np.ones(nhte).astype(int))
```

```
# Combine the testing data sets and calculate sound arrays
data_test = Calculate([good_test, broken_test, heavyload_test])
```

```
# Use method to get the all sound files and corresponding labels and filenames
# These sound files will be used to generate the additional calculations
test_sound, test_label, test_files = data_test.getsound()
```

Figure 2.5: Filenames and instantiation of test data objects in assignment2.ipynb

2.2 Data exploration

Raw sound files are difficult to analyze. They contain so much data that is similar, each 3 second clip has 132300 samples. However, by looking at the sound files there are still some distinct differences on the training data, when comparing the good, broken and heavy load.

```
# Plot a single good, broken and heavyload sound
plotdata = [train_sound[0], train_sound[150], train_sound[260]]
plotlabels = [train_label[0], train_label[150], train_label[260]]
plotfiles = [train_file[0], train_file[150], train_file[260]]

Sound.plotmultsound(plotdata, fs, plotfiles, plotlabels )
```

Figure

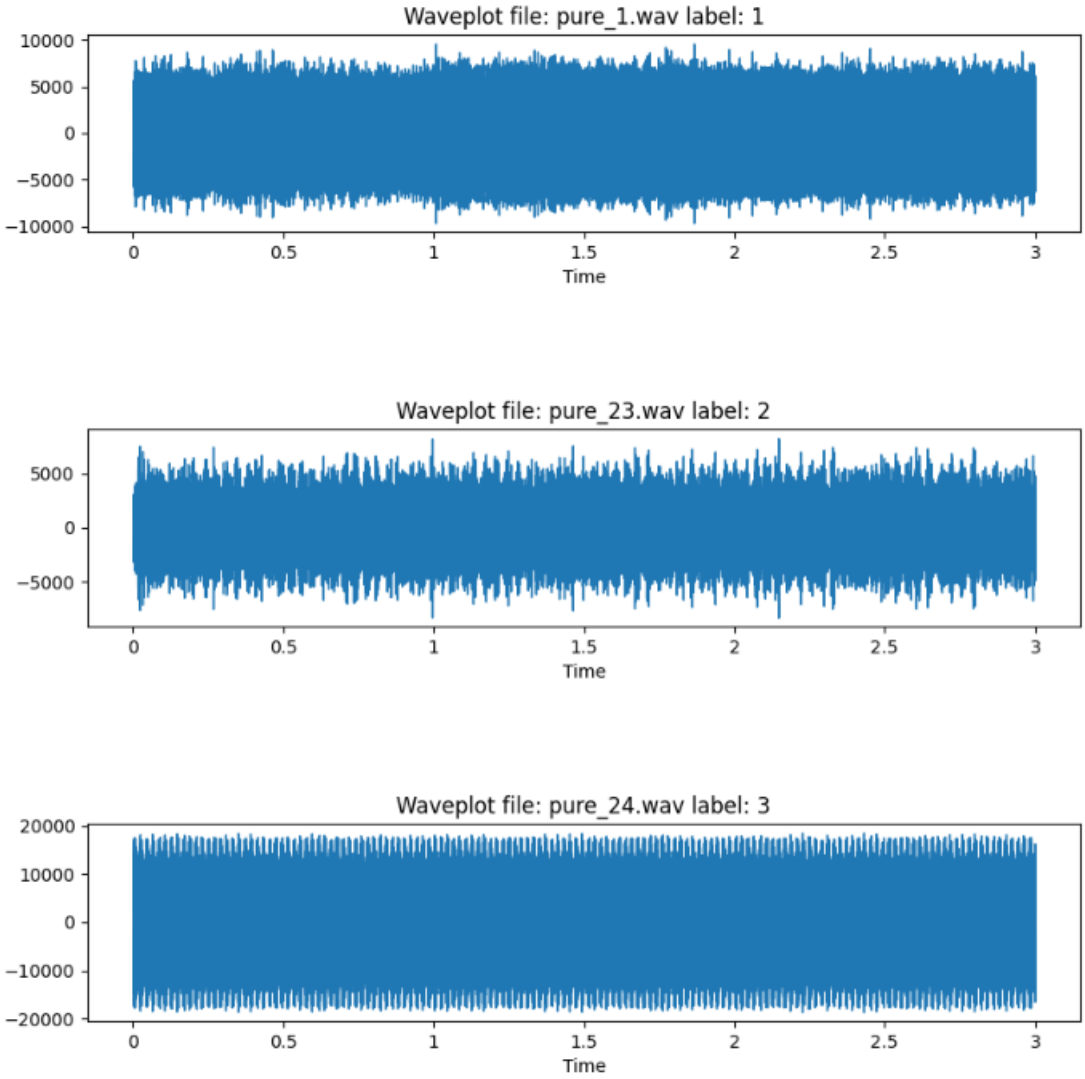

Figure 2.6: Sound file plots for 3 different labels from assignment2.ipynb

These differences become even more distinct when looking at the RMS (root mean square) and the Crest Factor. RMS is a measure of the effective value of a signal, could be seen as a representation of the total energy of the signal, not a point value, see [6] for more.

```
# Calculate RMS for the sound files
train_rms, train_label, train_file = data_train.calcrms()
print(train_rms.shape)
print(train_rms[sl])
print(train_label[sl])
print(train_file[sl])
```

```
(357,)
[1819.64016263 1893.97204848]
[2 2]
['pure_89.wav', 'pure_2.wav']
```

```
# Plot training data RMS values for each file
Sound.plotsimpleall(train_rms, 'RMS for training data', 'Wav sample', 'RMS')
```

Figure

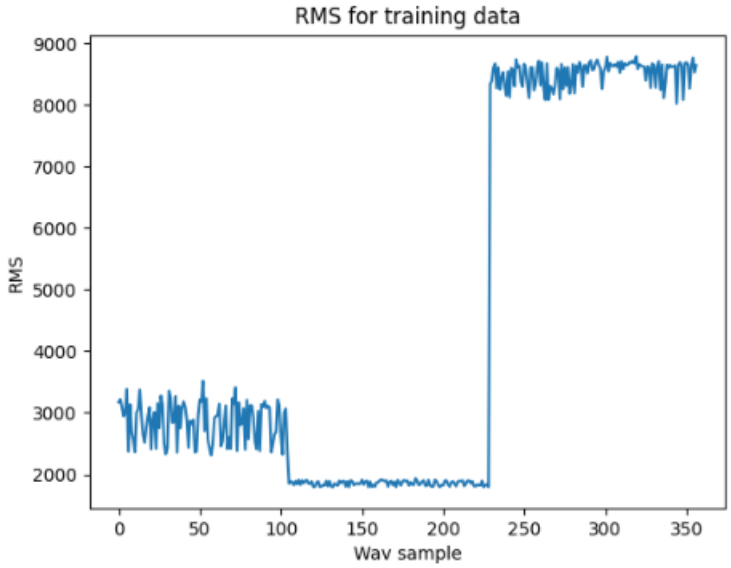

Figure 2.6: RMS plot for all samples in training set assignment2.ipynb

In a similar way the Crest Factor can be calculated as a new feature based on each sound file. The Crest factor is the ratio between peak amplitude and the RMS. This also seem to differ between the different labels, however there is some disturbance around sample file index 250.

```
train_crest, train_label, train_file = data_train.calccrest()
print(train_crest.shape)
print(train_crest[sl])
print(train_label[sl])
print(train_file[sl])
```

```
(357,)
[4.8899778  4.59246482]
[2 2]
['pure_89.wav', 'pure_2.wav']
```

```
# Plot training data crest factor values for each file
Sound.plotsimpleall(train_crest, 'Crest factor for training data', 'Wav sample', 'Crest Factor')
```

Figure

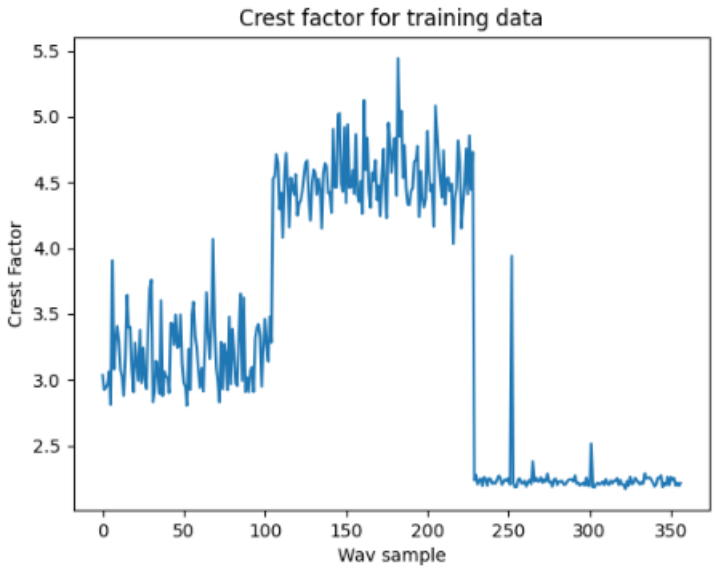

Figure 2.7: Crest factor plot for all samples in training set assignment2.ipynb

Next feature to evaluate is the DFT that can be calculated by the FFT, refer to theory in [1]. The DFT is useful to get an overview of the frequency content of the data. Below are plots of a single sample from each of the labels. There are distinct differences at the lower part of the frequency axis. Both in number of different frequencies present and the magnitude. The assignment2.ipynb also contain some attempts to filter the DFT. This was successful, but not used in the current workflow, so no more detail of that is presented. For future use.

```
# Plot good motor dft
Sound.plotdft(train_fft[0], fs, train_file[0], train_label[0])
```

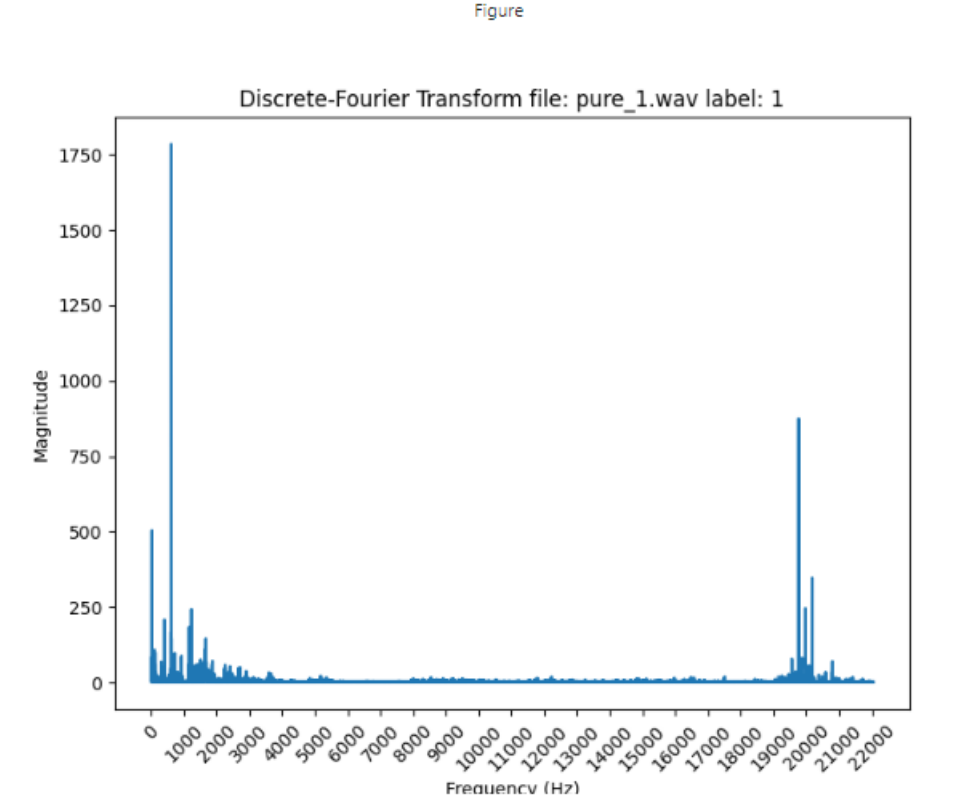

Figure 2.8: DFT plot for good motor sample from assignment2.ipynb

```
# Plot broken motor dft
Sound.plotdft(train_fft[150], fs, train_file[150], train_label[150])
```

Figure

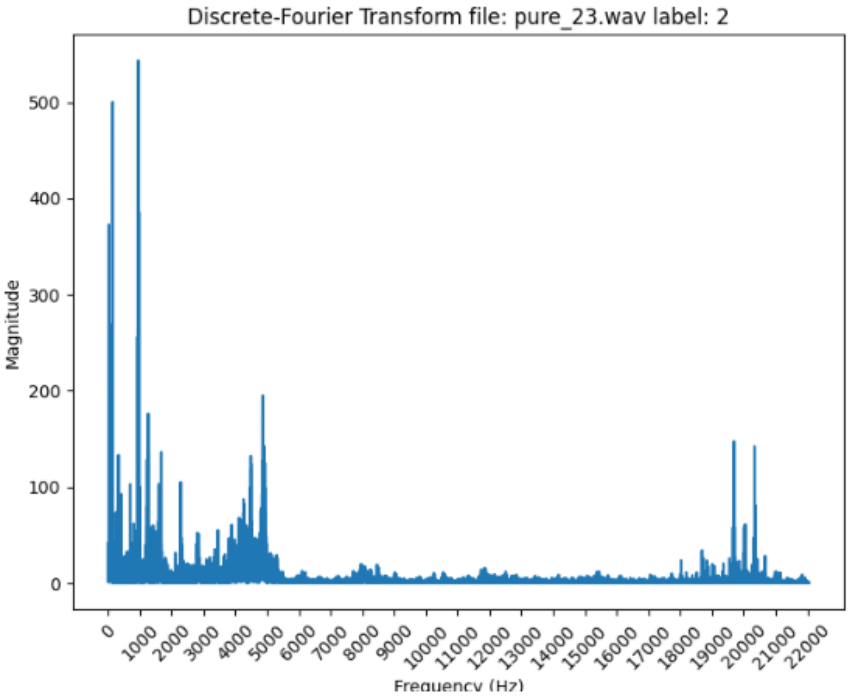

Figure 2.9: DFT plot for broken motor sample from assignment2.ipynb

```
# Plot heavy load motor dft
Sound.plotdft(train_fft[260], fs, train_file[260], train_label[260])
```

Figure

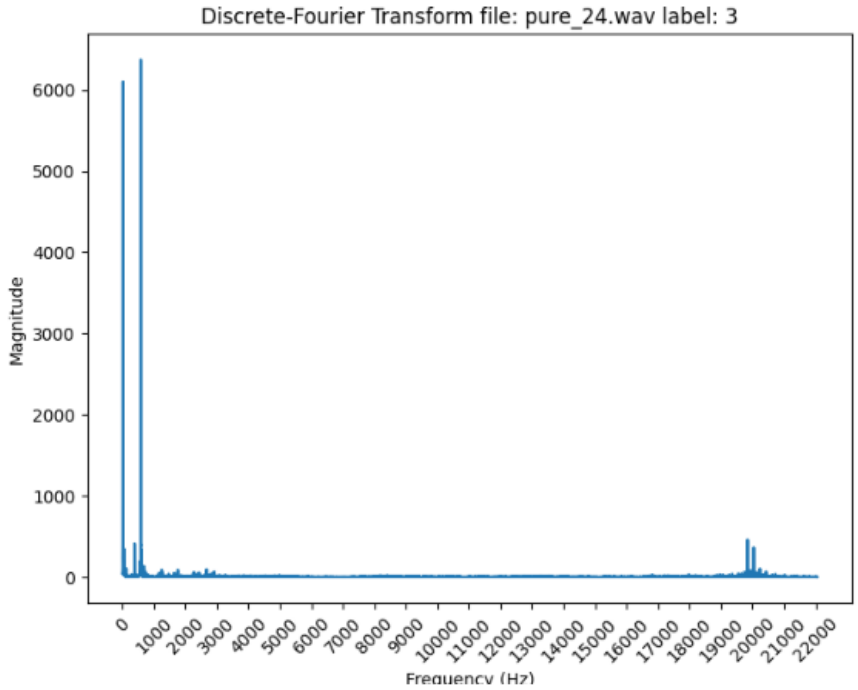

Figure 2.10: DFT plot for heavy load motor sample from assignment2.ipynb

The final feature to be calculated is a Mel Spectrogram. The idea behind this spectrogram is to change the scale on y axis (the frequency) and the intensity (the amplitude, here represented by color scale). Commonly used in processing of sound to adjust the spectrogram for what we humans can perceive, it might not be the best option for electrical motor sounds. Still, this was chosen as it seemed interesting enough. A couple of good examples were found online in [3] and [4]. The Librosa package was chosen as it had good examples and easy to use. Follows three Mel Spectrograms for three samples from training set. The hop length, i.e. how many samples to jump between each spectrogram calculation was set to 1024 and the window (nfft length) was set to 2048. These are powers of 2 and results in more compact 128x130 arrays. This is to a cost of information loss, but would require more computing power too.

```
# Plot good load motor mel spectrogram
Sound.plotspec(train_spec[0], fs, 1024, train_file[0], train_label[0])
```

Figure

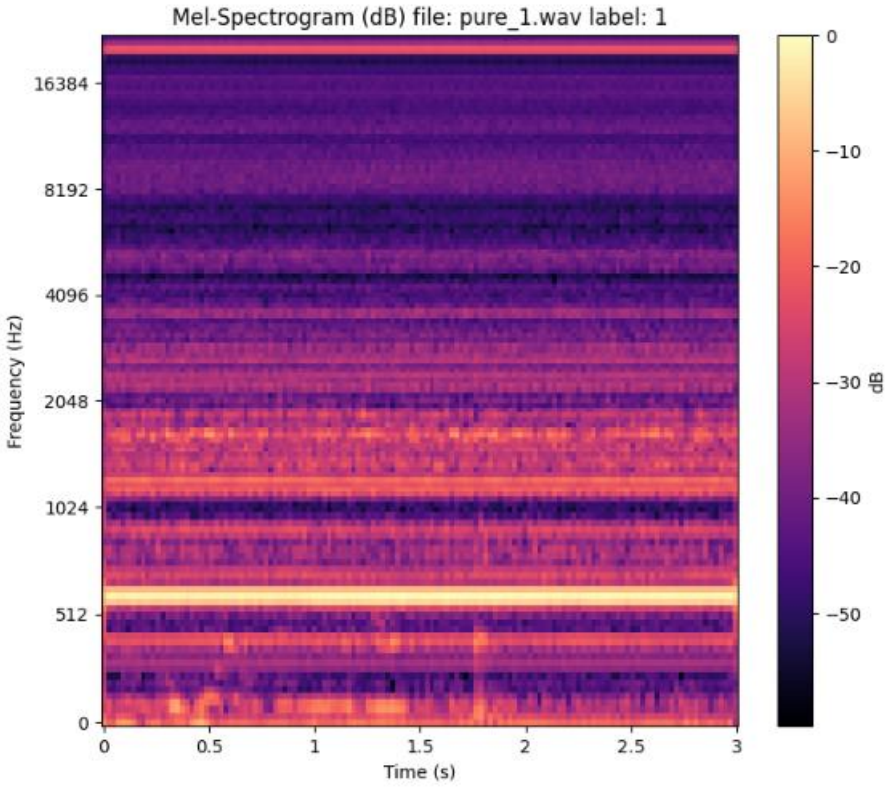

Figure 2.11: Mel spectrogram for good motor sample from assignment2.ipynb

```
# Plot broken load motor mel spectrogram
Sound.plotspec(train_spec[150], fs, 1024, train_file[150], train_label[150])
```

Figure

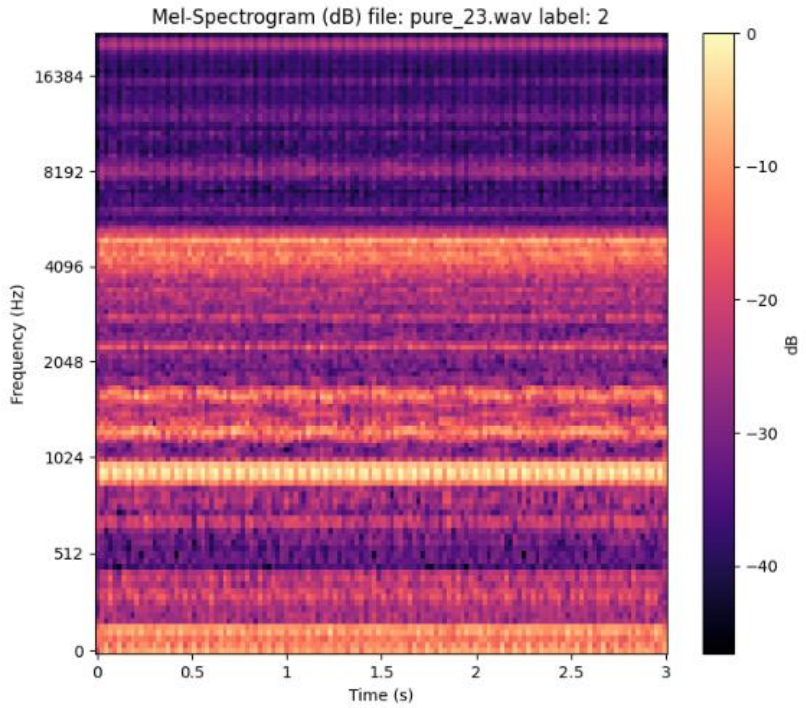

Figure 2.12: Mel spectrogram for broken motor sample from assignment2.ipynb

```
# Plot heavy load motor mel spectrogram
Sound.plotspec(train_spec[260], fs, 1024, train_file[260], train_label[260])
```

Figure

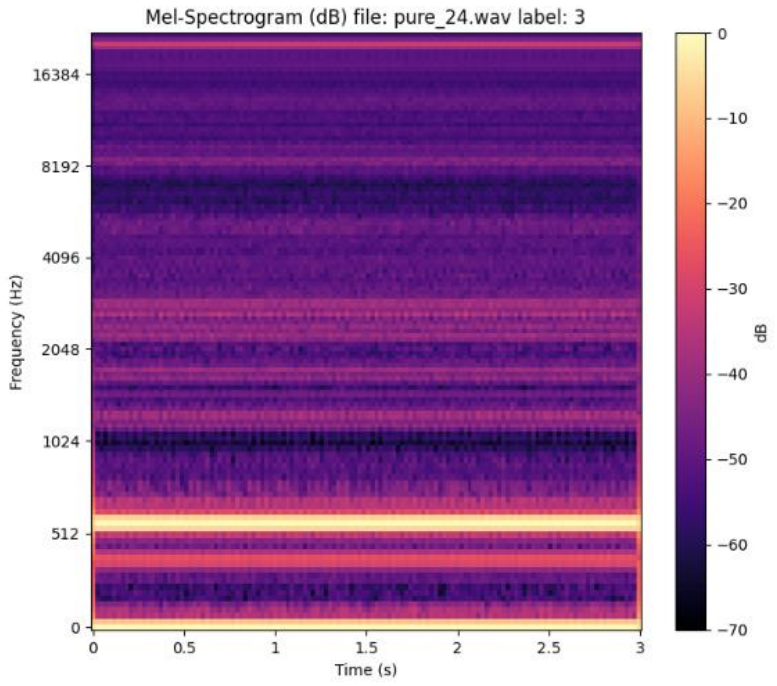

Figure 2.13: Mel spectrogram for heavy load motor sample from assignment2.ipynb

By using spectrogram, the classification problem was turned into one the reassembles image classification (such as MNIST) example in [2]. The spectrogram can be seen as a 2D image with variable color intensity. Not much work was put into preprocessing the data, it was calculated and put into training of the classifier. Normalization and flattening were applied to the training data. In addition to the Mel Spectrogram, the RMS was also chosen for training ML models. Training data was also shuffled so the labels appear in random order.

```
# PREPARE TRAINING DATA FOR MODEL
```

```
# Normalize rms training data
train_rmsnorm = normalize(train_rms.reshape(1,-1), norm='max')
# Make random selection among all
train_rmsnorm_shuff, train_label_rms_shuff, train_file_rms_shuff = shuffle(train_rmsnorm[0], train_label, train_file)#, random_state=123)
print(train_rmsnorm.shape)
print(train_rmsnorm[0][sl])
print(train_rmsnorm_shuff[sl])
print(train_label[sl])
print(train_label_rms_shuff[sl])
print(train_file[sl])
print(train_file_rms_shuff[sl])
```

```
(1, 357)
[0.20711661 0.21557728]
[0.35566243 0.21509183]
[2 2]
[1 2]
['pure_89.wav', 'pure_2.wav']
['pure_37.wav', 'pure_14.wav']
```

```
# Reshape the spectogram array to 1D
train_spec1d = train_spec.reshape(len(train_spec), 128*130)
# Normalize spectrogram data
train_spec1dnorm = normalize(train_spec1d, norm='max')
# Make random selection among all
train_spec1dnorm_shuff, train_label_spec_shuff, train_file_spec_shuff = shuffle(train_spec1dnorm, train_label, train_file)#, random_state=123)
print(train_spec1dnorm_shuff.shape)
print(train_spec1dnorm[sl])
print(train_spec1dnorm_shuff[sl])
print(train_label[sl])
print(train_label_spec_shuff[sl])
print(train_file[sl])
print(train_file_spec_shuff[sl])
```

```
(357, 16640)
[[-0.24843584 -0.21312936 -0.15407363 ... -0.8760817  -0.76598436
  -0.9137184 ]
 [-0.23593366 -0.26152813 -0.22377905 ... -0.82631874 -0.74925154
  -0.9003624 ]]
[[-0.04528823 -0.00655286 -0.00691354 ... -0.71998215 -0.72185224
  -0.7073244 ]
 [-0.10436007 -0.00541328 -0.00148081 ... -0.75968784 -0.76573247
  -0.7475652 ]]
[2 2]
[3 3]
['pure_89.wav', 'pure_2.wav']
['pure_40.wav', 'pure_110.wav']
```

Figure 2.14: Training data preparation from assignment2.ipynb

The test data was also prepared in as similar way, but without the shuffling of order. See assignment2.ipynb for details.

2.3 Machine Learning

Much time was spent on the data modelling, so the actual machine learning part got a bit reduced, compared to the initial plan. For example, the feature modelling is missing. There are some examples for using time and frequency masks as regularization. Similarily for RMS, had some ideas to use filters to extract the core characteristics of the pure signals, before applying RMS (or Crest Factor) calculations. This would have been interesting to try out but was left out here.

The amount of training data (viewed as number of wav files, not individual samples) is not so large. So, it was not clear if the cross validation and hyper parameter tuning would make so much sense here. It was also left out (however, some cross validation prediction was made).

Based on examples in [2] and the KNN (K-Nearest Neighbors) and SGD (steepest gradient decent) was chosen

2.3.1 KNN Classifier Mel Spectrogram

```
# TRAINING MODEL
```

```
#Imports
from sklearn.linear_model import SGDClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix
from sklearn import metrics as ms
from sklearn.metrics import accuracy_score, classification_report
```

```
# Train and test mel spectorgram with KNN classifier
```

```
# Define and train the model for mel spectrogram
knn_model_mel = KNeighborsClassifier(n_neighbors=3)
knn_model_mel.fit(train_spec1dnorm_shuff, train_label_spec_shuff)
```

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

```
# Cross validation predict (with an untrained model)
knn_model2 = KNeighborsClassifier(n_neighbors=3)
y_train_pred_knn = cross_val_predict(knn_model2, train_spec1dnorm_shuff, train_label_spec_shuff, cv=3)
cm = confusion_matrix(train_label_spec_shuff, y_train_pred_knn)
cm
```

```
array([[105,   0,   0],
       [  0, 124,   0],
       [  0,   0, 128]], dtype=int64)
```

Figure 2.15: KNN classifier training from assignment2.ipynb

The result of cross validation prediction (using the training data only) shows a perfect outcome. This is unclear, either the pure data is so distinct, so the model works well, or there are some issues with the training data setup for this cross validation (too few samples of each type maybe). This is something that will repeat for all classifiers.

The results of the KNN classifier are good. Now the test set is ca 1/3 of the test data (refer to comment in 2.1). The model was set to n=3 and it did not change much when trying other nearby n values (not tuned properly). We get a accuracy around 96% which is high. It is not clear if this is abnormal due to overfitting or other problem in the model. It would require more work to investigate the results and make a conclusion.

```
# Check accuracy
y_pred_knn_mel = knn_model_mel.predict(test_spec1dnorm)
print("Accuracy of model = %2f%%" % (accuracy_score(test_label, y_pred_knn_mel)*100))
```

```
Accuracy of model = 96.285290%
```

```
print(classification_report(test_label, y_pred_knn_mel, target_names=['good', 'broken', 'heavyload']))
```

```
              precision    recall  f1-score   support

        good       1.00      1.00      1.00       223
      broken       1.00      0.89      0.94       221
   heavyload       0.90      1.00      0.95       229

    accuracy                           0.96       673
   macro avg       0.97      0.96      0.96       673
weighted avg       0.97      0.96      0.96       673
```

```
# Test with the random samples
#np.random.seed(123)
n_test=400
idxs_test = np.random.choice(len(test_spec1dnorm), n_test, replace=False)
idxs_test
y_samples_knn_mel = knn_model_mel.predict(test_spec1dnorm[idxs_test])
print('Predicted with KNN: \n', y_samples_knn_mel, '\nReal label: \n', test_label[idxs_test])
print('all equal? ' , np.array_equal(y_samples_knn_mel, test_label[idxs_test]))
print('Fraction equal: ', np.sum(y_samples_knn_mel==test_label[idxs_test])/len(test_label[idxs_test]))
```

```
Predicted with KNN:
 [1 1 3 3 1 3 1 1 1 1 1 3 3 3 3 2 1 3 1 3 1 2 1 3 2 2 3 1 1 1 3 1 1 2 2 3 3
 2 3 2 1 1 3 2 1 3 3 3 2 2 2 1 3 2 1 2 2 2 3 2 2 3 3 3 2 2 1 3 1 3 2 1 3 1
 3 1 3 1 3 3 2 2 1 1 3 3 3 3 2 3 2 3 3 2 3 2 3 1 1 3 3 3 3 2 3 3 3 3 1 3 1
 3 2 1 3 3 2 2 1 2 2 1 1 1 3 1 3 1 3 3 3 1 2 1 2 3 1 3 1 2 3 3 3 3 2 1 2 3
 2 3 1 2 2 2 3 1 1 1 3 3 2 3 2 3 3 3 1 2 1 3 2 3 1 3 1 1 1 1 3 3 3 1 1 1 2
 2 3 1 2 3 1 3 3 3 1 3 1 3 3 3 1 1 3 3 2 1 2 1 1 3 1 3 3 2 2 3 3 2 3 2 1 2
 2 1 3 2 2 1 1 2 1 1 2 2 2 2 1 1 2 1 1 2 2 1 3 2 2 1 3 2 2 1 3 2 2 1 3 1 2 3 2 3 2
 3 1 3 3 2 2 1 3 3 2 1 2 1 2 2 1 3 3 2 1 2 3 2 1 2 2 1 2 1 3 3 2 2 1 2 2 3
 3 2 1 1 1 1 2 3 1 1 2 1 3 1 1 2 1 3 1 3 1 3 2 3 3 2 1 2 1 3 2 1 3 1 2 3 3
 2 2 2 3 3 2 3 3 1 3 3 3 3 3 2 1 1 3 1 3 3 3 1 2 2 2 2 3 1 3 3 2 3 3 2 2 3
 3 3 1 2 1 3 1 2 3 2 3 1 1 2 1 3 1 2 1 3 3 2 3 1 2 1 3 2 3 2]
Real label:
 [1 1 3 3 1 3 1 1 1 1 1 3 3 3 3 2 1 3 1 3 1 2 1 2 2 2 3 1 1 1 3 1 1 2 2 3 3
 2 3 2 1 1 3 2 1 3 3 3 2 2 2 1 3 2 1 2 2 2 3 2 2 3 3 3 2 2 1 3 1 3 2 1 3 1
 3 1 3 1 3 3 2 2 1 1 3 3 3 3 2 2 2 3 3 2 3 2 3 1 1 2 3 3 3 2 3 2 3 3 1 3 1
 2 2 1 3 3 2 2 1 2 2 1 1 1 3 1 3 1 2 3 3 1 2 1 2 3 1 2 1 2 3 3 3 3 2 1 2 3
 2 3 1 2 2 2 3 1 1 1 3 3 2 3 2 3 2 3 1 2 1 3 2 3 1 3 1 1 1 1 3 3 3 1 1 1 2
 2 3 1 2 3 1 3 3 3 1 3 1 3 3 3 1 1 3 3 2 1 2 1 1 3 1 3 2 2 2 3 3 2 3 2 1 2
 2 1 3 2 2 1 1 2 1 1 2 2 2 2 1 1 2 1 1 2 2 1 3 2 2 1 3 2 2 1 3 1 2 3 2 3 2
 3 1 3 3 2 2 1 3 3 2 1 2 1 2 2 1 3 3 2 1 2 3 2 1 2 2 1 2 1 3 3 2 2 1 2 2 2
 3 2 1 1 1 1 2 3 1 1 2 1 2 1 1 2 1 3 1 3 1 3 2 2 3 2 1 2 1 3 2 1 3 1 2 3 3
 2 2 2 3 3 2 3 3 1 3 3 3 3 3 2 1 1 3 1 2 3 3 1 2 2 2 2 3 1 3 3 2 3 3 2 2 3
 3 3 1 2 1 2 1 2 3 2 3 1 1 2 1 3 1 2 1 2 3 2 3 1 2 1 3 2 2 2]
all equal?  False
Fraction equal:  0.96
```

Figure 2.16: KNN classifier results from assignment2.ipynb

The evaluation contains a built-in classification report from Sci-kit Learn and a manual test of generating several hundred random indexes from test set and applying the trained model to it. Results are similar.

2.3.2 KNN Classifier RMS

```
# Train and test using RMS with KNN classifier
```

```
# # Define and train the model for RMS
knn_model_rms = KNeighborsClassifier(n_neighbors=3)
knn_model_rms.fit(train_rmsnorm_shuff.reshape(-1,1), train_label_rms_shuff)
```

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

```
# Cross validation predict (with an untrained model)
knn_model3 = KNeighborsClassifier(n_neighbors=3)
y_train_pred_knn_rms = cross_val_predict(knn_model3, train_rmsnorm_shuff.reshape(-1,1), train_label_rms_shuff, cv=2)
cm = confusion_matrix(train_label_rms_shuff, y_train_pred_knn_rms)
cm
```

```
array([[105,   0,   0],
       [  0, 124,   0],
       [  0,   0, 128]], dtype=int64)
```

Figure 2.17: KNN RMS classifier training from assignment2.ipynb

The result of cross validation prediction (using the training data only) shows again a perfect outcome.

The results of the KNN classifier very poor for the RMS feature. It only show around 30% accuracy.

```
# Check accuracy
y_pred_knn_rms = knn_model_rms.predict(test_rmsnorm.reshape(-1,1))
print("Accuracy of model = %2f%%" % (accuracy_score(test_label, y_pred_knn_rms)*100))
```

```
Accuracy of model = 31.649331%
```

```
print(classification_report(test_label, y_pred_knn_rms, target_names=['good', 'broken', 'heavyload']))
```

```
              precision    recall  f1-score   support

        good       0.03      0.03      0.03       223
      broken       0.49      0.92      0.64       221
  heavyload       0.17      0.01      0.02       229

    accuracy                           0.32       673
   macro avg       0.23      0.32      0.23       673
weighted avg       0.23      0.32      0.23       673
```

```
# Test with the same random samples as previously
y_samples_knn_rms = knn_model_rms.predict(test_rmsnorm.reshape(-1,1)[idxs_test])
print('Predicted with KNN: \n', y_samples_knn_rms, '\nReal label: \n', test_label[idxs_test])
print('all equal? ' , np.array_equal(y_samples_knn_rms, test_label[idxs_test]))
print('Fraction equal: ', np.sum(y_samples_knn_rms==test_label[idxs_test])/len(test_label[idxs_test]))
```

```
Predicted with KNN:
 [2 2 1 1 2 1 2 2 2 2 2 1 1 1 1 2 2 1 2 1 2 2 2 2 2 2 1 2 2 2 1 2 2 2 1 1 1
 2 1 2 2 2 1 2 3 1 1 1 1 2 2 2 1 2 2 2 2 2 2 1 1 2 1 1 1 2 2 2 1 2 1 2 1 1 1
 1 2 1 2 1 1 2 2 2 2 2 1 1 1 1 2 2 2 1 1 2 1 2 1 2 2 2 2 1 1 1 2 1 2 1 1 2 1 2
 2 3 2 1 1 2 1 2 2 3 2 2 2 3 2 1 2 2 1 1 2 2 2 2 1 2 2 2 2 2 1 1 1 1 2 2 2 1
 2 1 2 2 2 2 2 1 2 2 2 2 1 1 2 2 1 2 1 2 1 1 1 2 2 2 2 2 1 1 1 2 2 2 2 2 2
 2 1 2 2 1 2 2 1 1 1 2 1 2 2 1 1 1 2 2 2 1 1 3 2 2 2 2 1 2 1 2 2 2 2 1 1 2 3 2 2 2
 2 2 1 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 1 2 2 2 1 2 2 2 2 1 2 2 1 2 1 2
 1 2 1 1 2 2 2 1 1 2 2 2 2 2 2 2 2 1 1 2 2 2 1 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2
 1 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 1 2 1 2 1 2 2 1 2 2 2 3 1 2 2 1 2 2 1 1
 2 2 2 1 1 2 1 1 2 1 1 1 1 1 2 2 2 1 2 2 1 1 2 2 2 2 2 2 1 2 1 1 2 1 1 2 2 1
 1 1 2 2 2 2 2 2 2 1 1 1 2 2 2 2 1 2 2 2 2 1 1 1 1 2 2 2 1 2 2 2]
Real label:
 [1 1 3 3 1 3 1 1 1 1 1 3 3 3 3 2 1 3 1 3 1 2 1 2 2 2 3 1 1 1 3 1 1 2 2 3 3
 2 3 2 1 1 3 2 1 3 3 3 2 2 2 1 3 2 1 2 2 2 3 2 2 3 3 3 2 2 1 3 1 3 2 1 3 1
 3 1 3 1 3 3 2 2 1 1 3 3 3 3 2 2 2 3 3 2 3 2 3 1 1 2 3 3 3 2 3 2 3 3 1 3 1
 2 2 1 3 3 2 2 1 2 2 1 1 1 3 1 3 1 2 3 3 1 2 1 2 3 1 2 1 2 3 3 3 3 2 1 2 3
 2 3 1 2 2 2 3 1 1 1 3 3 2 3 2 3 2 3 1 2 1 3 2 3 1 3 1 1 1 1 3 3 3 1 1 1 2
 2 3 1 2 3 1 3 3 3 1 3 1 3 3 3 1 1 3 3 2 1 2 1 1 3 1 3 2 2 2 3 3 2 3 2 1 2
 2 1 3 2 2 1 1 2 1 1 2 2 2 2 1 1 2 1 1 2 2 1 3 2 2 1 3 2 2 1 3 1 2 3 2 3 2
 3 1 3 3 2 2 1 3 3 2 1 2 1 2 2 1 3 3 2 1 2 3 2 1 2 2 1 2 1 3 3 2 2 1 2 2 2
 3 2 1 1 1 1 2 3 1 1 2 1 2 1 1 2 1 3 1 3 1 3 2 2 3 2 1 2 1 3 2 1 3 1 2 3 3
 2 2 2 3 3 2 3 3 1 3 3 3 3 3 2 1 1 3 1 2 3 3 1 2 2 2 2 3 1 3 3 2 3 3 2 2 3
 3 3 1 2 1 2 1 2 3 2 3 1 1 2 1 3 1 2 1 2 3 2 3 1 2 1 3 2 2 2]
all equal?  False
Fraction equal:  0.3225
```

Figure 2.18: KNN RMS classifier results from assignment2.ipynb

By looking at the training data RMS calculation it is clear the pure RMS is very different from the RMS in the noisy test data. Based on this result, no more training is done with the RMS feature. As mentioned, there are probably ways to pre-process data before using RMS, the RMS still has the advantage that it is just a 1D number, not a large 2D or 3D array. Similar situation for the Crest Factor, see Appendix 1 for a plot.

```
# Plot test data RMS values for each file
Sound.plotsimpleall(test_rms, 'RMS for test data', 'Wav sample', 'RMS')
```

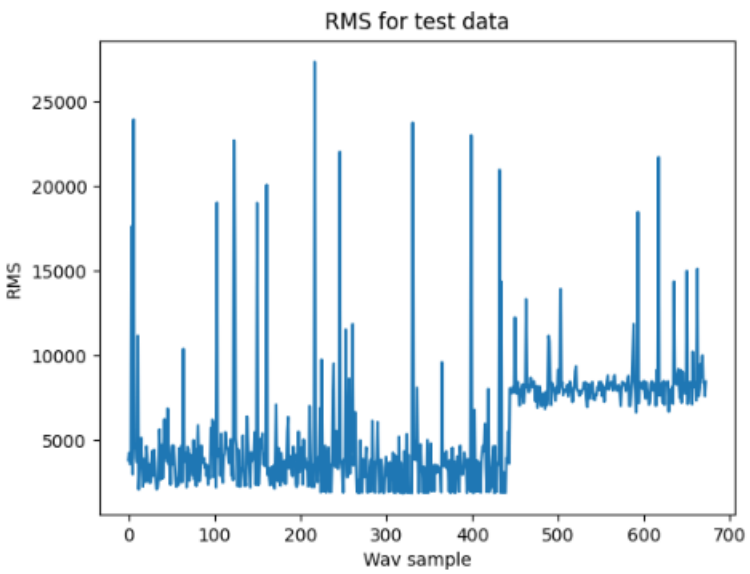

Figure 2.19: RMS for test data from assignment2.ipynb

2.3.3 SGD Classifier Mel Spectrogram

```
# Train and test mel spectograms with SGD classifier

# Define and train the model for mel spectrogram
sgd_model_mel = SGDClassifier(loss='log_loss', alpha=0.8, max_iter=1000)#, random_state=123
sgd_model_mel.fit(train_spec1dnorm_shuff, train_label_spec_shuff)
```

```
SGDClassifier
SGDClassifier(alpha=0.8, loss='log_loss')
```

```
# Cross validation predict (with an untrained model)
sgd_model2 = SGDClassifier(loss='log_loss', alpha=0.1, max_iter=1000)#, random_state=123)
y_train_pred_sgd_mel = cross_val_predict(sgd_model2, train_spec1dnorm_shuff, train_label_spec_shuff, cv=2)
cm = confusion_matrix(train_label_spec_shuff, y_train_pred_sgd_mel)
cm
```

```
array([[105,   0,   0],
       [  0, 124,   0],
       [  0,   0, 128]], dtype=int64)
```

Figure 2.20: SGD classifier training from assignment2.ipynb

The result of cross validation prediction (using the training data only) shows again a perfect outcome. See comments in 2.3.1.

The results of the SGN classifier are very good. The test set is ca 1/3 of the test data (refer to comment in 2.1). The model was set to alpha=0.8 and log_loss (not tuned properly, just trying some values). We get an accuracy around 99% which is very high. It is not clear if this is abnormal due to overfitting or other problem in the model. It would require more work to investigate the results and make a conclusion.

```
# Check accuracy
y_pred_sgd_mel =sgd_model_mel.predict(test_spec1dnorm)
print("Accuracy of model = %2f%%" % (accuracy_score(test_label, y_pred_sgd_mel )*100))
```

```
Accuracy of model = 98.514116%
```

```
print(classification_report(test_label, y_pred_sgd_mel, target_names=['good', 'broken', 'heavyload']))
```

```
              precision    recall  f1-score   support

        good       1.00      0.96      0.98       223
      broken       1.00      1.00      1.00       221
   heavyload       0.96      1.00      0.98       229

    accuracy                           0.99       673
   macro avg       0.99      0.99      0.99       673
weighted avg       0.99      0.99      0.99       673
```

```
# Test with the random samples
y_samples_sgd_mel = sgd_model_mel.predict(test_spec1dnorm[idxs_test])
print('Predicted with SDG: \n', y_samples_sgd_mel, '\nReal label: \n', test_label[idxs_test])
print('all equal? ' , np.array_equal(y_samples_sgd_mel, test_label[idxs_test]))
print('Fraction equal: ', np.sum(y_samples_sgd_mel==test_label[idxs_test])/len(test_label[idxs_test]))
```

```
Predicted with SDG:
 [1 1 3 3 1 3 1 1 1 1 1 3 3 3 3 2 1 3 1 3 1 2 1 2 2 2 3 1 1 1 3 1 1 2 2 3 3
 2 3 2 1 1 3 2 3 3 3 3 2 2 2 1 3 2 1 2 2 2 3 2 2 3 3 3 2 2 1 3 1 3 2 3 3 3
 3 1 3 1 3 3 2 2 1 1 3 3 3 3 2 2 2 3 3 2 3 2 3 1 1 2 3 3 3 2 3 2 3 3 1 3 1
 2 2 1 3 3 2 2 1 2 2 1 1 1 3 1 3 1 2 3 3 1 2 1 2 3 1 2 1 2 3 3 3 3 2 1 2 3
 2 3 1 2 2 2 3 1 1 1 3 3 2 3 2 3 2 3 1 2 1 3 2 3 1 3 1 1 1 1 3 3 3 1 1 1 2
 2 3 1 2 3 1 3 3 3 1 3 3 3 1 1 3 3 2 1 2 1 1 3 1 3 2 2 2 3 3 2 3 2 1 2
 2 1 3 2 2 1 1 2 1 1 2 2 2 2 1 1 2 1 1 2 2 1 3 2 2 1 3 2 2 1 3 1 2 3 2 3 2
 3 1 3 3 2 2 1 3 3 2 1 2 1 2 2 1 3 3 2 1 2 3 2 1 2 2 1 2 1 3 3 2 2 1 2 2 2
 3 2 1 1 1 1 2 3 1 1 2 1 2 1 1 2 1 3 1 3 1 3 2 2 3 2 1 2 3 3 2 1 3 1 2 3 3
 2 2 2 3 3 2 3 3 1 3 3 3 3 3 2 1 1 3 1 2 3 3 1 2 2 2 2 3 1 3 3 2 3 3 2 2 3
 3 3 1 2 1 2 1 2 3 2 3 1 1 2 1 3 1 2 1 2 3 2 3 1 2 1 3 2 2 2]
Real label:
 [1 1 3 3 1 3 1 1 1 1 1 3 3 3 3 2 1 3 1 3 1 2 1 2 2 2 3 1 1 1 3 1 1 2 2 3 3
 2 3 2 1 1 3 2 1 3 3 3 2 2 2 1 3 2 1 2 2 2 3 2 2 3 3 3 2 2 1 3 1 3 2 1 3 1
 3 1 3 1 3 3 2 2 1 1 3 3 3 3 2 2 2 3 3 2 3 2 3 1 1 2 3 3 3 2 3 2 3 3 1 3 1
 2 2 1 3 3 2 2 1 2 2 1 1 1 3 1 3 1 2 3 3 1 2 1 2 3 1 2 1 2 3 3 3 3 2 1 2 3
 2 3 1 2 2 2 3 1 1 1 3 3 2 3 2 3 2 3 1 2 1 3 2 3 1 3 1 1 1 1 3 3 3 1 1 1 2
 2 3 1 2 3 1 3 3 3 1 3 1 3 3 3 1 1 3 3 2 1 2 1 1 3 1 3 2 2 2 3 3 2 3 2 1 2
 2 1 3 2 2 1 1 2 1 1 2 2 2 2 1 1 2 1 1 2 2 1 3 2 2 1 3 2 2 1 3 1 2 3 2 3 2
 3 1 3 3 2 2 1 3 3 2 1 2 1 2 2 1 3 3 2 1 2 3 2 1 2 2 1 2 1 3 3 2 2 1 2 2 2
 3 2 1 1 1 1 2 3 1 1 2 1 2 1 1 2 1 3 1 3 1 3 2 2 3 2 1 2 1 3 2 1 3 1 2 3 3
 2 2 2 3 3 2 3 3 1 3 3 3 3 3 2 1 1 3 1 2 3 3 1 2 2 2 2 3 1 3 3 2 3 3 2 2 3
 3 3 1 2 1 2 1 2 3 2 3 1 1 2 1 3 1 2 1 2 3 2 3 1 2 1 3 2 2 2]
all equal?  False
Fraction equal:  0.99
```

Figure 2.21: SGD classifier results from assignment2.ipynb

The evaluation contains a built-in classification report from Sci-kit Learn and a manual test of generating several hundred random indexes from test set and applying the trained model to it. To further rule out some issues a second training and testing was done, the rest for the original training set as mentioned in 2.1. This test set is larger than the original test set used in this section.

2.3.4 SGD Classifier Mel Spectrogram with rest of test data

This step requires preparation of the rest batch of test data, which include calculation of Mel Spectrogram and normalization. More details can be seen in Appendix 1.

```
# EXSTRA TEST WITH UNSEEN TEST DATA

# The test dataset is big only 1/3 was used for original test.
# Now the remaining will be tested on the SGD classifier again
```

```
# Define the datasets for testing with links to files, folders and labels
# Labels are defined as numbers 1 = good, 2 = broken and 3 = heavy load
good_test_rest = Dataset(good_files_test_rest, good_folder_test, fs, np.ones(len(good_files_test_rest)).astype(int))
broken_test_rest = Dataset(broken_files_test_rest, broken_folder_test, fs, 2*np.ones(len(broken_files_test_rest)).astype(int))
heavyload_test_rest = Dataset(heavyload_files_test_rest, heavyload_folder_test, fs, 3*np.ones(len(heavyload_files_test_rest)).astype(int))
```

```
# Combine the testing data sets and calculate sound arrays
data_test_rest = Calculate([good_test_rest, broken_test_rest, heavyload_test_rest])
```

```
# Calculate the mel spectrogram for test data
test_spec_rest, test_label_rest, test_files_rest = data_test_rest.calcspec()
print(test_spec_rest.shape)
print(test_spec_rest[sl2])
print(test_label_rest[sl2])
print(test_files_rest[sl2])
```

```
(1348, 128, 130)
[[[-20.530205  -16.077065  -17.92067   ... -16.985428  -14.223267
   -16.517006 ]
  [-20.75354   -18.00978   -21.617218  ... -19.388115  -12.928658
   -16.432259 ]
  [-22.830948  -14.9807205 -14.277832  ... -12.333511   -7.073616
   -10.883652 ]
```

Figure 2.22: Preparation test data for rest from assignment2.ipynb

The results are again very good at 98%. The test dataset here is much larger (2/3 of samples) than the original test data set (1/3 of samples). However, there is still a risk for an issue somewhere, and the whole process should be validated.

```
# Check results again using same trained SGD classifier as for the first part of test set
# It is now fed with the remaining data from test set, that was splitted in the beginning.
y_pred_sgd_mel_rest = sgd_model_mel.predict(test_spec1dnorm_rest)
print("Accuracy of model = %2f%%" % (accuracy_score(test_label_rest, y_pred_sgd_mel_rest)*100))
```

```
Accuracy of model = 98.516320%
```

```
print(classification_report(test_label_rest, y_pred_sgd_mel_rest, target_names=['good', 'broken', 'heavyload']))
```

```
              precision    recall  f1-score   support

        good       1.00      0.96      0.98       446
      broken       1.00      1.00      1.00       444
   heavyload       0.96      1.00      0.98       458

    accuracy                           0.99      1348
   macro avg       0.99      0.99      0.99      1348
weighted avg       0.99      0.99      0.99      1348
```

```
# Test with the random samples
n_test_rest=500
idxs_test_rest = np.random.choice(len(test_spec1dnorm_rest), n_test_rest, replace=False)
idxs_test_rest
y_samples_sgd_mel_rest = sgd_model_mel.predict(test_spec1dnorm_rest[idxs_test_rest])
print('Predicted with SDG: \n', y_samples_sgd_mel_rest, '\nReal label: \n', test_label_rest[idxs_test_rest])
print('all equal? ' , np.array_equal(y_samples_sgd_mel_rest, test_label_rest[idxs_test_rest]))
print('Fraction equal: ', np.sum(y_samples_sgd_mel_rest==test_label_rest[idxs_test_rest])/len(test_label_rest[idxs_test_rest]))
```

```
Predicted with SDG:
 [3 1 2 1 1 3 3 2 2 1 2 3 2 2 3 1 2 3 2 2 2 1 3 3 2 2 2 3 3 3 1 1 3 3 3 2 3
 1 3 1 1 1 2 1 3 3 3 1 3 1 1 3 3 3 2 1 2 2 3 3 3 3 3 1 2 2 2 1 2 3 3 2 2 3
 2 3 2 3 3 1 1 2 1 1 3 2 1 2 1 1 1 3 1 3 3 1 2 3 1 1 3 2 2 3 1 3 1 3 1 1 3
 2 1 1 2 2 2 2 1 2 3 1 1 3 2 2 1 3 1 3 1 2 3 2 3 3 2 2 2 3 3 1 3 1 2 1 2 3
 1 2 3 2 1 2 3 3 1 1 1 2 3 3 1 2 2 3 3 1 1 2 1 1 3 2 1 1 3 1 1 1 3 1 3 2 1
 2 3 2 2 2 3 1 3 2 2 3 2 3 2 2 3 2 3 2 2 1 3 2 2 2 1 1 1 1 3 1 1 3 1 2 2 2
 1 2 2 2 2 1 1 1 1 3 2 2 2 2 1 2 2 3 1 3 3 3 3 3 2 3 3 1 2 2 2 1 1 1 2 1 1
 2 1 1 3 3 1 3 1 3 3 3 1 3 2 2 3 2 3 1 2 1 3 1 3 2 3 2 2 1 2 3 1 2 1 1 3
 1 2 2 1 2 2 3 1 1 2 2 1 2 2 2 3 3 1 1 3 2 3 3 1 3 3 2 3 2 1 2 1 1 2 2 3 3
 1 1 2 2 3 1 1 2 3 2 2 1 1 1 2 1 3 2 1 2 3 1 1 1 3 2 2 2 3 1 3 3 2 1 3 3 3
 1 2 2 3 1 3 3 1 2 2 2 3 2 3 1 2 1 2 2 2 1 2 1 3 3 1 1 3 2 2 1 1 2 2 2 1 2
 3 2 3 3 2 3 2 3 2 2 3 1 3 3 1 3 2 2 3 1 1 1 2 2 2 1 3 2 2 2 3 1 1 3 1 3 2
 3 3 3 2 3 3 1 3 3 3 3 1 3 3 2 1 2 1 3]
Real label:
 [3 1 2 1 1 3 3 2 2 1 2 3 2 2 3 1 2 3 2 2 2 1 3 3 2 2 2 3 3 3 1 1 3 3 3 2 3
 1 3 1 1 1 2 1 3 3 3 3 1 3 1 1 3 3 3 2 1 2 2 3 3 3 3 3 1 2 2 2 1 2 3 3 2 2 3
 2 3 2 3 3 1 1 2 1 1 1 2 1 2 1 1 1 3 1 3 3 1 2 3 1 1 3 2 2 3 1 3 1 3 1 1 3
 2 1 1 2 2 2 2 1 2 3 1 1 3 2 2 1 3 1 3 1 2 3 2 3 3 2 2 2 3 3 1 3 1 2 1 2 3
 1 2 3 2 1 2 3 3 1 1 1 2 3 3 1 2 2 3 3 1 1 2 1 1 3 2 1 1 3 1 1 1 3 1 3 2 1
 2 3 2 2 2 3 1 3 2 2 3 2 3 2 2 3 2 3 2 2 1 3 2 2 2 1 1 1 1 3 1 1 3 1 2 2 2
 1 2 2 2 2 1 1 1 1 3 2 2 2 2 1 2 2 3 1 3 3 3 3 3 2 3 3 1 2 2 2 1 1 1 2 1 1
 2 1 1 3 3 1 3 1 3 3 3 1 3 2 2 3 2 3 1 2 1 3 1 3 2 3 2 2 1 2 3 1 2 1 1 3
 1 1 1 2 1 1 2 3 1 1 3 3 1 3 2 2 3 2 3 1 2 1 3 1 3 2 3 2 2 1 2 3 1 2 1 1 3
 3 2 2 1 2 2 3 1 1 2 2 1 2 2 2 3 3 1 1 3 2 3 3 1 3 3 2 3 2 1 2 1 1 2 2 3 3
 1 1 2 2 3 1 1 2 3 2 2 1 1 1 2 1 3 2 1 2 3 1 1 1 3 2 2 2 3 1 3 3 2 1 3 3 3
 1 2 2 3 1 3 3 1 2 2 2 3 2 3 1 2 1 2 2 2 1 2 1 3 3 1 1 3 2 2 1 1 2 2 2 1 2
 3 2 3 3 2 3 2 3 2 2 3 1 3 3 1 3 2 2 3 1 1 1 2 2 2 1 3 2 2 2 3 1 1 3 1 3 2
 3 3 1 2 3 3 1 3 3 3 3 1 3 3 2 1 2 1 3]
all equal?  False
Fraction equal:  0.986
```

Figure 2.23: SGD classifier test of rest from assignment2.ipynb

3 Discussion

This task was very interesting as it covered a broad range of fields, and it contained real data from real machines. First the large amount of data was a bit overwhelming, so it was decided to spend a lot of time developing a data model in Python, using an object-oriented approach. Quite happy with the result and it will be useful in future projects.

Downside was that there was not enough time left for the data processing. Instead, new features were calculated, amongst where the Mel Spectrogram was the one that caught most attention. Certainly, the link between machine vision and sound analysis is much closer than expected. As shown in [3] this approach is suited for deep learning models too.

The failure of classification based on RMS is also important, as it shows the preprocessing is important, and also that some features appear good in theory (no noise) but get really bad in a practical environment (noise added). Would however been less intensive to calculate with RMS, as it is a simple feature to use.

Finally, the results for Mel Spectrograms are still a bit suspicious due to the high accuracy that was gotten with so little tuning. The spectrogram contains a lot of information, from both time and frequency domain, so it is a feature that “has it all”. Would still like to investigate any issues in the data processing, that could cause leak between test and train data. This has not been done but could be a great continuation to this task.

References

[1] Introduction to engineering experimentation, Wheeler, Anthony J.; Ganji, Ahmad R. 3rd ed. BostonPearson c2010

[2] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd Edition, Aurélien Géron3O'Reilly Media, Inc 2022

[3] Audio Deep Learning Made Simple: Sound Classification, Step-by-Step | by Ketan Doshi | Towards Data Science

[4] How to Create & Understand Mel-Spectrograms | by Christopher Lewis | Medium

[5] https://www.idmt.fraunhofer.de/en/publications/datasets/isa-electric-engine.html

[6] https://dynamox.net/en/blog/the-peak-peak-to-peak-and-rms-values-in-vibration-analysis

Appendices

Appendix 1 – assignment2.ipynb

Appendix 2 – sound.py

Appendix 3 – dataset.py

Appendix 4 – calculation.py