Experiment A

Per Signal Anomaly Detection

- Per Signal Anomaly Detection with Twitter

```
#########################################################################
# Per Signal Anomaly Detection with Twitter Anomaly Detection Algorithm #
#########################################################################

# Install twitter anomaly detection package
## install.packages("devtools")
## devtools::install_github("twitter/AnomalyDetection")


# Load twitter anomaly detection package
library(AnomalyDetection)

# Get information about functions
## help(AnomalyDetectionTs)
## help(AnomalyDetectionVec)

# Load sample data
data(raw_data)

# Detect anomalies in sample data with the TimeSeries Anomaly DetectionFunction
res = AnomalyDetectionTs(raw_data, max_anoms=0.02, direction='both', plot=TRUE)
res$plot

res2 = AnomalyDetectionVec(raw_data[,2], max_anoms=0.02, period=1440, direction='both', only_last=FALSE, plot=TRUE)
res2$plot

####### Generate signal for anomaly detection ##############

period=200
reps = 6


######################### Signal 1 ##########################

# Generate signal 1
t <- seq(from = pi, to = -pi, length.out = 200)
t <- rep(t,6)
tmp = 2 * sin(t)
tmp = tmp + runif(200*6 ,min = -0.2, max = 0.2)
tmp[590:600] = 0.8 + runif(11, -0.1, 0.1)
tmp[720] = tmp[720] + 1
tmp[790] = tmp[790] + 1
tmp[900] = tmp[900] - 1.5
tmp[480:500] = -0.8 + runif(21, -0.05, 0.05)
tmp[1030:1080] = tmp[1030:1080] - 1.2 + runif(51,-0.1,0.1)
plot(tmp)

signal1 = tmp

# Detect anomalies in Signal 1
res_signal1 = AnomalyDetectionVec(signal1, max_anoms=0.1, period=200, direction='both', only_last=FALSE, plot=TRUE)
res_signal1$plot


######################### Signal 2 ##########################

# Generate signal 2
t <- seq(from = pi, to = -pi, length.out = 200)
t <- rep(t,6)
tmp = 2 * sin(t)
tmp = tmp + runif(200*6 ,min = -0.2, max = 0.2)
tmp[420:480] = tmp[420:480] + runif(61, -0.8, 0.8)
tmp[1000:1020] = tmp[1000:1020] + 0.5
tmp[1020:1080] = tmp[1020:1080] + 1.1
tmp[1080:1100] = tmp[1080:1100] + 0.5
plot(tmp)

signal2 = tmp

# Detect anomalies in Signal 2
res_signal2 = AnomalyDetectionVec(signal2, max_anoms=0.1, period=200, direction='both', only_last=FALSE, plot=TRUE)
res_signal2$plot


######################### Signal 3 ##########################

# Generate signal 3
t <- seq(from = pi, to = -pi+(pi/200), length.out = 200)
t <- rep(t,6)
tmp = 2 * sin(t)
tmp[510:520] = tmp[510:520]  + runif(11, -0.7, 0.7)
tmp[800:810] = tmp[800:810] + runif(11, -0.4, 0.4)
tmp[900:910] = tmp[900:910] + runif(11, -0.7, 0.7)
tmp[1050:1200] = 2
plot(tmp)

signal3 = tmp

# Detect anomalies in Signal 3
res_signal3 = AnomalyDetectionVec(signal3, max_anoms=0.15, period=200, direction='both', only_last=FALSE, plot=TRUE)
res_signal3$plot


######################### Signal 4 ##########################

# Generate signal 4
t <- seq(from = 0, to = 2, length.out = 1200)
tmp = t + runif(1200, -0.05, 0.05)
tmp[510:520] = tmp[510:520] + 0.2
tmp[800:810] = tmp[800:810] + 0.2
tmp[1100:1200] = seq(from=tmp[1099], to=2.2, length.out = 101) + runif(101,-0.05, 0.05)
plot(tmp)

signal4 = tmp

# Detect anomalies in Signal 4
```

```
res_signal4 = AnomalyDetectionVec(signal4, max_anoms=0.10, period=400, direction='both', only_last=FALSE, plot=TRUE)
res_signal4$plot


######################## Detrend Signal 4 ##########################
# Load library for detrending
library(pracma)

# Detrend signal 4
signal4_detrended = detrend(signal4, 'linear')
plot(signal4_detrended)


# Detect anomalies in Detrended Signal 4
res_signal4_detrend = AnomalyDetectionVec(signal4_detrended[1:1200], max_anoms=0.04, period=400, direction='both', only_last=FALSE, plot=TRUE)
res_signal4_detrend$plot
```

Experiment B

Anomaly Detection

- Ideal data OCSVM
- Simulator OCSVM Grid-search
- SECOM pre-processing and OCSVM

One-class SVM fake data

May 30, 2017

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager
from sklearn import svm
%matplotlib notebook
import time


n_training = 200
n_test = 40
n_outliers = 20


# Generate coordinate matrix from -10 to 10. Resolution 500
x_vector, y_vector = np.meshgrid(np.linspace(0, 20, 1000), np.linspace(0, 20, 1000))

# Generate train data
X1 = 0.8 * np.random.randn(round(n_training/2), 2)
X2 = 0.8 * np.random.randn(round(n_training/2), 2)
X_train = np.r_[X1 + 16, X2 + 6] # Add train data to existing axis

# Generate inliers
X1 = 0.8 * np.random.randn(round(n_test/2), 2)
X2 = 0.8 * np.random.randn(round(n_test/2), 2)
X_test = np.r_[X1 + 16, X2 + 6] # Add inliers to existing axis

# Generate outliers
```

```
np.random.seed(7)
X_outliers = np.random.uniform(low=0, high=20, size=(n_outliers, 2))

# print(X_test,'\n')
# print(X_outliers,'\n')
# print(np.concatenate((X_test,X_outliers),0),'\n')

# USE THESE!!!!
# X_test = np.concatenate((X_test,X_outliers),0)
# X_test = np.random.shuffle(X_test)

# fit the model
nu = .1 #0.1 before
gamma = 0.3 #0.1 before
# clf = svm.OneClassSVM(nu=nu, kernel="rbf", gamma=gamma)
clf = svm.OneClassSVM(nu = nu, gamma = gamma, kernel = 'rbf')
clf.fit(X_train)
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
y_pred_outliers = clf.predict(X_outliers)


n_error_train = y_pred_train[y_pred_train == -1].size
n_error_test = y_pred_test[y_pred_test == -1].size
n_error_outliers = y_pred_outliers[y_pred_outliers == 1].size

# plot the line, the points, and the nearest vectors to the plane
Z = clf.decision_function(np.c_[x_vector.ravel(), y_vector.ravel()])
Z = Z.reshape(x_vector.shape)
print(Z.min())

plt.figure(1, figsize=(9,7))
# plt.title("Anomaly detection with One-class SVM, RBF, nu=%s, gamma=%s"%(nu, gamma))
plt.title("Anomaly detection with One-class SVM")
plt.contourf(x_vector, y_vector, Z, levels=np.linspace(Z.min(), 0, 7), cmap=plt.cm.PuBu)
a = plt.contour(x_vector, y_vector, Z, levels=[0], linewidths=2, colors='darkred')
plt.contourf(x_vector, y_vector, Z, levels=[0, Z.max()], colors='palevioletred')
```

```
        s = 40
        b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white', s=s)
        b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='blueviolet', s=s)
        c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='gold', s=s)
        plt.axis('tight')
        plt.xlim((0, 20))
        plt.ylim((0, 20))
        plt.legend([a.collections[0], b1, b2, c],
                   ["Decision line", "Training points",
                    "Normal values", "Failures"],
                   loc="upper left",
                   prop=matplotlib.font_manager.FontProperties(size=11))
        plt.xlabel('Temperature [Celsius]')
        plt.ylabel('Pressure [mbar]')
        # plt.xlabel(
        #     "training data classified as outliers: %d/%s - inliers classified as outliers: %d/%s - "
        #     "outliers classified as inliers: %d/%s"
        #     % (n_error_train,n_training, n_error_test,n_test, n_error_outliers,n_outliers))
        plt.show()
```

```
-3.0693398254
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

In [ ]:

Simulator OCSVM Grid-search

May 30, 2017

In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import OneClassSVM
from sklearn import preprocessing
from sklearn import metrics
from sklearn.decomposition import PCA
import numpy as np
from numpy import set_printoptions
import time
import matplotlib
import datetime
import seaborn as sns


def plot_timeseries(data,columns=[0],t1=None,t2=None, locator=None, format=None,
                    title=' ',xlabel=' ',ylabel=' ',legends=None, grid=True):
    dataset=get_dataset(data,t1,t2)
    timespan = dataset.index.max()-dataset.index.min()

    if locator == None or format == None:
        locator,format = set_auto_ticks(timespan)

    fig, ax = plt.subplots(1)
    ax.plot(dataset.index, dataset[columns])

    # Add grid
```

```
    ax.grid(grid)

    ## Set ticks format and frequency:
    ax.xaxis.set_major_locator(locator)
    ax.xaxis.set_major_formatter(format)
    fig.autofmt_xdate()

    ## Set title, xlabel, ylabel, legend
    fig.suptitle(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    if legends is not None:
        ax.legend(labels=legends, loc='best')

    return fig

def fill_between_time(fig,t1,t2,color='navajowhite',alpha=0.7):
    ax = fig.gca()

    date1 = datetime.datetime.strptime(t1, "%Y-%m-%d %H:%M:%S")
    date2 = datetime.datetime.strptime(t2, "%Y-%m-%d %H:%M:%S")

    ax.axvspan(date1, date2, color=color, alpha=alpha)
    fig.add_axes(ax)
    return fig

def add_pointer(fig,dataset,pointer_time,pointer_title,color='black',point_len=500):
    ax = fig.gca()

    pointer_list = generate_pointer_list(pointer_time,pointer_title)

    point_len = dataset[[0]].max()/4
    for date, label in pointer_list:
        ax.annotate(label, xy=(date, dataset.asof(date) + 20),
                    xytext=(date, dataset.asof(date) + point_len),
                    arrowprops=dict(facecolor=color),
                    horizontalalignment='left', verticalalignment='top')
```

```
    fig.add_axes(ax)
    return fig

# Sets correct timespan for the plot depending on inputted timestamps.
def get_dataset(dataset1,t1,t2):
    if t1 != None and t2 == None:
        dataset = dataset1.loc[t1::]
    elif t1 != None and t2 != None:
        dataset = dataset1.loc[t1:t2]
    elif t1 == None and t2 != None:
        dataset = dataset1.loc[:t2]
    else:
        dataset = dataset1

    return dataset



# Sets automatic x-axis datetime ticks and format according to plot timespan.
def set_auto_ticks(timespan):
    if timespan < datetime.timedelta(days=4) and timespan > datetime.timedelta(days=2):
        locator = matplotlib.dates.HourLocator(byhour=range(0, 24, 3))
        format = matplotlib.dates.DateFormatter('%d.%m %H:%M')  # '%a %d\n%b %Y'
        # format = matplotlib.dates.DateFormatter('%d. %b %H:%M') #'%a %d\n%b %Y'
    elif timespan < datetime.timedelta(days=2) and timespan > datetime.timedelta(hours=12):
        locator = matplotlib.dates.HourLocator(byhour=range(0, 24, 3))
        format = matplotlib.dates.DateFormatter('%d.%m %H:%M')
    elif timespan < datetime.timedelta(hours=12) and timespan > datetime.timedelta(hours=4):
        locator = matplotlib.dates.HourLocator(byhour=range(0, 24, 1))
        format = matplotlib.dates.DateFormatter('%H:%M')
    elif timespan < datetime.timedelta(hours=4) and timespan > datetime.timedelta(hours=2):
        locator = matplotlib.dates.MinuteLocator(byminute=range(0, 65, 5))
        format = matplotlib.dates.DateFormatter('%H:%M')
    elif timespan < datetime.timedelta(hours=2) and timespan > datetime.timedelta(hours=1):
        locator = matplotlib.dates.MinuteLocator(byminute=range(0, 65, 15))
        format = matplotlib.dates.DateFormatter('%H:%M:%S')
    elif timespan < datetime.timedelta(hours=1) and timespan > datetime.timedelta(minutes=15):
```

```
        locator = matplotlib.dates.MinuteLocator(byminute=range(0, 65, 2))
        format = matplotlib.dates.DateFormatter('%H:%M:%S')
    elif timespan < datetime.timedelta(minutes=15) and timespan > datetime.timedelta(minutes=5):
        locator = matplotlib.dates.MinuteLocator(byminute=range(0, 65, 2))
        format = matplotlib.dates.DateFormatter('%H:%M:%S')
    elif timespan < datetime.timedelta(minutes=5) and timespan > datetime.timedelta(minutes=2):
        locator = matplotlib.dates.SecondLocator(bysecond=range(0, 65, 30))
        format = matplotlib.dates.DateFormatter('%H:%M:%S')
    elif timespan < datetime.timedelta(minutes=2):
        locator = matplotlib.dates.SecondLocator(bysecond=range(0, 65, 10))
        format = matplotlib.dates.DateFormatter('%H:%M:%S')
    else:
        locator = matplotlib.dates.HourLocator(byhour=range(0, 24, 3))
        # locator = matplotlib.dates.DayLocator(interval=2)
        # format = matplotlib.dates.DateFormatter('%a %d\n%b %Y')
        format = matplotlib.dates.DateFormatter('%d.%m %H:%M')

    return locator, format

def generate_pointer_list(time_string,title_string):
    pointer_data=[]
    for i in range(len(time_string)):
        pointer_data.append((datetime.datetime.strptime(time_string[i], "%Y-%m-%d %H:%M:%S"), title_string[i]))
    return pointer_data

# Return sample label edges. When a alarm is detected on the raise and on the fall.
def get_label_edges(data):
    df = data.loc[data['Scored Labels']==1]
    timedelta = datetime.timedelta(minutes=1)
    prev_date=df.index[0]
    return_array=[prev_date]
    for date in df.index:
        if (date - prev_date) >  timedelta:
            return_array.append(prev_date)
            return_array.append(date)
        prev_date=date
    return return_array
```

```
def normalize_binary(binary_df, min = -1, max = 1):
    # Normalize binary (0,1) -> (min,max) if unspecified -> (-1,1)
    binary_df[binary_df == 0] = min
    binary_df[binary_df == 1] = max
    # print(df_binary.head())

    return binary_df


def normalize_numeric(numeric_df, min = -1, max = 1):
    # Normalize numeric
    min_max_scaler = preprocessing.MinMaxScaler(feature_range=(-1, 1))
    np_scaled = min_max_scaler.fit_transform(numeric_df)
    numeric_df = pd.DataFrame(np_scaled, columns=numeric_df.columns, index=numeric_df.index)
    # print(numeric_df.head())

    return  numeric_df


def score_one_class(y_pred, y_true = None):
    if y_true == None:
        y_true = [1] * len(y_pred)
    print('One class model scoring')
    print('Accuracy: ' + str(metrics.accuracy_score(y_true, y_pred)))
    print('Precision score: : ' + str(metrics.precision_score(y_true, y_pred)))

def save_fig_pred_and_true(y_pred, param_nu, param_gamma, param_kernel, train_time, precision, show_fig=False):
    fig = plot_timeseries(y_pred,
                                  title='nu = ' + str(param_nu) + ', gamma = ' + str(param_gamma) + ', kernel = ' + str(
                                      param_kernel) + '\n ' 'Train time: ' + str(
                                      train_time) + 's, precision: ' + str(
                                      precision) + '\nTrained with three days of normal simulator data, tested with '
                                                   'one, true errors marked red')
    # Manually added errors
    fill_between_time(fig, t1='2017-03-24 10:51:17', t2='2017-03-24 10:51:25', color='red', alpha=1)
```

```
            fill_between_time(fig, t1='2017-03-24 10:55:45', t2='2017-03-24 10:56:12', color='red', alpha=1)
            fill_between_time(fig, t1='2017-03-24 10:57:24', t2='2017-03-24 10:57:33', color='red', alpha=1)
            fill_between_time(fig, t1='2017-03-24 10:57:40', t2='2017-03-24 10:57:43', color='red', alpha=1)
            fill_between_time(fig, t1='2017-03-24 06:40:22', t2='2017-03-24 06:40:32', color='red', alpha=1)
            fill_between_time(fig, t1='2017-03-24 06:13:38', t2='2017-03-24 06:13:47', color='red', alpha=1)
            fig.set_size_inches(26, 10)
            fig.savefig('C:/ putt path her, diff , extra errors ' + 'nu' + str(param_nu) + ' ' + str(
                param_kernel) + ' gamma' + str(param_gamma) + '.png', format='png')  # , dpi=800)
            if show_fig:
                plt.show()
            else:
                plt.close(fig)


In [ ]: if __name__ == '__main__':
            set_printoptions(linewidth=200)
            pd.set_option('display.width', 300)
            # Fill NA, set datetime as index

            df = pd.read_csv(
                'C:/putt path her/Reactor001Values20.03-24.03Header.csv')
            print('************** read datafile from Google Drive *************')

            df.index = pd.to_datetime(df['DateTime'])
            print('****************** index set to datetime *******************')

            df = df.drop(
                ['DateTime', 'Reactor_001.Auto', 'StorageTank_001.MaxLevel', 'StorageTank_001.MinLevel', 'StorageTank_001.Auto',
                 'Reactor_001.SteamValve_InMaintenance'],1)
            #, 'Reactor_001.ReactLevel', 'Reactor_001.ReactTemp', 'StorageTank_001.ProdLevel'], 1)
            print(df.columns)
            print('**************** dropped irrelevant columns ****************')

            df = df.fillna(method='ffill')
            print('****************** NaNs are forward filled *****************')

            train_df = df.loc['2017-03-20 00:00:00':'2017-03-23 23:59:59']
            test_df = df.loc['2017-03-24 00:00:00':'2017-03-24 23:59:00']
```

```
    print('*************** cut into train and test set ***************')

    if np.all(test_df.columns == train_df.columns):
        print('*************** train and test columns equal ***************')

    # Lable real errors
    test_labels = pd.DataFrame(np.ones(shape=(len(test_df.index), 1)), columns=['labels'], index=test_df.index)
    test_labels.loc['2017-03-24 10:51:17':'2017-03-24 10:51:25', ['labels']] = -1   # First error
    test_labels.loc['2017-03-24 10:54:05':'2017-03-24 10:54:21', ['labels']] = -1   # Second error
    test_labels.loc['2017-03-24 10:55:45':'2017-03-24 10:56:12', ['labels']] = -1  # Third error
    test_labels.loc['2017-03-24 10:57:24':'2017-03-24 10:57:33', ['labels']] = -1  # Fourth error
    test_labels.loc['2017-03-24 10:57:40':'2017-03-24 10:57:43', ['labels']] = -1  # Fourth error

    # Generate 10 numeric errors and lable them
    test_df.loc['2017-03-24 06:40:22':'2017-03-24 06:40:32', ['Reactor_001.ReactTemp']] = 20   # test errors
    test_labels.loc['2017-03-24 06:40:22':'2017-03-24 06:40:32', ['labels']] = -1   # test error labels

    window = 2
    # Generate 10 binary errors and lable them
    test_df.loc['2017-03-24 06:13:38':'2017-03-24 06:13:47', ['Reactor_001.TransferPump']] = 0   # test errors
    test_labels.loc['2017-03-24 06:13:38':'2017-03-24 06:13:47', ['labels']] = -1   # test error labels
    y_true = test_labels['labels']
    y_true.drop(y_true.head(window).index, inplace=True)
    print('*************** generated labels for test set ***************')

    # Generate features
    train_df = train_df.assign(tempDiff = train_df['Reactor_001.ReactTemp'].diff(periods = window))
    test_df = test_df.assign(tempDiff = test_df['Reactor_001.ReactTemp'].diff(periods = window))
    train_df = train_df.assign(levelDiff = train_df['Reactor_001.ReactLevel'].diff(periods = window))
    test_df = test_df.assign(levelDiff = test_df['Reactor_001.ReactLevel'].diff(periods = window))
    train_df = train_df.assign(prodLevelDiff = train_df['StorageTank_001.ProdLevel'].diff(periods = window))
    test_df = test_df.assign(prodLevelDiff = test_df['StorageTank_001.ProdLevel'].diff(periods = window))
    train_df.drop(train_df.head(window).index, inplace=True)
    test_df.drop(test_df.head(window).index, inplace=True)

    # Drop sensor features
#     train_df = train_df.drop(['Reactor_001.ReactTemp', 'Reactor_001.ReactLevel', 'StorageTank_001.ProdLevel'], 1)
```

```
#     test_df = test_df.drop(['Reactor_001.ReactTemp', 'Reactor_001.ReactLevel', 'StorageTank_001.ProdLevel'], 1)

    '''pca = PCA(n_components=7)
    np_train_df = pca.fit_transform(train_df)
    train_df = pd.DataFrame(np_train_df, index=train_df.index)
    np_test_df = pca.transform(test_df)
    test_df = pd.DataFrame(np_test_df, index = test_df.index)
    print('************ PCA completed for train and test set **********')'''

    min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))

    np_scaled_train = min_max_scaler.fit_transform(train_df)
    train_df_norm = pd.DataFrame(np_scaled_train, columns=train_df.columns, index=train_df.index)

    np_scaled_test = min_max_scaler.transform(test_df)
    test_df_norm = pd.DataFrame(np_scaled_test, columns=test_df.columns, index=test_df.index)
    print('************* train and test set are normalized ************')


    # param_nu = 0.0001
    # param_kernel = 'rbf' # ['linear', 'poly', 'rbf', 'sigmoid']
    # param_gamma = 0.05
    kernels = np.array(['rbf', 'sigmoid'])
    nus = np.array([0.01, 0.001, 0.0001, 0.00001, 0.000001, 0.0000001, 0.000000000001])
    # 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001, 0.0000001
    nus = nus[::-1] # flip nus array to get an efficient start
    gammas = np.logspace(-8, 4, 21)
    start_for = time.clock()
    for param_kernel in kernels:
        for param_nu in nus:
            for param_gamma in gammas:
                start = time.clock()
                clf = OneClassSVM(nu=param_nu, kernel=str(param_kernel), gamma=param_gamma)
                clf.fit(train_df_norm)
                y_pred_test = clf.predict(test_df_norm)
                # print(y_pred_test)
                mlp_time = format(time.clock() - start, '.2f')
```

```
                    y_pred = pd.DataFrame(y_pred_test, columns=['predicted label'], index=test_df.index)

                    precision = metrics.precision_score(y_true, y_pred, pos_label=-1, average='binary')
                    print(metrics.confusion_matrix(y_true,y_pred))
                    print('accuracy: ', metrics.accuracy_score(y_true, y_pred))
                    print('recall: ', metrics.recall_score(y_true, y_pred, average='binary'))
                    print('nu = ' + str(param_nu) + ', gamma = ' + str(param_gamma) + ', kernel = ' + str(
                        param_kernel) + '\n ' 'Train time: ' + str(
                        mlp_time) + 's, precision = ' + str(
                        precision) + '\n')

                    if precision > (0.5): # 0.2 *
                        save_fig_pred_and_true(y_pred, param_nu, param_gamma,
                                               param_kernel, mlp_time, precision, show_fig=False)

        tot_time = format(time.clock() - start_for, '.2f')
```

In [ ]:

SECOM pre-processing and OCSVM

May 30, 2017

0.1 SECOM (Semiconductor manufacturing) dataset.

0.1.1 Semi-supervised learning - One-Class SVM

About the dataset Key facts: Data Structure: The data consists of 2 files the dataset file SECOM consisting of 1567 examples each with 591 features a 1567 x 591 matrix and a labels file containing the classifications and date time stamp for each example.

As with any real life data situations this data contains null values varying in intensity depending on the individuals features. This needs to be taken into consideration when investigating the data either through pre-processing or within the technique applied.

The data is represented in a raw text file each line representing an individual example and the features seperated by spaces. The null values are represented by the 'NaN' value as per Python.

Dr.-Ing. Thorsten Wuest's comment about the dataset It has to be noted that the SECOM data set represents a very challenging data set. It was published as part of the "causality challenge" which suggests that classification will not be an easy task. The baseline results of classification performance published in accordance with the SECOM data set by McCann et al. (2010) show the difficulty of achieving good classification results with this data set.

This test This dataset will first be splitted into two parts - one for training and one for testing. Since one of Intelecy's cases is considering semi-supervised anomaly detection from manufacturing data, the training set will be cleaned for fault values, and further. This is done to assure that we only train with normal state data. 1. Indexing the dataset with the timestamp for convenience. 2. Splitting into two parts, train and test.

Further, we do pre-processing based on only the knowledge we gain from the train set. 3. Remove vectors missing more than n% (first 6%) of its values. 4. Remove features with little or no variance (no variance on first test) 5. Scaling (normalization)

As the dataset is reduced both in vectors and dimensions, we know how to pre-process the test set. 6. Pre-process the test set. 7. Split the pre-processed test set into an optimization set and a validation set. This way we can iterate trough predictions and optimize the parameters based on scored predictions on the optimization set.

Now we are ready for some ML. 8. Run initial test with default One-Class SVM values and validate it with validation set. This will hopefully indicate feasibilty. 9. Loop through parameters and optimize with optimization set. Print settings and results for precision > 0.4. 10. Test best solutions on validation set!

One of the problems that we meet using the One-Class SVM is that the test set must be further splitted into an optimization set and a validation set. This is because the test set is the only labelled data we have, and to optimize, we need a score from predictions on labelled data. ###### "Scoring on the test set will increase the chance of remembering instead of learning!"

In [439]:
```
# from sklearn.model_selection import train_test_split
from sklearn.svm import OneClassSVM
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import numpy as np
import time
from sklearn import metrics
# from IPython.core.interactiveshell import InteractiveShell
# InteractiveShell.ast_node_interactivity = "all"
```

In [440]:
```
# generate spark DataFrame from csv-file in blob.
csvFile = spark.read.csv('wasb:///SECOM.csv', header=True, inferSchema=True)
df = csvFile.toPandas()
print('Initial shape of dataset: ', df.shape)
df = df.dropna(0, subset=['Yield_Pass_Fail'])
print('After removing rows with missing label - these give nothing: ', df.shape)
```

```
Initial shape of dataset:  (1576, 592)
After removing rows with missing label - these give nothing:  (1567, 592)
```

In [441]:
```
# Convenient function to understand data in a hurry.
def get_quality_report(df):
    result = pd.DataFrame()
    result['Data Type'] = df.dtypes
    result['Missing Values'] =df.isnull().sum()
    result['Present Values'] = df.count()
    result['Missing Value Ratio'] = result['Missing Values'] / df.index.size
    result['Unique Values'] = df.apply(lambda x: len(x.unique()))
    #unique_values = pd.DataFrame(unique_values, columns=['Unique Values'])
    result['Min'] = df.min()
    result['Max'] = df.max()
    result['Mean'] = df.mean()
    result['Median'] = df.median()
    result['Std dev'] = df.std()
    result['Var'] = df.var()
```

```
            return result

        # Correlation matrix function which requires
        def get_correlation_matrix(df,method='pearson'):
            correlation_matrix = df.corr(method=method)
            return correlation_matrix
```

Indexing with timestamp For convinience we will add the timestamp as an index

```
In [442]: df.index = df['Timestamp']
          df = df.drop('Timestamp', 1)
          print('Current shape of dataset: ', df.shape)

Current shape of dataset:  (1567, 591)
```

Key values Limit of missing value ratio before deleting entire dimension, lowest variance to keep, etc.:

```
In [443]: observationsInRow = 0.06 # remove rows with higher missing values ratio than
          missingRatioRemove = 0.6# remove columns with higher missing values ratio than
          varianceRemove = 0 # remove columns with lower variance than

          train_ratio = 0.66 # split dataset into two parts, n * 100 percent in train set
          opti_ratio = 0.6 # split test set into two parts, n * 100 percent in optimization set

          lines_to_show = 5
```

Splitting dataset Splitting dataset into train and test set. Removing all non-normals from train set. Removing label from train set. (Further splitting test set into optimization and validation set.) Splitting should be done by train_test_split when sklearn.model_selection package is available.

```
In [444]: train_size = round(train_ratio*len(df)) # REMOVE WHEN train_test_split is available.
          df_train = df[:train_size]              # REMOVE WHEN train_test_split is available.
          df_test = df[(train_size+1):-1]         # REMOVE WHEN train_test_split is available.

          # Removing non-normals
          df_train = df_train.loc[df_train['Yield_Pass_Fail'] == 0]
          df_train = df_train.drop('Yield_Pass_Fail', 1)
```

```
init_missing_values = df_train.isnull().sum().sum()
print('Initial missing value count, train set: ', init_missing_values)

# df_train, df_test = train_test_split(df, train_size = train_ratio)    # Uncomment when train_test_split is available.
# Split randomly with equal class ratio

print('Shape of df_train: ', df_train.shape, 'after non-normals and label are removed')
```

```
Initial missing value count, train set:  28339
Shape of df_train:  (957, 590) after non-normals and label are removed
```

Removing vectors (samples/ rows) missing more than n% of the values As can be seen in the print of the dataset, some rows/ vectors miss a lot of values. Each vector missing more than n% of the values are removed at first. When these are removed, we have less confusing samples.

```
In [445]: n_observations = round((1-observationsInRow) * len(df_train.columns))
          print('Previous shape of train set: ', df_train.shape)
          prev_length = len(df_train)
          df_train = df_train.dropna(thresh=n_observations)
          print('Current shape of train set: ', df_train.shape)
          print('The train set is reduced by ', prev_length - len(df_train), ' rows')
```

```
Previous shape of train set:  (957, 590)
Current shape of train set:  (686, 590)
The train set is reduced by  271  rows
```

Quality report To get an understanding of the dataset, we create a quality report and study it with respect to missing values and variance, at first. Learning from the quality report is an important step to avoid noise when using semi-supervised learning.

```
In [446]: qr_train = get_quality_report(df_train)
          print('A preview of the quality report. Remove .head() to see (almost) full report \n\n', qr_train.head(lines_to_show))
```

```
A preview of the quality report. Remove .head() to see (almost) full report

          Data Type  Missing Values  Present Values  Missing Value Ratio  \
Sensor#1   float64               5             681             0.007289
Sensor#2   float64               0             686             0.000000
```

```
Sensor#3   float64               0             686             0.000000
Sensor#4   float64               0             686             0.000000
Sensor#5   float64               0             686             0.000000

          Unique Values         Min         Max          Mean     Median  \
Sensor#1            678  2787.4900  3266.0400  3011.444728  3009.7100
Sensor#2            672  2162.8700  2846.4400  2494.151676  2497.2900
Sensor#3            298  2060.6600  2315.2667  2198.287452  2200.0666
Sensor#4            303   847.7976  3715.0417  1372.628160  1275.7793
Sensor#5            295     0.6815     3.8894     1.300375     1.2998

             Std dev            Var
Sensor#1   66.631357    4439.737713
Sensor#2   86.257874    7440.420891
Sensor#3   30.186035     911.196688
Sensor#4  376.245574  141560.731691
Sensor#5    0.403575       0.162873
```

In [447]:
```
# finding the 5 features with the most missing values
print('The %s sensors with the most missing values. \n\n'
      %lines_to_show, qr_train.nlargest(lines_to_show, 'Missing Value Ratio'))
```

```
The 5 sensors with the most missing values.

            Data Type  Missing Values  Present Values  Missing Value Ratio  \
Sensor#158   float64             614              72             0.895044
Sensor#159   float64             614              72             0.895044
Sensor#293   float64             614              72             0.895044
Sensor#294   float64             614              72             0.895044
Sensor#86    float64             600              86             0.874636

            Unique Values       Min        Max         Mean      Median  \
Sensor#158             71    0.0185     0.2876     0.053701     0.03750
Sensor#159             73  450.6001  2505.2998  1092.490331  1034.49975
Sensor#293             62    0.0061     0.0831     0.016554     0.01200
Sensor#294             73  161.3905   879.2260   356.165481   332.32990
Sensor#86              63    0.1068     0.1173     0.112916     0.11380
```

```
               Std dev            Var
Sensor#158    0.048462       0.002349
Sensor#159  389.571361  151765.844956
Sensor#293    0.014110       0.000199
Sensor#294  136.319180   18582.918887
Sensor#86     0.002705       0.000007
```

```
In [448]: # finding the 5 features with the smallest variance
          print('The %s sensors with the least missing values. \n\n'
                %lines_to_show, qr_train.nsmallest(lines_to_show, 'Var'))
```

```
The 5 sensors with the least missing values.

           Data Type  Missing Values  Present Values  Missing Value Ratio  \
Sensor#6    float64               0             686                  0.0
Sensor#14   float64               0             686                  0.0
Sensor#43   float64               0             686                  0.0
Sensor#50   float64               0             686                  0.0
Sensor#53   float64               0             686                  0.0

           Unique Values    Min    Max   Mean  Median  Std dev  Var
Sensor#6               1  100.0  100.0  100.0   100.0      0.0  0.0
Sensor#14              1    0.0    0.0    0.0     0.0      0.0  0.0
Sensor#43              1   70.0   70.0   70.0    70.0      0.0  0.0
Sensor#50              1    1.0    1.0    1.0     1.0      0.0  0.0
Sensor#53              1    0.0    0.0    0.0     0.0      0.0  0.0
```

No variance features As seen in the quality report, there are some features with no variance. These are likely to reduce the precision of the ML models instead of increasing it. The low variance features are therefore removed:

```
In [449]: qr_train = get_quality_report(df_train)
          low_var_cols = df_train.var()[df_train.var() <= 0].index.values
          var_reduction = 100*len(low_var_cols)/len(df_train)
          print('Number of columns with variance <=  %s : %s columns \n' %(varianceRemove, len(low_var_cols)))
          print('Previous shape of dataset: ',df_train.shape)
          df_train = df_train.drop(low_var_cols,1)
          print('Removed features with variance <= %s. New shape is %s' %(varianceRemove, df_train.shape))
```

```
          print('\nNumber of dimensions reduced by %s ' %len(low_var_cols))
          print('Dataset reduced by %.2f percent ' %round(var_reduction,2))
          # "%.2f" % round(a,2)

Number of columns with variance <=  0 : 122 columns

Previous shape of dataset:  (686, 590)
Removed features with variance <= 0. New shape is (686, 468)

Number of dimensions reduced by 122
Dataset reduced by 17.78 percent
```

Missing values From the quality report, we also see that the frame contains a lot of missing values. We therefore delete the columns with more missing values than within our threshold:

```
In [450]: high_missing_ratio_cols = df_train.columns[(df_train.isnull().sum() / len(df_train.index)) > missingRatioRemove]
          missing_reduction = 100*len(high_missing_ratio_cols)/len(df_train.index)
          print('Columns with missing value ratio >  %s : %s columns \n' %(missingRatioRemove, len(high_missing_ratio_cols)))
          print('Previous shape of dataset: ', df_train.shape)
          df_train = df_train.drop(high_missing_ratio_cols,1)
          print('Removed features with missing value ratio > %s. New shape is %s' %(missingRatioRemove, df_train.shape))

          print('\nNumber of dimensions reduced by %s ' %len(high_missing_ratio_cols))
          print('Dataset reduced by %.2f percent ' %round(missing_reduction,2))

Columns with missing value ratio >  0.6 : 20 columns

Previous shape of dataset:  (686, 468)
Removed features with missing value ratio > 0.6. New shape is (686, 448)

Number of dimensions reduced by 20
Dataset reduced by 2.92 percent
```

Still missing values After having removed a vast amount of both rows/ vectors and columns/ features containing an unnacceptable amount of NaN's, we now have decreased the number of missing values drastically. For the scaling/ normalization, a dataset with no NaN's is needed. Since these data are not continuous, fixed rate or delta sampled, we have no reason to assume any specific values for the empty fields! Therfore we can freely test which apporach gives the best results. Some typical approaches are filling with; 0, mean of feature, median of feature or a randomly selected value from another sample. We first try with 0.

```
In [451]: current_missing_count = df_train.isnull().sum().sum()
          reduction_count = init_missing_values - current_missing_count
          print('We have reduced the missing value count from %s to %s NaNs. A reduction of %s or %.2f percent.'
                %(init_missing_values, current_missing_count, reduction_count, round(100*reduction_count/init_missing_values,2)))

          df_train = df_train.fillna(0)
          print('Filling missing values with 0, there is now no missing values and we are ready for pre-processing')
```

```
We have reduced the missing value count from 28339 to 4981 NaNs. A reduction of 23358 or 82.42 percent.
Filling missing values with 0, there is now no missing values and we are ready for pre-processing
```

Scaling/ normalization Normalization is done to give each feature an equal amount of importance. We don't want the algorith to think that a temperature of 10 degrees is less important than a pressure of 1000mmH2O, just because 10 is much less than 1000. Some normalization approches exist. Ranges such as -1 to 1, 0 to 1 are the most typical for linear scaling. Here we use 0 to 1.

```
In [452]: scaler = MinMaxScaler(feature_range=(0,1))
          np_train_norm = scaler.fit_transform(df_train)
          df_train_norm = pd.DataFrame(np_train_norm, columns=df_train.columns, index=df_train.index)
          df_train_norm.head(lines_to_show)
```

```
                     Sensor#1  Sensor#2  Sensor#3  Sensor#4  Sensor#5  \
Timestamp
19/07/2008 14:43:00  0.915090  0.463786  0.543479  0.021622  0.199165
19/07/2008 17:53:00  0.902086  0.394941  0.678327  0.166963  0.265563
19/07/2008 19:45:00  0.936572  0.771362  0.739336  0.054642  0.033324
19/07/2008 20:24:00  0.908648  0.640169  0.739336  0.054642  0.033324
19/07/2008 21:35:00  0.923476  0.388402  0.739336  0.054642  0.033324

                     Sensor#7  Sensor#8  Sensor#9  Sensor#10  Sensor#11  \
Timestamp
19/07/2008 14:43:00  0.731149  0.452381  0.638590   0.329085   0.436578
19/07/2008 17:53:00  0.596255  0.595238  0.725612   0.666280   0.566372
19/07/2008 19:45:00  0.801522  0.198413  0.696820   0.654693   0.495575
19/07/2008 20:24:00  0.801522  0.198413  0.740868   0.601390   0.387906
19/07/2008 21:35:00  0.801522  0.198413  0.745810   0.657010   0.557522

                        ...      Sensor#581  Sensor#582  Sensor#583  \
Timestamp               ...
```

```
19/07/2008 14:43:00     ...         0.293333    0.104472    0.664577
19/07/2008 17:53:00     ...         0.346667    0.062261    0.536050
19/07/2008 19:45:00     ...         0.420000    0.134448    0.645768
19/07/2008 20:24:00     ...         0.300000    0.157964    0.673981
19/07/2008 21:35:00     ...         0.486667    0.127655    0.592476

                    Sensor#584  Sensor#585  Sensor#586  Sensor#587  \
Timestamp
19/07/2008 14:43:00    0.008174    0.006856    0.007906    0.410774
19/07/2008 17:53:00    0.026672    0.025465    0.026116    0.646465
19/07/2008 19:45:00    0.008819    0.015671    0.008643    0.414141
19/07/2008 20:24:00    0.023016    0.027424    0.022193    0.257576
19/07/2008 21:35:00    0.018714    0.019589    0.018322    0.427609

                    Sensor#588  Sensor#589  Sensor#590
Timestamp
19/07/2008 14:43:00    0.211881    0.214815    0.104472
19/07/2008 17:53:00    0.215842    0.274074    0.062261
19/07/2008 19:45:00    0.300990    0.355556    0.134448
19/07/2008 20:24:00    0.162376    0.222222    0.157964
19/07/2008 21:35:00    0.295050    0.429630    0.127655

[5 rows x 448 columns]
```

Pre-processing test set Since we now know how to pre-process the data, only based on the knowledge we extracted from the train set, we can apply the same processing to the test set before splitting it into optimization and validation sets.

```
In [453]: # putting labels in an own series before pre-processing
          test_label = df_test['Yield_Pass_Fail']
          df_test = df_test.drop(['Yield_Pass_Fail'],1)

          # removing vectors
          n_observations_test = round((1-observationsInRow) * (len(df_test.columns)))
          df_test = df_test.dropna(thresh=n_observations_test)

          # removing low variance features (based on which had low var in train set)
          df_test = df_test.drop(low_var_cols,1)
```

```
        # removing high missing value ratio features (based on high missing value ratio columns in train set)
        df_test = df_test.drop(high_missing_ratio_cols,1)

        # replacing still missing values
        df_test = df_test.fillna(0)

        # scaling/ normalization
        np_test_norm = scaler.transform(df_test)
        df_test_norm = pd.DataFrame(np_test_norm, columns=df_test.columns, index=df_test.index)


        print('Pre-processed test set\n')
        print(df_test_norm.head(lines_to_show))
        print('\n\nPre-processed train set\n')
        print(df_train_norm.head(lines_to_show))
```

```
Pre-processed test set

                     Sensor#1  Sensor#2  Sensor#3  Sensor#4  Sensor#5  \
Timestamp
23/09/2008 13:37:00  0.915693  0.389104  0.740733  0.424542  0.448923
23/09/2008 15:23:00  0.912043  0.407127  0.762160  0.395328  0.443624
23/09/2008 15:33:00  0.929869  0.417675  0.527463  0.090828  0.176471
23/09/2008 15:58:00  0.920610  0.518191  0.762160  0.395328  0.443624
23/09/2008 18:10:00  0.906489  0.596267  0.571408  0.358948  0.429939

                     Sensor#7  Sensor#8  Sensor#9  Sensor#10  Sensor#11  \
Timestamp
23/09/2008 13:37:00  0.434662  0.492063  0.380748   0.345307   0.359882
23/09/2008 15:23:00  0.263234  0.507937  0.605071   0.520278   0.573746
23/09/2008 15:33:00  0.662490  0.333333  0.695960   0.501738   0.392330
23/09/2008 15:58:00  0.263234  0.507937  0.857542   0.390498   0.672566
23/09/2008 18:10:00  0.440086  0.682540  0.737215   0.878331   0.384956

                        ...      Sensor#581  Sensor#582  Sensor#583  \
Timestamp               ...
23/09/2008 13:37:00     ...        0.420000    0.112640    0.583072
23/09/2008 15:23:00     ...        0.000000    0.000000    0.589342
```

```
23/09/2008 15:33:00     ...        0.000000    0.000000    0.514107
23/09/2008 15:58:00     ...        0.620000    0.068434    0.758621
23/09/2008 18:10:00     ...        0.493333    0.064658    0.498433

                    Sensor#584  Sensor#585  Sensor#586  Sensor#587  \
Timestamp
23/09/2008 13:37:00    0.009249    0.009794    0.009134    0.456229
23/09/2008 15:23:00    0.002796    0.005877    0.002876    0.456229
23/09/2008 15:33:00    0.023016    0.033301    0.022509    0.456229
23/09/2008 15:58:00    0.009895    0.012733    0.009588    0.922559
23/09/2008 18:10:00    0.017423    0.016650    0.017272    0.806397

                    Sensor#588  Sensor#589  Sensor#590
Timestamp
23/09/2008 13:37:00    0.277228    0.355556    0.112640
23/09/2008 15:23:00    0.277228    0.355556    0.112640
23/09/2008 15:33:00    0.277228    0.355556    0.112640
23/09/2008 15:58:00    0.401980    0.577778    0.068434
23/09/2008 18:10:00    0.312871    0.437037    0.064658

[5 rows x 448 columns]


Pre-processed train set

                    Sensor#1  Sensor#2  Sensor#3  Sensor#4  Sensor#5  \
Timestamp
19/07/2008 14:43:00  0.915090  0.463786  0.543479  0.021622  0.199165
19/07/2008 17:53:00  0.902086  0.394941  0.678327  0.166963  0.265563
19/07/2008 19:45:00  0.936572  0.771362  0.739336  0.054642  0.033324
19/07/2008 20:24:00  0.908648  0.640169  0.739336  0.054642  0.033324
19/07/2008 21:35:00  0.923476  0.388402  0.739336  0.054642  0.033324

                    Sensor#7  Sensor#8  Sensor#9  Sensor#10  Sensor#11  \
Timestamp
19/07/2008 14:43:00  0.731149  0.452381  0.638590   0.329085   0.436578
19/07/2008 17:53:00  0.596255  0.595238  0.725612   0.666280   0.566372
```

```
19/07/2008 19:45:00  0.801522  0.198413  0.696820   0.654693   0.495575
19/07/2008 20:24:00  0.801522  0.198413  0.740868   0.601390   0.387906
19/07/2008 21:35:00  0.801522  0.198413  0.745810   0.657010   0.557522

                        ...      Sensor#581  Sensor#582  Sensor#583  \
Timestamp               ...
19/07/2008 14:43:00     ...        0.293333    0.104472    0.664577
19/07/2008 17:53:00     ...        0.346667    0.062261    0.536050
19/07/2008 19:45:00     ...        0.420000    0.134448    0.645768
19/07/2008 20:24:00     ...        0.300000    0.157964    0.673981
19/07/2008 21:35:00     ...        0.486667    0.127655    0.592476

                    Sensor#584  Sensor#585  Sensor#586  Sensor#587  \
Timestamp
19/07/2008 14:43:00    0.008174    0.006856    0.007906    0.410774
19/07/2008 17:53:00    0.026672    0.025465    0.026116    0.646465
19/07/2008 19:45:00    0.008819    0.015671    0.008643    0.414141
19/07/2008 20:24:00    0.023016    0.027424    0.022193    0.257576
19/07/2008 21:35:00    0.018714    0.019589    0.018322    0.427609

                    Sensor#588  Sensor#589  Sensor#590
Timestamp
19/07/2008 14:43:00    0.211881    0.214815    0.104472
19/07/2008 17:53:00    0.215842    0.274074    0.062261
19/07/2008 19:45:00    0.300990    0.355556    0.134448
19/07/2008 20:24:00    0.162376    0.222222    0.157964
19/07/2008 21:35:00    0.295050    0.429630    0.127655

[5 rows x 448 columns]
```

Varify that datasets are equal and re-attach label The re-attached label must be changed to the labels that One-Class SVM operates with, inliers are labeled 1, while outliers are labeled -1. In our dataset, inliers are labeled 0, while outlisers are labeled 1.

```
In [454]: if (df_test_norm.columns == df_train_norm.columns).all():
              print('Test and train column labels are equal!')

          df_test_norm = df_test_norm.join(test_label)
          print('Label re-attached to test set')
```

```
df_test_norm['Yield_Pass_Fail'] = df_test_norm['Yield_Pass_Fail'].replace(1,-1)
df_test_norm['Yield_Pass_Fail'] = df_test_norm['Yield_Pass_Fail'].replace(0,1)
```

```
Test and train column labels are equal!
Label re-attached to test set
```

Splitting test set For optimization, we need to split the labelled test set into a part for optimization and a part for validation. As mentioned, we cannot optimize the algorithm with the same data as we validate with. After splitting the set, the labels for each set is un-attached for scoring.

```
In [455]: opti_vali_size = round(opti_ratio*len(df_test))      # REMOVE WHEN train_test_split is available.
          df_optimize = df_test_norm[:opti_vali_size]          # REMOVE WHEN train_test_split is available.
          df_validate = df_test_norm[(opti_vali_size+1):-1]    # REMOVE WHEN train_test_split is available.

          optimize_label = df_optimize['Yield_Pass_Fail']
          df_optimize = df_optimize.drop(['Yield_Pass_Fail'],1)
          # optimize_label = optimize_label.replace(1,-1)
          # optimize_label = optimize_label.replace(0,1)

          validate_label = df_validate['Yield_Pass_Fail']
          df_validate = df_validate.drop(['Yield_Pass_Fail'],1)
          # validate_label = validate_label.replace(1,-1)
          # validate_label = validate_label.replace(0,1)


          print('Shape of df_optimize: ',df_optimize.shape,' including label')
          print('Shape of df_validate:',df_validate.shape,' including label')

          # print(df_optimize[[1]].head(10))
          # print(optimize_label.head(10))

          # df_optimize, df_validate = train_test_split(df_test, train_size = opti_ratio)
          # Uncomment when train_test_split is available.
          # Split randomly with equal class ratio
```

```
Shape of df_optimize:  (296, 448)  including label
Shape of df_validate: (214, 448)  including label
```

Initial ML with One-Class SVM with default settings

```
In [456]: start = time.clock()
          clf = OneClassSVM()
          clf.fit(df_train_norm)
          train_time = format(time.clock() - start, '.4f')
          y_pred_vali = clf.predict(df_validate)
```

Initial Scoring of One-Class SVM with default settings

```
In [457]: init_precision = metrics.precision_score(validate_label, y_pred_vali, pos_label=-1, average='binary')
          print('Current PRECISION: %s, after a training time of: %s sec' %(init_precision, train_time))

Current PRECISION: 0.0877192982456, after a training time of: 0.2042 sec
```

ML with One-Class SVM with good settings from another dataset

```
In [458]: start = time.clock()
          clf = OneClassSVM(kernel='rbf', nu=0.001, gamma=0.00016)
          clf.fit(df_train_norm)
          train_time = format(time.clock() - start, '.4f')
          y_pred_vali = clf.predict(df_validate)
```

Scoring of One-Class SVM with good settings from another dataset

```
In [459]: precision = metrics.precision_score(validate_label, y_pred_vali, pos_label=-1, average='binary')
          print('Using the best paramteres from simulator test, PRECISION: %s, after training for %s sec'
                %(precision, train_time))

          print('This is a change of: %.4f points from the initial test.'
                %(round(precision-init_precision,4)))

Using the best paramteres from simulator test, PRECISION: 0.0683760683761, after training for 0.0043 sec
This is a change of: -0.0193 points from the initial test.
```

Experiment C

Classification

- SECOM_preprosessing
- GradientBoosting
- kNN
- MLPC

Experiment C - SECOM_preprosessing.py

```
from pandas import read_csv
from pandas import set_option as so
from numpy import set_printoptions as sp
from collections import Counter as count
from sklearn.preprocessing import MinMaxScaler as mms
import lib.KnowledgeExtractor as ke
import pandas as pd

# Set print options
sp(precision=3, linewidth=250)
so('display.width', 250)

# Import data set
secom = read_csv('SECOM_org.csv', index_col='Timestamp')
print(secom.shape, secom.shape[0]*secom.shape[1])
print(ke.get_NaN_count(secom))

# Drop rows with more than 6% missing values
secom = secom.dropna(thresh=(591*0.94))
print(secom.shape, secom.shape[0]*secom.shape[1])
print(ke.get_NaN_count(secom))

# Fill missing values in columns with 0 variance
info = pd.DataFrame({'Variance':secom.var()})
var_0_col = info.loc[info['Variance'] == 0].index.values
qr = ke.get_quality_report(secom)['Max']
for col in var_0_col:
    secom[col] = qr[col]
print(secom.shape, secom.shape[0]*secom.shape[1])
print(ke.get_NaN_count(secom))

# Drop columns with number of missing values > 10
qr = ke.get_quality_report(secom)
secom = secom.drop(qr.loc[qr['Missing Values'] > 10].index.values ,axis=1)
print(secom.shape, secom.shape[0]*secom.shape[1])
print(ke.get_NaN_count(secom))

# Drop rows with missing values
secom = secom.dropna()
print(secom.shape, secom.shape[0]*secom.shape[1])
print(ke.get_NaN_count(secom))

# Normalize data set
scaler = mms(feature_range=(-1, 1))
norm = scaler.fit_transform(secom)
secom = pd.DataFrame(norm, columns=secom.columns.values, index=secom.index.values)

print(secom.shape, secom.shape[0]*secom.shape[1])
print(count(secom['Yield_Pass_Fail']))
```

Experiment C - GradientBoosting.py

```
from pandas import read_csv
from pandas import set_option as so
from numpy import set_printoptions as sp
from sklearn.ensemble import GradientBoostingClassifier as gbc
from sklearn.preprocessing import MinMaxScaler as mms
from sklearn.model_selection import train_test_split as tts
from sklearn.model_selection import GridSearchCV
from collections import Counter as count
import lib.ResultAnalyzer as ra
import pandas as pd
from sklearn.decomposition import PCA


# Set print options
sp(precision=3, linewidth=250)
so('display.width', 250)

############################################################
# Simulator dataset
############################################################
print('Simulator dataset')
# Import data set
sim_df = read_csv('SimulatorDataLabeled.csv', index_col='DateTime')

# Drop columns with only one value (static)
sim_df = sim_df.drop(['StorageRef_Done', 'ReactorSteamValve_InMaintenance'], axis=1)

# Normalize dataset
scaler = mms(feature_range=(-1, 1))
scaler.fit(sim_df['2017-03-20': '2017-03-24 09:00'])
sim_df_norm = scaler.transform(sim_df)
sim_df = pd.DataFrame(sim_df_norm, columns=sim_df.columns.values, index=sim_df.index.values)

# Split data set into training and testing
X_train, X_test, Y_train, Y_test = tts(sim_df[sim_df.columns[0:-1]], sim_df['Class'], train_size=0.66,
                                       stratify=sim_df['Class'])

# Grid search parameters
max_depth = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
param_grid = dict(max_depth=max_depth)

# Create ML model
clf = gbc(random_state=7)

# Set up and train the grid search
grid = GridSearchCV(estimator=clf, param_grid=param_grid)
grid.fit(X_train, Y_train)

# Print information from the grid search
print('Grid search information')
print(grid.best_score_)
print(grid.best_estimator_.max_depth)

# Use the best estimator from the grid search to predict
result = grid.best_estimator_.predict(X_test)

# Validate the prediction
validate = pd.DataFrame()
validate['Y_test'] = Y_test
validate['Y_pred'] = result
print(ra.get_confusion_matrix(validate))
ra.evaluate(validate)
print(count(Y_test))

print('\n')

############################################################
# SECOM dataset
############################################################
print('SECOM dataset')
# Import data set
secom_df_pre = read_csv('Secom_preprocessed.csv', index_col=0)

# Split into train and test
X_train_pre, X_test_pre, Y_train_pre, Y_test_pre = tts(secom_df_pre[secom_df_pre.columns[0:-1]], secom_df_pre[
    'Yield_Pass_Fail'], train_size=0.66, stratify=secom_df_pre['Yield_Pass_Fail'])

# Grid search parameters
max_depth = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```
param_grid = dict(max_depth=max_depth)

# Create ML model
clf_secom = gbc()

# Set up and train grid search
grid = GridSearchCV(estimator=clf_secom, param_grid=param_grid)
grid.fit(X_train_pre, Y_train_pre)

# Print grid search information
print('Grid search information')
print(grid.best_score_)
print(grid.best_estimator_.max_depth)

# Use the best estimator from the grid search to predict
result = grid.best_estimator_.predict(X_test_pre)

# Validate the prediction
validate = pd.DataFrame()
validate['Y_test'] = Y_test_pre
validate['Y_pred'] = result
print(ra.get_confusion_matrix(validate))
ra.evaluate(validate)
print(count(Y_test))

# Extract max depth to use in PCA
max_depth = grid.best_estimator_.max_depth

print('\n')

# Use PCA to reduce dimensions
for x in [10, 20, 40, 80, 160]:
    # Create PCA object and transform dataset
    pca = PCA(x)
    pca_df = pca.fit_transform(secom_df_pre[secom_df_pre.columns[0:-1]])

    # Put PCA data into a dataframe
    pca_df = pd.DataFrame(pca_df, index=secom_df_pre.index.values)
    pca_df['Yield_Pass_Fail'] = secom_df_pre['Yield_Pass_Fail'].values

    # Split data set
    X_train, X_test, Y_train, Y_test = tts(pca_df[pca_df.columns[0:-1]], pca_df[pca_df.columns[-1]], train_size=0.66,
                                           stratify=pca_df['Yield_Pass_Fail'], random_state=7)

    # Create ML model and train
    clf1 = gbc(max_depth=max_depth)
    clf1.fit(X_train, Y_train)

    # Predict from model
    result = clf1.predict(X_test)

    # Validate the prediction
    validate = pd.DataFrame()
    validate['Y_test'] = Y_test
    validate['Y_pred'] = result
    print('PCA comp: ', x)
    print(ra.get_confusion_matrix(validate))
    ra.evaluate(validate)
    print(count(Y_test))
```

Experiment C - kNN_Supervised.py

```
from pandas import read_csv
from pandas import set_option as so
from numpy import set_printoptions as sp
from sklearn.neighbors import KNeighborsClassifier as knc
from sklearn.preprocessing import MinMaxScaler as mms
from sklearn.model_selection import train_test_split as tts
import lib.ResultAnalyzer as ra
import pandas as pd
from sklearn.decomposition import PCA

# Set print options
sp(precision=3, linewidth=250)
so('display.width', 250)

# Neighbour set
n_neighbours = [1, 2, 10, 15]

########################################################################################################################
# Simulator dataset
########################################################################################################################
print('Simulator dataset')
# Import dataset
sim_df = read_csv('SimulatorDataLabeled.csv', index_col=0)
print(sim_df.shape)

# Drop columns with only one value (static)
sim_df = sim_df.drop(['StorageRef_Done', 'ReactorSteamValve_InMaintenance'], axis=1)

# Normalize data set
scaler = mms(feature_range=(-1,1))
scaler.fit(sim_df['2017-03-20' : '2017-03-24 09:00'])
sim_df_norm = scaler.transform(sim_df)
sim_df = pd.DataFrame(sim_df_norm, columns=sim_df.columns.values, index=sim_df.index.values)

# Split data set into training and testing
X_train, X_test, Y_train, Y_test = tts(sim_df[sim_df.columns[0:-1]], sim_df['Class'], train_size=0.66,
                                       stratify=sim_df['Class'])

# Test neighbour set
for x in n_neighbours:
    # Create and train  ML model
    clf = knc(n_neighbors=x, algorithm='kd_tree')
    clf.fit(X_train, Y_train)

    # Predict from model
    result = clf.predict(X_test)

    # Validate the prediction
    validate = pd.DataFrame()
    validate['Y_true'] = Y_test
    validate['Y_pred'] = result
    print('n_neighbors = %d' % x)
    print(ra.get_confusion_matrix(validate))
    ra.evaluate(validate)

print('\n')

########################################################################################################################
# SECOM dataset
########################################################################################################################
print('SECOM dataset')
# Import data set
secom_df_pre = read_csv('Secom_preprocessed.csv', index_col=0)

df = secom_df_pre.drop(secom_df_pre.var().loc[secom_df_pre.var() == 0].index.values, axis=1)


# Split into train and test
X_train_pre, X_test_pre, Y_train_pre, Y_test_pre = tts(secom_df_pre[secom_df_pre.columns[0:-1]], secom_df_pre[
    'Yield_Pass_Fail'], train_size=0.66, stratify=secom_df_pre['Yield_Pass_Fail'])

# Test neighbour set
for x in n_neighbours:
    # Create train ML model
    clf_pre = knc(n_neighbors=x, algorithm='brute')
    clf_pre.fit(X_train_pre, Y_train_pre)
```

```
    # Predict from model
    result_pre = clf_pre.predict(X_test_pre)

    # Validate model
    validate_pre = pd.DataFrame()
    validate_pre['Y_true'] = Y_test_pre
    validate_pre['Y_pred'] = result_pre
    print('n_neighbors = %d' % x)
    print(ra.get_confusion_matrix(validate_pre))
    ra.evaluate(validate_pre)

print('\n')

# Use PCA to reduce dimantions
for x in [10, 20, 40, 80, 160]:
    # Create PCA object and transform dataset
    pca = PCA(x)
    pca_df = pca.fit_transform(secom_df_pre[secom_df_pre.columns[0:-1]])
    pca_df = pd.DataFrame(pca_df, index=secom_df_pre.index.values)
    pca_df['Yield_Pass_Fail'] = secom_df_pre['Yield_Pass_Fail'].values

    # Split data
    X_train_pca, X_test_pca, Y_train_pca, Y_test_pca = tts(pca_df[pca_df.columns[0:-1]], pca_df[
        'Yield_Pass_Fail'], train_size=0.66, stratify=pca_df['Yield_Pass_Fail'])

    # Create ML model
    clf_pca = knc(n_neighbors=1, algorithm='kd_tree')
    clf_pca.fit(X_train_pca, Y_train_pca)

    # Predict from model
    result_pca = clf_pca.predict(X_test_pca)

    # Validate the prediction
    validate_pca = pd.DataFrame()
    validate_pca['Y_true'] = Y_test_pca
    validate_pca['Y_pred'] = result_pca
    print('PCA comp: ', x)
    print(ra.get_confusion_matrix(validate_pca))
    ra.evaluate(validate_pca)
```

Experiment C - MLPC.py

```
from sklearn.neural_network import MLPClassifier
from pandas import read_csv
import pandas as pd
from numpy import set_printoptions as sp
from pandas import set_option as so
from sklearn.preprocessing import MinMaxScaler as mms
from sklearn.model_selection import train_test_split as tts
import lib.ResultAnalyzer as ra
from collections import Counter as count
from sklearn.model_selection import GridSearchCV
from sklearn.decomposition import PCA

# Set print options
sp(precision=3, linewidth=250)
so('display.width', 250)

#########################################################################################################################
# SECOM dataset
#########################################################################################################################
print('SECOM dataset')
# Import data set
df = read_csv('Secom_preprocessed.csv', index_col=0)

#  Remove features with 0 variance
df = df.drop(df.var().loc[df.var() == 0].index.values, axis=1)

# Split data set
X_train, X_test, Y_train, Y_test = tts(df[df.columns[0:-1]], df[df.columns[-1]], train_size=0.66, stratify=df[
    'Yield_Pass_Fail'], random_state=7)

# Grid search parameters
hidden_layers = [(1000, 500), (500, 250), (250, 100), (150, 50), (100, 50), (100, 20)]
param_grid = dict(hidden_layer_sizes=hidden_layers)

# Create ML model
model = MLPClassifier(solver='lbfgs', random_state=7)

# Set up and train the grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid.fit(X_train, Y_train)

# Print information from the grid search
print('Grid search information')
print(grid.best_score_)
print(grid.best_estimator_.hidden_layer_sizes)

# Use the best estimator from the grid search to predict
result = grid.best_estimator_.predict(X_test)

# Validate the prediction
validate = pd.DataFrame()
validate['Y_test'] = Y_test
validate['Y_pred'] = result
print(ra.get_confusion_matrix(validate))
print(count(Y_test))

# Extract hidden layer size to use in PCA
hidden_layer = grid.best_estimator_.hidden_layer_sizes

print('\n')

# Use PCA to reduce dimensions
for x in [10, 20, 40, 80, 160]:
    # Create PCA object and transform dataset
    pca = PCA(x)
    pca_df = pca.fit_transform(df[df.columns[0:-1]])

    # Put PCA data into a dataframe
    pca_df = pd.DataFrame(pca_df, index=df.index.values)
    pca_df['Yield_Pass_Fail'] = df['Yield_Pass_Fail'].values

    # Split data set
    X_train, X_test, Y_train, Y_test = tts(pca_df[pca_df.columns[0:-1]], pca_df[pca_df.columns[-1]], train_size=0.66, stratify=pca_df[
        'Yield_Pass_Fail'], random_state=7)

    # Create ML model and train
    clf1 = MLPClassifier(solver='lbfgs', random_state=7, hidden_layer_sizes=hidden_layer)
    clf1.fit(X_train, Y_train)

    # Predict from model
    result = clf1.predict(X_test)

    # Validate the prediction
    validate = pd.DataFrame()
    validate['Y_test'] = Y_test
    validate['Y_pred'] = result
    print('PCA comp: ', x)
    print(ra.get_confusion_matrix(validate))
    ra.evaluate(validate)
    print(count(Y_test), '\n')
```

```
print('\n')

# ##############################################################################################################################
# # Simulator dataset
# ##############################################################################################################################
print('Simulator dataset')
# Import data set
sim_df = read_csv('SimulatorDataLabeled.csv', index_col=0)

# Drop columns with only one value (static)
sim_df = sim_df.drop(['StorageRef_Done', 'ReactorSteamValve_InMaintenance'], axis=1)

# Normalize data set
scaler = mms(feature_range=(-1, 1))
scaler.fit(sim_df['2017-03-20': '2017-03-24 09:00'])
sim_df_norm = scaler.transform(sim_df)
sim_df = pd.DataFrame(sim_df_norm, columns=sim_df.columns.values, index=sim_df.index.values)

# Split data set into training and testing
X_train, X_test, Y_train, Y_test = tts(sim_df[sim_df.columns[0:-1]], sim_df['Class'], train_size=0.66,
                                       stratify=sim_df['Class'])

# Grid search parameters
hidden_layers = [(100, 50), (80, 40), (60, 30), (50, 25), (40, 20), (25, 10)]
param_grid = dict(hidden_layer_sizes=hidden_layers)

# Create ML model
model = MLPClassifier(solver='lbfgs', random_state=7)

# Set up and train the grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid.fit(X_train, Y_train)

# Print information from the grid search
print('Grid search information')
print(grid.best_score_)
print(grid.best_estimator_.hidden_layer_sizes)

# Use the best estimator from the grid search to predict
result = grid.best_estimator_.predict(X_test)

# Validate the prediction
validate = pd.DataFrame()
validate['Y_test'] = Y_test
validate['Y_pred'] = result
print(ra.get_confusion_matrix(validate))
ra.evaluate(validate)
print(count(Y_test))
```

Experiment D

Event Prediction

- Turbofan Engine Degradation
  - Non-linear RUL
  - Classification
- Battery Life Prediction

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import random
import time

from mpl_toolkits.mplot3d import Axes3D
from sklearn import decomposition
from sklearn.cluster import KMeans
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor

from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import MinMaxScaler


# Generate a assortment of statistics from the data
def get_quality_report(df):
    result = pd.DataFrame()
    result['Data Type'] = df.dtypes
    result['Missing Values'] =df.isnull().sum()
    result['Present Values'] = df.count()
    result['Missing Value Ratio'] = result['Missing Values'] / df.index.size
    result['Unique Values'] = df.apply(lambda x: len(x.unique()))
    result['Min'] = df.min()
    result['Max'] = df.max()
    result['Mean'] = df.mean()
    result['Median'] = df.median()
    result['Std dev'] = df.std()
    return result


# Create scatter plot of the three operational settings
def opSettingsPCA3DPlot():
    fig = plt.figure(1)
    plt.clf()
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

    plt.cla()
    pca = decomposition.PCA(n_components=3)
    dataPCA = pca.fit(data[[0, 1, 2]]).transform(data[[0, 1, 2]])

    ax.scatter(dataPCA[:, 0], dataPCA[:, 1], dataPCA[:, 2], cmap=plt.cm.spectral)
    ax.set_xlabel("Operational Setting 1")
    ax.set_ylabel("Operational Setting 2")
    ax.set_zlabel("Operational Setting 3")
    plt.title('Scatter plot of operational settings')
    return


# Create scatter plot of dimensionality reduced dataset - 3 dimensions
def PCA3DPlot():
    fig = plt.figure(2)
    plt.clf()
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

    plt.cla()
    pca = decomposition.PCA(n_components=3)
    pca.fit(data)
    dataPCA = pca.fit(data).transform(data)

    ax.scatter(dataPCA[:, 0], dataPCA[:, 1], dataPCA[:, 2], cmap=plt.cm.spectral)
    plt.title('PCA reduced - 3 components')


# Create scatter plot of dimensionality reduced dataset - 2 dimensions
def PCA2DPlot():
    fig = plt.figure(3)
    plt.clf()

    plt.cla()
    pca = decomposition.PCA(n_components=2)
    dataPCA = pca.fit(data).transform(data)

    plt.tick_params(axis='both', which='both', bottom='off', top='off', labelbottom='off',
                    right='off', left='off', labelleft='off')

    plt.scatter(dataPCA[:, 0], dataPCA[:, 1])


# Create plot of cluster centroids from operational settings in 3D
def kmeansPlot():
    fig = plt.figure(4)
    plt.clf()

    plt.cla()
    pca = decomposition.PCA(n_components=2)
    dataPCA = pca.fit(data).transform(data)
    kmeans = KMeans(n_clusters=6).fit(dataPCA)
    centroids = kmeans.cluster_centers_

    plt.tick_params(axis='both', which='both', bottom='off', top='off', labelbottom='off',
                    right='off', left='off', labelleft='off')

    plt.scatter(dataPCA[:, 0], dataPCA[:, 1])
    plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', s=75, linewidths=2, color='w', zorder=10)
    plt.title('Cluster centres and PCA reduced data')
```

```
# Adds columns for time (nr. of cycles) spent in each of the 6 modes
def add_mode_duration(data):
    data['Mode 1'] = 0; data['Mode 2'] = 0; data['Mode 3'] = 0
    data['Mode 4'] = 0; data['Mode 5'] = 0; data['Mode 6'] = 0

    ind = data.index.get_level_values(0)
    units = len(ind.unique())
    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()
        column = np.zeros((unit_len, 6), int)

        for index, row in data.loc[unit].iterrows():
            mode = int(row['Operational Mode']) #row[-7]
            if index != 1:
                column[index - 1] = column[index - 2]
            column[index - 1][mode] = column[index - 1][mode] + 1

        data.loc[unit].loc[:, data.columns[-6::]] = column

    return data


# Transforms the data from one range to another (0 to 1 or based on mean and standard deviation in each feature)
def normalize(df, zero2one=False):
    result = df.copy()
    for feature_name in df.columns:
        if zero2one == True:
            min = df[feature_name].min()
            max = df[feature_name].max()
            result[feature_name] = (df[feature_name] - min) / (max - min)
        else:
            mean = df[feature_name].mean()
            std = df[feature_name].std()
            result[feature_name] = (df[feature_name] - mean) / (std)
    #print(result)
    return result


# Extracts all values in each operational mode, for every column, and normalise based on those
def normalize_by_mode(df, zero2one=False):
    result = df.copy()

    for i in range(6):
        mean = (df.loc[df['Operational Mode'] == i]).mean()
        std = (df.loc[df['Operational Mode'] == i]).std()
        if(zero2one==True):
            mean = (df.loc[df['Operational Mode'] == i]).min()
            std = (df.loc[df['Operational Mode'] == i]).max() - (df.loc[df['Operational Mode'] == i]).min()


        for feature_name in df.columns[0:-1]:
            result[feature_name].loc[result['Operational Mode']==i] = (df[feature_name].loc[df['Operational Mode']==i] - mean[feature_name])/(std[feat

    # Vurdere hva som skal gjøres når std blir 0. Kan eventuelt kjøre fillna.
    result = result.fillna(0)
    result[result.columns[-7::]]=normalize(result[result.columns[-7::]], zero2one=zero2one)
    result = result.fillna(0)

    return result


# Adds a linear labeled RUL column
def label_rul(data):
    data['RUL'] = 0

    ind = data.index.get_level_values(0)
    units = len(ind.unique())
    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()
        data.loc[unit,'RUL'] = range(unit_len,0,-1)

    return data


# Calculates the moving average of the predictions
def moving_average(predicted_rul, window_size):
    ma_predicted_rul = pd.DataFrame(data=predicted_rul, index=y_test.index)

    for j in range(len(testing_units)):
        ra_engine = ma_predicted_rul.loc[testing_units[j]].rolling(window_size).mean()

        ##### Sets NaN values to the same as in predicted array #####
        for i in range(window_size - 1):
            ra_engine.loc[i + 1] = ma_predicted_rul.loc[testing_units[j]].loc[i + 1]

        ma_predicted_rul.loc[testing_units[j]] = ra_engine.values

    return ma_predicted_rul


# Corrects the moving average filtered predictions by subtracting half the window size from each prediction
def corr_moving_average(predicted_rul, window_size):
    corr_ma = pd.DataFrame(data=predicted_rul, index=y_test.index)

    for i in range(len(corr_ma.values)):
        corr_ma.values[i] = corr_ma.values[i] - (window_size / 2)

    return corr_ma
```

```
if __name__ == '__main__':

    # Load training data set - The dataset can be downloaded from:
    #           https://ti.arc.nasa.gov/tech/dash/pcoe/prognostic-data-repository/
    data = pd.read_csv('train.txt', sep=" ", header=None, index_col=[0, 1],
                       names=["operational_setting 1",
                              "operational_setting 2", "operational_setting 3", "sensor 1", "sensor 2", "sensor 3",
                              "sensor 4", "sensor 5", "sensor 6", "sensor 7", "sensor 8", "sensor 9", "sensor 10",
                              "sensor 11", "sensor 12", "sensor 13", "sensor 14", "sensor 15", "sensor 16", "sensor 17",
                             "sensor 18", "sensor 19", "sensor 20", "sensor 21", "", ""])
    data = data[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]]

    ##################################################################################################
    #                                 Knowledge discovery
    ##################################################################################################

    # Cluster operational settings into 6 clusters (operational modes)
    kmeans = KMeans(n_clusters=6, random_state=4).fit(data[[0,1,2]])

    # Add operational mode as a feature
    operational_mode = kmeans.predict(data[[0,1,2]])
    data['Operational Mode']=operational_mode

    # Scatter plot of operational settings
    opSettingsPCA3DPlot()

    # Scatter plot of 3 component PCA on the dataset
    PCA3DPlot()

    # Plot of detected clusters from k-means and 2-dimensional PCA of the dataset
    kmeansPlot()
    plt.show()
    # Print Data Quality Report
    #print(get_quality_report(data))

    # Add mode duration as features - Cycles spent in each mode for every engine
    data = add_mode_duration(data)

    # Normalise by operational mode. (Comment to normalise with 0 to 1 scaling)
    data = normalize_by_mode(data, zero2one=True)

    # Add rolling features - Rolling difference and mean with window size = 4
    data['sensor 11 Diff'] = data['sensor 11'].diff(periods=4)
    data['sensor 3 Diff'] = data['sensor 3'].diff(periods=4)
    data['sensor 11 Mean'] = data['sensor 11'].rolling(4).mean()
    data['sensor 3 Mean'] = data['sensor 3'].rolling(4).mean()

    # Make sure that there are no NaN values in the dataset
    data = data.fillna(0)

    # Label data with remaining useful lifetime
    data = label_rul(data)

    # Drop the superfluous features
    data = data.drop(['operational_setting 1','operational_setting 2','operational_setting 3','sensor 1', 'sensor 2', 'sensor 5', 'sensor 6',
        'sensor 8', 'sensor 10', 'sensor 13', 'sensor 14', 'sensor 15', 'sensor 16',
       'sensor 17', 'sensor 18', 'sensor 19', 'sensor 20', 'sensor 21', 'Operational Mode', 'Mode 1', 'Mode 2','Mode 6'], axis=1)


    # Select units for training and testing by random (with seed). 67:33 split
    random.seed(a=5)
    rnd_vector=random.sample(range(1, 219), 218)
    training_units=rnd_vector[0:145]
    testing_units=rnd_vector[145::]
    test_set = data.loc[testing_units].copy()
    train_set = data.loc[training_units].copy()

    # x_train is training set with input parameters. Remove target
    x_train = train_set[train_set.columns[0:-1]]

    # x_test is test set with input parameters. Remove target
    x_test = test_set[test_set.columns[0:-1]]

    #### Normalise between 0 and 1 with scikit-learn. To run this, uncomment normalisation by mode.
        #scaler = MinMaxScaler(feature_range=(0,1))
        #x_train = scaler.fit_transform(x_train)
        #x_test = scaler.transform(x_test)

    # Target for training
    y_train = train_set.RUL

    # Target for testing
    y_test = test_set.RUL



    ##################################################################################################
    #                                      Machine learning
    ##################################################################################################

    # Grid search example for MLP optimisation
    '''
    mlp = MLPRegressor()
    param_grid = [{'hidden_layer_sizes': [(75), (50), (25), (10), (10, 5), 5],
                   'solver': ['lbfgs'], 'activation': ['tanh', 'relu', 'logistic'],
                   'alpha': [0.001, 0.0001, 0.00001],
                   'learning_rate': ['constant', 'invscaling', 'adaptive']}]
    param_grid = [{'hidden_layer_sizes': [(75), (50)]}]

    grid = GridSearchCV(mlp, param_grid, scoring='neg_mean_squared_error')
    grid.fit(x_train, y_train)

    print(grid.best_params_)
    model = grid.best_estimator_
```

```
'''

############################## MLP ####################################
# Create and train Multilayer Perceptron Regressor. Measure training time.
start = time.clock()
mlp = MLPRegressor(solver='lbfgs', alpha=1e-7, hidden_layer_sizes=(30),
                   activation='relu', random_state=None, momentum=0.9)
mlp.fit(x_train, y_train)
mlp_time = format((time.clock() - start), '.2f')

# Predict on test set and measure prediction time
start = time.clock()
mlp.predict(x_test)
mlp_pred_time = format((time.clock() - start), '.2f')

###################### Random Forest Regressor ########################
# Create and train Random Forest Regressor. Measure training time.
start = time.clock()
rndFor = RandomForestRegressor(n_estimators=25)
rndFor.fit(x_train, y_train)
rndFor_time = format((time.clock() - start), '.2f')

# Predict on test set and measure prediction time
start = time.clock()
rndFor.predict(x_test)
rndFor_pred_time = format((time.clock() - start), '.2f')

##################### Gradient Boosting Regressor ######################
# Create and train Gradient Boosting Regressor. Measure training time.
start = time.clock()
gbr = GradientBoostingRegressor(n_estimators=100, max_depth=8)
gbr.fit(x_train, y_train)
gbr_time = format((time.clock() - start), '.2f')

# Predict on test set and measure prediction time
start = time.clock()
gbr.predict(x_test)
gbr_pred_time = format((time.clock() - start), '.2f')

######################### K-neighbour Regressor ########################
# Create and train K-neighbour Regressor. Measure training time.
start = time.clock()
kneig = KNeighborsRegressor(algorithm='auto', leaf_size=95, n_neighbors=100, weights='distance')
kneig.fit(x_train, y_train)
kneig_time = format((time.clock() - start), '.2f')

# Predict on test set and measure prediction time
start = time.clock()
kneig.predict(x_test)
kneig_pred_time = format((time.clock() - start), '.2f')

########################## Evaluate ####################################
# Model, training time, prediction time and names for the models used
models = [mlp, rndFor, gbr, kneig]
model_times = [mlp_time, rndFor_time, gbr_time, kneig_time]
model_names = ['MLP', 'Random Forest Regressor',
               'Gradient Boosting Regressor', 'K-Neighbour Regressor']
pred_times = [mlp_pred_time, rndFor_pred_time, gbr_pred_time, kneig_pred_time]


# Initialise dataframe for predictions and for filtered predictions
preres = pd.DataFrame(y_test.values, index=y_test.index, columns=['Actual RUL'])
mares = pd.DataFrame(y_test.values, index=y_test.index, columns=['Actual RUL'])

# For loop to present result of each individual model and its prediction.
for j in range(len(models)):
    model = models[j]
    name = model_names[j]
    timet = model_times[j]
    pred_time = pred_times[j]


    temp_prediction = model.predict(x_test)
    window_size = 3
    # Prediction without filtering
    print('                  Model: ' + name + ' | MSE: ' + str(mean_squared_error(y_test, temp_prediction)) +
          ' |  MAE: ' + str(sklearn.metrics.mean_absolute_error(y_test, temp_prediction)) +
          ' | Training Time: ' + str(timet) + ' sec ' + '| Prediction Time: ' + str(pred_time))

    # Predictions with filtering
    print('          Filtered Model: ' + name + ' | MSE: '
          + str(mean_squared_error(y_test, moving_average(temp_prediction, window_size)))
          + ' |  MAE: ' + str(sklearn.metrics.mean_absolute_error(y_test, moving_average(temp_prediction, window_size))))

    # Predictions with corrected filtering
    print('Corrected Filtered Model: ' + name + ' | MSE: '
          + str(mean_squared_error(y_test, corr_moving_average(temp_prediction, window_size)))
          + ' |  MAE: ' + str(sklearn.metrics.mean_absolute_error(y_test, corr_moving_average(temp_prediction, window_size))))

    # Add predictions with each model
    preres[name] = model.predict(x_test)

    # Add filtered predictions with each model
    mares['Filtered ' + name] = moving_average(temp_prediction, window_size).loc[:, 0].values

    # Add Corrected filtered predictions with each model
    mares['Corrected Filtered ' + name] = corr_moving_average(temp_prediction, window_size).loc[:, 0].values


#########################################################################################################
#                                       Visualisation
#########################################################################################################

# Plot predictions on 6 different engines using MLP and K-Neighbor Regressor
```

```
fig = plt.figure(4)
plt.title('Engine Life Predictions - RUL')
subnum = 231
for k in range(30, 36):
    m = testing_units[k]
    plt.subplot(subnum)
    plt.plot(y_test.loc[m].index, y_test.loc[m].values)
    plt.plot(y_test.loc[m].index, mlp.predict(x_test.loc[m]))
    plt.plot(y_test.loc[m].index, kneig.predict(x_test.loc[m]))
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.title('Engine ' + str(m))
    plt.legend(['Actual RUL', 'MLP', 'K-Neighbor Regressor'])
    subnum = subnum + 1

# Plot predictions on 6 different engines using Random Forest Regressor and Gradient Boosting Regressor
fig = plt.figure(5)
plt.title('Engine Life Predictions - RUL')
subnum = 231
for k in range(30, 36):
    m = testing_units[k]
    plt.subplot(subnum)
    plt.plot(y_test.loc[m].index, y_test.loc[m].values)
    plt.plot(y_test.loc[m].index, rndFor.predict(x_test.loc[m]))
    plt.plot(y_test.loc[m].index, gbr.predict(x_test.loc[m]))
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.title('Engine ' + str(m))
    legends = ['Actual RUL']
    plt.legend(['Actual RUL', 'Random Forest Regressor', 'Gradient Boosting Regressor'])
    subnum = subnum + 1

# Predictions on 6 different engines using Gradient Boosting Regressor. Includes filtered predictions
fig = plt.figure(6)
plt.suptitle('Gradient Boosting Regressor Predictions - Filtering window size = 3')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['Gradient Boosting Regressor'], 'green')
    plt.plot(y_test.loc[j].index, mares.loc[j]['Filtered Gradient Boosting Regressor'], 'blue')
    plt.plot(y_test.loc[j].index, mares.loc[j]['Corrected Filtered Gradient Boosting Regressor'], 'red')
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'GBR Predicted', 'MA Filtered', 'Corrected MA'], loc='lower left')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

# Predictions on 6 different engines using MLP Regressor. Includes filtered predictions
fig = plt.figure(7)
plt.suptitle('MLP Regressor Predictions - Filtering window size = 3')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['MLP'], 'green')
    plt.plot(y_test.loc[j].index, mares.loc[j]['Filtered MLP'], 'blue')
    plt.plot(y_test.loc[j].index, mares.loc[j]['Corrected Filtered MLP'], 'red')
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'MLP Predicted', 'MA Filtered', 'Corrected MA'], loc='lower left')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

# Show created plots
plt.show()
```

Experiment D - Non-linear RUL.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
import time

from sklearn.cluster import KMeans
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error
import sklearn.ensemble

# Adds columns for time (nr. of cycles) spent in each of the 6 modes
def add_mode_cycles(data):
    data['Mode 1'] = 0; data['Mode 2'] = 0; data['Mode 3'] = 0
    data['Mode 4'] = 0; data['Mode 5'] = 0; data['Mode 6'] = 0

    ind = data.index.get_level_values(0)
    units = len(ind.unique())
    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()
        column = np.zeros((unit_len, 6), int)

        for index, row in data.loc[unit].iterrows():
            mode = int(row['Operational Mode'])
            if index != 1:
                column[index - 1] = column[index - 2]
            column[index - 1][mode] = column[index - 1][mode] + 1

        data.loc[unit].loc[:, data.columns[-6::]] = column

    return data

# Transforms the data from one range to another (0 to 1 or based on mean and standard deviation in each feature)
def normalize(df, zero2one=False):
    result = df.copy()
    for feature_name in df.columns:
        if zero2one == True:
            min = df[feature_name].min()
            max = df[feature_name].max()
            result[feature_name] = (df[feature_name] - min) / (max - min)
        else:
            mean = df[feature_name].mean()
            std = df[feature_name].std()
            result[feature_name] = (df[feature_name] - mean) / (std)

    return result

# Extracts all values in each operational mode, for every column, and normalise based on those
def normalize_by_mode(df, zero2one=False):
    result = df.copy()

    for i in range(6):
        mean = (df.loc[df['Operational Mode'] == i]).mean()
        std = (df.loc[df['Operational Mode'] == i]).std()
        if(zero2one == True):
            mean = (df.loc[df['Operational Mode'] == i]).min()
            std = (df.loc[df['Operational Mode'] == i]).max() - (df.loc[df['Operational Mode'] == i]).min()

        for feature_name in df.columns[0:-1]:
            result[feature_name].loc[result['Operational Mode'] == i] = \
                (df[feature_name].loc[df['Operational Mode'] == i] - mean[feature_name])/(std[feature_name])

    result = result.fillna(0)
    result[result.columns[-7::]] = normalize(result[result.columns[-7::]], zero2one=zero2one)
    result = result.fillna(0)

    return result

# Finds the shortest run length of all engines
def get_min_run_length(data):
    min_run_length = 1000

    ind = data.index.get_level_values(0)
    units = len(ind.unique())

    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()
```

```
        if min_run_length >= unit_len:
            min_run_length = unit_len

    return min_run_length

# Adds a linear labeled RUL column
def linear_label_rul(data):
    data['RUL'] = 0

    ind = data.index.get_level_values(0)
    units = len(ind.unique())
    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()
        data.loc[unit, 'RUL'] = range(unit_len, 0, -1)

    return data

# Adds a non-linear labeled RUL column
def label_rul(data):
    data['RUL'] = 0
    min_run_length = get_min_run_length(data)

    ind = data.index.get_level_values(0)
    units = len(ind.unique())
    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()

        data.loc[unit, 'RUL'] = range(unit_len, 0, -1)

        const = unit_len - (min_run_length - 50)
        data.loc[unit, 'RUL'].loc[0:const] = (min_run_length - 50)

    return data

# Calculates the moving average of the predictions
def moving_average(predicted_rul, window_size):
    ma_predicted_rul = pd.DataFrame(data=predicted_rul, index=y_test.index)

    for j in range(len(testing_units)):
        ra_engine = ma_predicted_rul.loc[testing_units[j]].rolling(window_size).mean()

        ##### Sets NaN values to the same as in predicted array #####
        for i in range(window_size - 1):
            ra_engine.loc[i + 1] = ma_predicted_rul.loc[testing_units[j]].loc[i + 1]

        ma_predicted_rul.loc[testing_units[j]] = ra_engine.values

    return ma_predicted_rul

# Corrects the moving average filtered predictions by subtracting half the window size from each prediction
def corr_moving_average(predicted_rul, window_size):
    corr_ma = pd.DataFrame(data=predicted_rul, index=y_test.index)

    for i in range(len(corr_ma.values)):
        corr_ma.values[i] = corr_ma.values[i] - (window_size / 2)

    return corr_ma


if __name__ == '__main__':
    ###################################### Load Training Data ############################################################
    data = pd.read_csv('train.txt', sep=" ", header=None, index_col=[0, 1],
                       names=["operational_setting 1",
                              "operational_setting 2", "operational_setting 3", "sensor 1", "sensor 2", "sensor 3",
                              "sensor 4", "sensor 5", "sensor 6", "sensor 7", "sensor 8", "sensor 9", "sensor 10",
                              "sensor 11", "sensor 12", "sensor 13", "sensor 14", "sensor 15", "sensor 16",
                              "sensor 17", "sensor 18", "sensor 19", "sensor 20", "sensor 21", "", ""])
    data = data[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]]

    # Features below can be tested in different combinations

    # Clusters data into 6 clusters and add it to dataset as operational mode
    kmeans = KMeans(n_clusters=6).fit(data[[0,1,2]])
    operational_mode = kmeans.predict(data[[0,1,2]])
    data['Operational Mode']=operational_mode

    # Drops 'unnecessary' sensors
    data = data.drop(['sensor 1','sensor 2', 'sensor 5','sensor 6','sensor 8','sensor 10', 'sensor 13','sensor 14',
                      'sensor 15' ,'sensor 16','sensor 17', 'sensor 18', 'sensor 19','sensor 20', 'sensor 21'],
                      axis=1)

    # Adds mode cycles to data
```

```
data = add_mode_cycles(data)

# Normalise data by mode
data = normalize_by_mode(data, zero2one=True)

# Normalise data from 0 to 1
#data = normalize(data, zero2one=True)

# Calculates the rolling difference on sensor 3 and 11
data['sensor 11 Diff'] = data['sensor 11'].diff(periods=4)
data['sensor 3 Diff'] = data['sensor 3'].diff(periods=4)

# Calculates the rolling mean on sensor 3 and 11
data['sensor 11 Mean'] = data['sensor 11'].rolling(4).mean()
data['sensor 3 Mean'] = data['sensor 3'].rolling(4).mean()

# Fill missing data with 0
data = data.fillna(0)

# Labels the dataset with linear RUL to be able to compare slices later
norm_data = linear_label_rul(data)

# Removes 'Operational Mode' from dataset
norm_data = norm_data.drop(['Operational Mode'], axis=1)

# Saves the dataset as .csv file
data.to_csv('DatasetForAlgos.csv')

# Reading .csv fil can be used after first run. Then the code above can be commented
norm_data = pd.read_csv('DatasetForAlgos.csv', sep=',', header=0, index_col=[0, 1])

# Drops 'Operational Mode' and Mode 1, 2 and 6 from dataset
# norm_data = norm_data.drop(['Operational Mode', 'Mode 1', 'Mode 2', 'Mode 6'], axis=1)

# Makes a copy of the normalised dataset and add RUL column with non-linear RUL
labeled_data = norm_data.copy(deep=True)
del labeled_data['RUL']
labeled_data = label_rul(labeled_data)

# Separate Train and Test
random.seed(a=5)
rnd_vector=random.sample(range(1, 219), 218)
training_units=rnd_vector[0:145]
testing_units=rnd_vector[145::]

# Splits the two datasets into training and test sets
test_set_norm = norm_data[norm_data.columns[3::]].loc[testing_units].copy()
train_set_norm = norm_data[norm_data.columns[3::]].loc[training_units].copy()
test_set_labeled = labeled_data[labeled_data.columns[3::]].loc[testing_units].copy()
train_set_labeled = labeled_data[labeled_data.columns[3::]].loc[training_units].copy()

# Further split the train and test dataset to not contain the RUL
x_train = train_set_norm[train_set_norm.columns[0:-1]]
x_test = test_set_norm[test_set_norm.columns[0:-1]]
x_train_labeled = train_set_labeled[train_set_labeled.columns[0:-1]]
x_test_labeled = test_set_labeled[test_set_labeled.columns[0:-1]]

# Split train set to target values (RUL)
y_train = train_set_norm.RUL
y_test = test_set_norm.RUL
y_train_labeled = train_set_labeled.RUL
y_test_labeled = test_set_labeled.RUL

##################################### Machine learning ############################################################
mlp = MLPRegressor(solver='lbfgs', alpha=1e-7, hidden_layer_sizes=(30), activation='relu', random_state=None,
                   momentum=0.9)
gbr = sklearn.ensemble.GradientBoostingRegressor(n_estimators=100, max_depth=8)

# List of models that will be trained
models = [mlp, gbr]
model_names = ['MLP', 'Gradient Boosting Regressor']

# Dataframe for predicted results (Preres = predicted results)
preres = pd.DataFrame(y_test.values, index=y_test.index, columns=['Actual RUL'])

# Dataframe for predicted results filtered with moving average (Mares = moving average predicted results)
mares = pd.DataFrame(y_test.values, index=y_test.index, columns=['Actual RUL'])

# Dataframe for predicted results with non-linear RUL (Non-linear RUL predicted results)
preres_labeled = pd.DataFrame(y_test_labeled.values, index=y_test_labeled.index, columns=['Actual RUL'])

# Dataframe for predicted results filtered with moving average and non-linear RUL (Non-linear RUL filtered results)
mares_labeled = pd.DataFrame(y_test_labeled.values, index=y_test_labeled.index, columns=['Actual RUL'])
```

```
# Different window sizes that will be tested on the moving average filter
windows_sizes = [2, 3, 4, 7, 10]

# Training and prediction of machine learning models
for i in range(len(models)):
    start = time.clock()
    # Fit the ML model to the linear labeled train set
    models[i].fit(x_train, y_train)
    train_time = format((time.clock() - start), '.2f')

    start = time.clock()
    # Predicts with the current ML model
    linear_prediction = models[i].predict(x_test)

    # Prints MSE, training time and prediction time
    print('MODEL ' + str(i + 1) + ' - ' + model_names[i] + '|| MSE :' +
          str(mean_squared_error(y_test, linear_prediction)) + 'Training time: ' + str(train_time) + ' sec')
    pred_time = format((time.clock() - start), '.2f')
    print(' Prediction time: ' + str(pred_time) + '\n')

    # Adds the predicted results with the current ML model
    predicted_x_test = linear_prediction
    preres[model_names[i]] = predicted_x_test

    start = time.clock()
    # Fit the ML model to the non-linear labeled train set
    models[i].fit(x_train_labeled, y_train_labeled)
    train_time = format((time.clock() - start), '.2f')

    start = time.clock()
    # Predicts with the current ML model
    non_linear_prediction = models[i].predict(x_test_labeled)

    # Prints MSE, training time and prediction time
    print('LABELED MODEL ' + str(i + 1) + ' - ' + model_names[i] + '|| MSE :' +
          str(mean_squared_error(y_test_labeled, non_linear_prediction)) +
          'Training time: ' + str(train_time) + ' sec')
    pred_time = format((time.clock() - start), '.2f')
    print(' Prediction time: ' + str(pred_time) + '\n')

    # Adds the predicted results with the current ML model
    predicted_x_test_labeled = non_linear_prediction
    preres_labeled[model_names[i]] = predicted_x_test_labeled

    # Moving average filter with different window sizes on each ML predictions
    for j in range(len(windows_sizes)):
        win_size = windows_sizes[j]

        start = time.clock()
        # Filters the predicted result with the current window size
        ma_x_test = moving_average(predicted_x_test, win_size)

        # Adds the filtered predictions with the current window size
        mares[str(win_size) + model_names[i]] = ma_x_test.loc[:, 0].values

        # Filters the predicted result with the current window size using a corrected filter
        corr_ma_x_test = corr_moving_average(ma_x_test, win_size)

        # Adds the corrected filtered predictions with the current window size
        mares[str(win_size) + model_names[i] + 'corr'] = corr_ma_x_test.loc[:, 0].values
        ma_time = (time.clock() - start)

        # Prints MSE and filtering time for both filters
        print(str(win_size) + ' MA Filtered Model: ' + model_names[i] + ' Mean Square error: ' +
              str(mean_squared_error(y_test, mares[str(win_size) + model_names[i]])) + ' Filtering time: ' +
              str(ma_time))
        print('Corrected ' + str(win_size) + ' MA Filtered Model: ' + model_names[i] + ' Mean Square error: ' +
            str(mean_squared_error(y_test, mares[str(win_size) + model_names[i] + 'corr'])) + ' Filtering time: ' +
            str(ma_time) + '\n')

        start = time.clock()
        # Filters the predicted result with non-linear RUL with the current window size
        ma_x_test_labeled = moving_average(predicted_x_test_labeled, win_size)

        # Adds the filtered predictions with the current window size
        mares_labeled[str(win_size) + model_names[i]] = ma_x_test_labeled.loc[:, 0].values

        # Filters the predicted result with non-linear RUL with the current window size using a corrected filter
        corr_ma_x_test_labeled = corr_moving_average(ma_x_test_labeled, win_size)

        # Adds the corrected filtered predictions with the current window size
        mares_labeled[str(win_size) + model_names[i] + 'corr'] = corr_ma_x_test_labeled.loc[:, 0].values
        ma_time = (time.clock() - start)
```

```
        # Prints MSE and filtering time for both filters
        print('Labeled ' + str(win_size) + ' MA Filtered Model: ' + model_names[i] + ' Mean Square error: ' +
              str(mean_squared_error(y_test_labeled, mares_labeled[str(win_size) + model_names[i]])) +
              ' Filtering time: ' + str(ma_time))
        print('Labeled ' + 'Corrected ' + str(win_size) + ' MA Filtered Model: ' + model_names[i] +
              ' Mean Square error: ' +
              str(mean_squared_error(y_test_labeled, mares_labeled[str(win_size) + model_names[i] + 'corr'])) +
              ' Filtering time: ' + str(ma_time) + '\n')

# Finds the minimum run length and subtract a constant value and 'forces' the RUL to keep this value until the RUL
# is less than this value (non-linear RUL)
corr_size = get_min_run_length(norm_data) - 50

# Slice all necessary data from the point where the non-linear RUL starts
pre_result = preres.loc[preres['Actual RUL'] < corr_size]
pre_result_labeled = preres_labeled.loc[preres_labeled['Actual RUL'] < corr_size]
ma_result = mares.loc[mares['Actual RUL'] < corr_size]
ma_result_labeled = mares_labeled.loc[mares_labeled['Actual RUL'] < corr_size]
y_test_corr = y_test[y_test.values < corr_size]
y_test_labeled_corr = y_test_labeled[y_test.values < corr_size]

# Print the comparisons for the predicted results (linear and non-linear RUL)
print('SLICED MODELS COMPARISON')
for i in range(len(models)):
    print('Model: ' + model_names[i] + ' Mean Square error: ' +
          str(mean_squared_error(y_test_corr, pre_result[model_names[i]])))
    print('Labeled ' + ' Model: ' + model_names[i] + ' Mean Square error: ' +
          str(mean_squared_error(y_test_labeled_corr, pre_result_labeled[model_names[i]])) + '\n')

# Print the comparisons for the filtered predicted results
print('FILTERED SLICED MODELS COMPARISON:')
for i in range(len(models)):
    for j in range(len(windows_sizes)):
        win_size = windows_sizes[j]

        print(str(win_size) + ' MA Filtered Model: ' + model_names[i] + ' Mean Square error: ' +
              str(mean_squared_error(y_test_corr, ma_result[str(win_size) + model_names[i]])))
        print('Labeled ' + str(win_size) + ' MA Filtered Model: ' + model_names[i] + ' Mean Square error: ' +
              str(mean_squared_error(y_test_labeled_corr, ma_result_labeled[str(win_size) + model_names[i]])))

        # Models that have been corrected after filter
        print('Corrected ' + str(win_size) + ' MA Filtered Model: ' + model_names[i] + ' Mean Square error: ' +
              str(mean_squared_error(y_test_corr, ma_result[str(win_size) + model_names[i] + 'corr'])))
        print('Labeled ' + 'Corrected ' + str(win_size) + ' MA Filtered Model: ' + model_names[i] +
              ' Mean Square error: ' +
              str(mean_squared_error(y_test_labeled_corr, ma_result_labeled[str(win_size) +
                                                                            model_names[i] + 'corr'])) + '\n')

# Visualises MLP and GBR predictions
fig = plt.figure(1)
plt.suptitle('MLP and GBR predictions')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['Gradient Boosting Regressor'], 'green')
    plt.plot(y_test.loc[j].index, preres.loc[j]['MLP'], 'blue')
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'GBR Predicted', 'MLP Predicted'], loc='lower left')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

# Visualises GBR predictions, filtered predictions and corrected filtered predictions
fig = plt.figure(2)
plt.suptitle('Gradient Boosting Regressor - Filter window size = 4')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['Gradient Boosting Regressor'], 'green')
    plt.plot(y_test.loc[j].index, mares.loc[j]['4Gradient Boosting Regressor'], 'blue')
    plt.plot(y_test.loc[j].index, mares.loc[j]['4Gradient Boosting Regressorcorr'], 'red')
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'GBR Predicted', 'MA Filtered', 'Corrected MA'], loc='lower left')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

# Visualises MLP predictions, filtered predictions and corrected filtered predictions
fig = plt.figure(3)
plt.suptitle('MLP Prediction - Filter window size = 3')
```

```
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['MLP'], 'green')
    plt.plot(y_test.loc[j].index, mares.loc[j]['3MLP'], 'blue')
    plt.plot(y_test.loc[j].index, mares.loc[j]['3MLPcorr'], 'red')
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'MLP Predicted', 'MA Filtered', 'Corrected MA'], loc='lower left')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

# Visualises non-linear RUL GBR predictions, filtered predictions and corrected filtered predictions
fig = plt.figure(4)
plt.suptitle('Labeled Gradient Boosting Regressor - Filter window size = 4')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test_labeled.loc[j].index, preres_labeled.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test_labeled.loc[j].index, preres_labeled.loc[j]['Gradient Boosting Regressor'], 'green')
    plt.plot(y_test_labeled.loc[j].index, mares_labeled.loc[j]['4Gradient Boosting Regressor'], 'blue')
    plt.plot(y_test_labeled.loc[j].index, mares_labeled.loc[j]['4Gradient Boosting Regressorcorr'], 'red')
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'GBR Predicted', 'MA Filtered', 'Corrected MA'], loc='lower left')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

# Visualises non-linear RUL MLP predictions, filtered predictions and corrected filtered predictions
fig = plt.figure(5)
plt.suptitle('Labeled MLP Prediction - Filter window size = 3')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test_labeled.loc[j].index, preres_labeled.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test_labeled.loc[j].index, preres_labeled.loc[j]['MLP'], 'green')
    plt.plot(y_test_labeled.loc[j].index, mares_labeled.loc[j]['3MLP'], 'blue')
    plt.plot(y_test_labeled.loc[j].index, mares_labeled.loc[j]['3MLPcorr'], 'red')
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'MLP Predicted', 'MA Filtered', 'Corrected MA'], loc='lower left')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

# Visualises the comparison of linear and non-linear predictions after data have been sliced
fig = plt.figure(6)
plt.suptitle('Comparison of linear and non-linear models after slicing')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test_labeled_corr.loc[j].index, pre_result_labeled.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test_corr.loc[j].index, pre_result.loc[j]['Gradient Boosting Regressor'][-corr_size::], 'green',
             alpha=1)
    plt.plot(y_test_corr.loc[j].index, pre_result.loc[j]['MLP'][-corr_size::], 'goldenrod', alpha=0.875)
    plt.plot(y_test_labeled_corr.loc[j].index, pre_result_labeled.loc[j]['Gradient Boosting Regressor'], 'blue',
             alpha=0.675)
    plt.plot(y_test_labeled_corr.loc[j].index, pre_result_labeled.loc[j]['MLP'], 'red', alpha=0.75)
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'GBR Predicted', 'MLP Predicted', 'Re-Labeled GBR Predicted',
                'Re-Labeled MLP Predicted'], loc='lower left', fontsize='small')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

# Visualises the comparison of filtered linear and non-linear predictions after data have been sliced
fig = plt.figure(7)
plt.suptitle('Comparison of filtered linear and non-linear models after slicing')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test_labeled_corr.loc[j].index, ma_result_labeled.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test_corr.loc[j].index, ma_result.loc[j]['4Gradient Boosting Regressor'][-corr_size::], 'green',
             alpha=1)
    plt.plot(y_test_corr.loc[j].index, ma_result.loc[j]['3MLP'][-corr_size::], 'goldenrod', alpha=0.875)
    plt.plot(y_test_labeled_corr.loc[j].index, ma_result_labeled.loc[j]['4Gradient Boosting Regressor'], 'blue',
             alpha=0.675)
    plt.plot(y_test_labeled_corr.loc[j].index, ma_result_labeled.loc[j]['3MLP'], 'red', alpha=0.75)
    plt.xlabel('Cycles')
```

```
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'GBR Predicted', 'MLP Predicted', 'Re-Labeled GBR Predicted',
                'Re-Labeled MLP Predicted'], loc='lower left', fontsize='small')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

# Visualises the comparison of linear and non-linear predictions on short life engines after data have been sliced
fig = plt.figure(8)
plt.suptitle('Comparison of filtered linear and non-linear models after slicing on short life engines')
subnum = 231
eng = [218, 67, 205, 58, 198, 55]
for i in range(len(eng)):
    plt.subplot(subnum)
    j = eng[i]
    plt.plot(y_test_labeled_corr.loc[j].index, ma_result_labeled.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test_corr.loc[j].index, ma_result.loc[j]['4Gradient Boosting Regressor'][-corr_size::], 'green',
             alpha=1)
    plt.plot(y_test_corr.loc[j].index, ma_result.loc[j]['3MLP'][-corr_size::], 'goldenrod', alpha=0.875)
    plt.plot(y_test_labeled_corr.loc[j].index, ma_result_labeled.loc[j]['4Gradient Boosting Regressor'], 'blue',
             alpha=0.675)
    plt.plot(y_test_labeled_corr.loc[j].index, ma_result_labeled.loc[j]['3MLP'], 'red', alpha=0.75)
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'GBR Predicted', 'MLP Predicted', 'Re-Labeled GBR Predicted',
                'Re-Labeled MLP Predicted'],
               loc='lower left', fontsize='small')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1

plt.show()
```

Experiment D - Classification.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
import time

from sklearn.cluster import KMeans
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble.gradient_boosting import GradientBoostingClassifier
from sklearn.metrics import precision_score
from sklearn.metrics import accuracy_score

# Adds columns for time (nr. of cycles) spent in each of the 6 modes
def add_mode_cycles(data):
    data['Mode 1'] = 0; data['Mode 2'] = 0; data['Mode 3'] = 0
    data['Mode 4'] = 0; data['Mode 5'] = 0; data['Mode 6'] = 0

    ind = data.index.get_level_values(0)
    units = len(ind.unique())
    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()
        column = np.zeros((unit_len, 6), int)

        for index, row in data.loc[unit].iterrows():
            mode = int(row['Operational Mode'])
            if index != 1:
                column[index - 1] = column[index - 2]
            column[index - 1][mode] = column[index - 1][mode] + 1

        data.loc[unit].loc[:, data.columns[-6::]] = column

    return data

# Transforms the data from one range to another (0 to 1 or based on mean and standard deviation in each feature)
def normalize(df, zero2one=False):
    result = df.copy()
    for feature_name in df.columns:
        if zero2one == True:
            min = df[feature_name].min()
            max = df[feature_name].max()
            result[feature_name] = (df[feature_name] - min) / (max - min)
        else:
            mean = df[feature_name].mean()
            std = df[feature_name].std()
            result[feature_name] = (df[feature_name] - mean) / (std)

    return result

# Extracts all values in each operational mode, for every column, and normalise based on those
def normalize_by_mode(df, zero2one=False):
    result = df.copy()

    for i in range(6):
        mean = (df.loc[df['Operational Mode'] == i]).mean()
        std = (df.loc[df['Operational Mode'] == i]).std()
        if(zero2one == True):
            mean = (df.loc[df['Operational Mode'] == i]).min()
            std = (df.loc[df['Operational Mode'] == i]).max() - (df.loc[df['Operational Mode'] == i]).min()

        for feature_name in df.columns[0:-1]:
            result[feature_name].loc[result['Operational Mode'] == i] = \
                (df[feature_name].loc[df['Operational Mode'] == i] - mean[feature_name])/(std[feature_name])

    result = result.fillna(0)
    result[result.columns[-7::]] = normalize(result[result.columns[-7::]], zero2one=zero2one)
    result = result.fillna(0)

    return result

# Adds a linear labeled RUL column
def linear_label_rul(data):
    data['RUL'] = 0

    ind = data.index.get_level_values(0)
    units = len(ind.unique())
    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()
```

```
        data.loc[unit, 'RUL'] = range(unit_len, 0, -1)

    return data


if __name__ == '__main__':
    ###################################### Load Training Data ################################################################
    data = pd.read_csv('train.txt', sep=" ", header=None, index_col=[0, 1],
                       names=["operational_setting 1",
                              "operational_setting 2", "operational_setting 3", "sensor 1", "sensor 2", "sensor 3",
                              "sensor 4", "sensor 5", "sensor 6", "sensor 7", "sensor 8", "sensor 9", "sensor 10",
                              "sensor 11", "sensor 12", "sensor 13", "sensor 14", "sensor 15", "sensor 16",
                              "sensor 17", "sensor 18", "sensor 19", "sensor 20", "sensor 21", "", ""])
    data = data[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]]

    # Features below can be tested in different combinations

    # Clusters data into 6 clusters and add it to dataset as operational mode
    kmeans = KMeans(n_clusters=6).fit(data[[0,1,2]])
    operational_mode = kmeans.predict(data[[0,1,2]])
    data['Operational Mode']=operational_mode

    # Drops 'unnecessary' sensors
    data = data.drop(['sensor 1','sensor 2', 'sensor 5','sensor 6','sensor 8','sensor 10', 'sensor 13','sensor 14',
                      'sensor 15' ,'sensor 16','sensor 17', 'sensor 18', 'sensor 19','sensor 20', 'sensor 21'],
                      axis=1)

    # Adds mode cycles to data
    data = add_mode_cycles(data)

    # Normalise data by mode
    data = normalize_by_mode(data, zero2one=True)

    # Normalise data from 0 to 1
    #data = normalize(data, zero2one=True)

    # Calculates the rolling difference on sensor 3 and 11
    data['sensor 11 Diff'] = data['sensor 11'].diff(periods=4)
    data['sensor 3 Diff'] = data['sensor 3'].diff(periods=4)

    # Calculates the rolling mean on sensor 3 and 11
    data['sensor 11 Mean'] = data['sensor 11'].rolling(4).mean()
    data['sensor 3 Mean'] = data['sensor 3'].rolling(4).mean()

    # Fill missing data with 0
    data = data.fillna(0)

    # Labels the dataset with linear RUL
    norm_data = linear_label_rul(data)

    # Removes 'Operational Mode' from dataset
    norm_data = norm_data.drop(['Operational Mode'], axis=1)

    # Saves the dataset as .csv file
    data.to_csv('DatasetForClassi.csv')

    # Reading .csv fil can be used after first run. Then the code above can be commented
    #norm_data = pd.read_csv('DatasetForClassi.csv', sep=',', header=0, index_col=[0, 1])

    # Drops 'Operational Mode' and Mode 1, 2 and 6 from dataset
    # norm_data = norm_data.drop(['Operational Mode', 'Mode 1', 'Mode 2', 'Mode 6'], axis=1)

    # Separate Train and Test
    random.seed(a=5)
    rnd_vector=random.sample(range(1, 219), 218)
    training_units=rnd_vector[0:145]
    testing_units=rnd_vector[145::]

    # Splits the dataset into training and test sets
    test_set_norm = norm_data[norm_data.columns[3::]].loc[testing_units].copy()
    train_set_norm = norm_data[norm_data.columns[3::]].loc[training_units].copy()

    # Further split the train and test dataset to not contain the RUL
    x_train = train_set_norm[train_set_norm.columns[0:-1]]
    x_test = test_set_norm[test_set_norm.columns[0:-1]]

    # Split train set to target values (RUL)
    y_train = train_set_norm.RUL
    y_test = test_set_norm.RUL

    ###################################### Machine learning ##################################################################
    mlp_clf = MLPClassifier(hidden_layer_sizes=30, activation='logistic', solver='adam', alpha=0.000001)
    knc_clf = KNeighborsClassifier(n_neighbors=7, weights='uniform', algorithm='auto', leaf_size=30)
    gbc_clf = GradientBoostingClassifier(loss='deviance', learning_rate=0.25, n_estimators=170, max_depth=3)
```

```
# List of models that will be trained
models = [mlp_clf, knc_clf, gbc_clf]
model_names = ['MLP', 'KNC', 'GBC']

# Dataframe for predicted results (Preres = predicted results)
preres = pd.DataFrame(y_test.values, index=y_test.index, columns=['Actual RUL'])
#
y_test_score = pd.DataFrame(y_test.values, index=y_test.index, columns=['Actual RUL'], copy='deep')

# The fixed threshold we want to classify
const_label = 50

# Minimum number of consecutive values that classify the RUL to be less than the const_label before we say that it
# actually is less (to avoid the earliest peaks)
min_nr_of_consecutives = 2

# Sets all values that are higher than the const_level (more than const_level cycles left) to 0, while the values
# less than the const_level (less than const_level cycles left) is set to 1
y_train[y_train.values >= const_label] = 0
y_train[y_train.values > 0] = 1
y_test_score[y_test_score.values >= const_label] = 0
y_test_score[y_test_score.values > 0] = 1

for i in range(len(models)):
    start = time.clock()
    # Fit the ML model to the train set
    models[i].fit(x_train, y_train)
    train_time = format((time.clock() - start), '.2f')

    start = time.clock()
    # Predicts with the current ML model
    predicted_x_test = models[i].predict(x_test)
    preres[model_names[i]] = predicted_x_test
    pred_time = format((time.clock() - start), '.2f')

    # Prints accuracy, precision, training time and prediction time
    print('MODEL ' + str(i + 1) + ' - ' + model_names[i] + ' || Accuracy :' +
          str(accuracy_score(np.transpose(y_test_score.values)[0], predicted_x_test)) + ' || Precision: ' +
          str(precision_score(np.transpose(y_test_score.values)[0], predicted_x_test)) + ' || Training time: ' +
          str(train_time) + ' sec')
    print('Prediction time: ' + str(pred_time) + '\n')

# Visualises the MLP classification
fig = plt.figure(1)
plt.suptitle('MLP: Hidden layer sizes = 30, Activation = logistic, Solver = Adam, Alpha = 0.000001')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    ax = fig.gca()
    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['MLP'] * const_label, 'blue')
    ax.axvspan(preres.loc[j]['Actual RUL'].loc[const_label], preres.loc[j]['Actual RUL'].loc[const_label - 30],
               facecolor='navajowhite', alpha=0.5)
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'MLP'], loc='lower left')
    plt.title('Engine ' + str(j))
    fig.add_axes(ax)
    subnum = subnum + 1

# Visualises the MLP classification where minimum nr. of consecutives is considered
fig = plt.figure(2)
plt.suptitle('MLP: Hidden layer sizes = 30, Activation = logistic, Solver = Adam, Alpha = 0.000001')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    ax = fig.gca()

    nr_of_consecutives = 0
    flag = False
    for k in range(len(preres.loc[j]['MLP'])):
        if preres.loc[j]['MLP'].loc[k + 1] == 1:
            nr_of_consecutives += 1
            if (nr_of_consecutives >= min_nr_of_consecutives) and not flag:
                preres.loc[j]['MLP'][1:k] = 0
                preres.loc[j]['MLP'][k::] = 1
                flag = True
        else:
            nr_of_consecutives = 0

    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
```

```
    plt.plot(y_test.loc[j].index, preres.loc[j]['MLP'] * const_label, 'blue')
    ax.axvspan(preres.loc[j]['Actual RUL'].loc[const_label], preres.loc[j]['Actual RUL'].loc[const_label - 30],
               facecolor='navajowhite', alpha=0.5)
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'MLP'], loc='lower left')
    plt.title('Engine ' + str(j))
    fig.add_axes(ax)
    subnum = subnum + 1

# Visualises the KNC classification
fig = plt.figure(3)
plt.suptitle('KNC: Nr. of neighbors = 7, Weights = uniform, Algorithm = auto, Leaf Size = 30')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    ax = fig.gca()
    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['KNC'] * const_label, 'blue')
    ax.axvspan(preres.loc[j]['Actual RUL'].loc[const_label], preres.loc[j]['Actual RUL'].loc[const_label - 30],
               facecolor='navajowhite', alpha=0.5)
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'KNC'], loc='lower left')
    plt.title('Engine ' + str(j))
    fig.add_axes(ax)
    subnum = subnum + 1

# Visualises the KNC classification where minimum nr. of consecutives is considered
fig = plt.figure(4)
plt.suptitle('KNC: Nr. of neighbors = 7, Weights = uniform, Algorithm = auto, Leaf Size = 30')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    ax = fig.gca()

    nr_of_consecutives = 0
    flag = False
    for k in range(len(preres.loc[j]['KNC'])):
        if preres.loc[j]['KNC'].loc[k + 1] == 1:
            nr_of_consecutives += 1
            if (nr_of_consecutives >= min_nr_of_consecutives) and not flag:
                preres.loc[j]['KNC'][1:k] = 0
                preres.loc[j]['KNC'][k::] = 1
                flag = True
        else:
            nr_of_consecutives = 0

    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['KNC'] * const_label, 'blue')
    ax.axvspan(preres.loc[j]['Actual RUL'].loc[const_label], preres.loc[j]['Actual RUL'].loc[const_label - 30],
               facecolor='navajowhite', alpha=0.5)
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'KNC'], loc='lower left')
    plt.title('Engine ' + str(j))
    fig.add_axes(ax)
    subnum = subnum + 1

# Visualises the GBC classification
fig = plt.figure(5)
plt.suptitle('GBC: Loss = deviance, Learning Rate = 0.25, Nr. of estimators = 170, Max Depth = 3')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    ax = fig.gca()
    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['GBC'] * const_label, 'blue')
    ax.axvspan(preres.loc[j]['Actual RUL'].loc[const_label], preres.loc[j]['Actual RUL'].loc[const_label - 30],
               facecolor='navajowhite', alpha=0.5)
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'GBC'], loc='lower left')
    plt.title('Engine ' + str(j))
    fig.add_axes(ax)
    subnum = subnum + 1

# Visualises the GBC classification where minimum nr. of consecutives is considered
fig = plt.figure(6)
plt.suptitle('GBC: Loss = deviance, Learning Rate = 0.25, Nr. of estimators = 170, Max Depth = 3')
subnum = 231
```

```
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    ax = fig.gca()

    nr_of_consecutives = 0
    flag = False
    for k in range(len(preres.loc[j]['GBC'])):
        if preres.loc[j]['GBC'].loc[k + 1] == 1:
            nr_of_consecutives += 1
            if (nr_of_consecutives >= min_nr_of_consecutives) and not flag:
                preres.loc[j]['GBC'][1:k] = 0
                preres.loc[j]['GBC'][k-1::] = 1
                flag = True
        else:
            nr_of_consecutives = 0

    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'black')
    plt.plot(y_test.loc[j].index, preres.loc[j]['GBC'] * const_label, 'blue')
    ax.axvspan(preres.loc[j]['Actual RUL'].loc[const_label], preres.loc[j]['Actual RUL'].loc[const_label - 30],
               facecolor='navajowhite', alpha=0.5)
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'GBC'], loc='lower left')
    plt.title('Engine ' + str(j))
    fig.add_axes(ax)
    subnum = subnum + 1

plt.show()
```

```
import pandas as pd
from pandas import *
import numpy as np
import matplotlib.pyplot as plt
import sklearn
from scipy.io import loadmat
import datetime
import random

from sklearn.neighbors import KNeighborsRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import MinMaxScaler
from sklearn import linear_model
import sklearn.ensemble


#######################################################################################################################
############################## Functions for loading dataset from .MAT file ###########################################
#######################################################################################################################
''' Working with .MAT (MATLAB) files in Python is complex
    The structure of the file is complex, as it consists of many vectors inside each other,
    and each individual value is inside a vector. The upcoming figures loads the matfiles and extracts the values.
'''


# Get data from discharge cycles
def get_discharge_data(filepath):
    # Load File
    raw_data = loadmat(filepath)
    raw_data = raw_data[filepath.split('.')[0]]

    # Create pandas dataframe with the correct columns
    discharge_data = pd.DataFrame(
        columns=['Cycles','Voltage_measured', 'Current_measured', 'Temperature_measured', 'Current_load', 'Voltage_load',
                 'Time', 'Capacity'])

    # Initialise vectors for each parameter
    Voltage_measured = []; Current_measured = []; Temperature_measured = []; Current_load = [];
    Voltage_load = []; Time = []; Capacity = []; Label_Time = []; cycle_number = []

    # Extract values and add to the vectors
    cyc = 0
    for i in range(len(raw_data[0][0][0][0])):
        if raw_data[0][0][0][0][i][0][0] == 'discharge':
            cyc += 1
            cycle_number.append(cyc)
            Voltage_measured.append(raw_data[0][0][0][0][i][3][0][0][0][0])
            Current_measured.append(raw_data[0][0][0][0][i][3][0][0][1][0])
            Temperature_measured.append(raw_data[0][0][0][0][i][3][0][0][2][0])
            Current_load.append(raw_data[0][0][0][0][i][3][0][0][3][0])
            Voltage_load.append(raw_data[0][0][0][0][i][3][0][0][4][0])
            Time.append(raw_data[0][0][0][0][i][3][0][0][-2][0])
            Label_Time.append(raw_data[0][0][0][0][i][3][0][0][-2][0][::-1])
            Capacity.append(raw_data[0][0][0][0][i][3][0][0][-1][0][0])

    # Initialise new vectors for each parameter
    sykluser = []; New_Voltage_measured = []; New_Current_measured = []; New_Temperature_measured = [];
    New_Current_load = []; New_Voltage_load = []; New_Time = []; New_Label_Time = []

    # Extract values for each parameter
    for j in range(len(cycle_number)):
        test = np.ones(len(Voltage_measured[j]))*(j+1)
        for vals in test:
            sykluser.append(vals)

        for n in range(len(Voltage_measured[j])):
            New_Voltage_measured.append(Voltage_measured[j][n])
            New_Current_measured.append(Current_measured[j][n])
            New_Temperature_measured.append(Temperature_measured[j][n])
            New_Current_load.append(Current_load[j][n])
            New_Voltage_load.append(Voltage_load[j][n])
            New_Time.append(Time[j][n])
            New_Label_Time.append(Label_Time[j][n])

    # Convert lists/vectors to numpy arrays
    Cycles = np.array(sykluser)
    New_Voltage_measured = np.array(New_Voltage_measured)
    New_Current_measured = np.array(New_Current_measured)
    New_Temperature_measured = np.array(New_Temperature_measured)
    New_Current_load = np.array(New_Current_load)
    New_Voltage_load = np.array(New_Voltage_load)
    New_Time = np.array(New_Time)
    New_Label_Time = np.array(New_Label_Time)

    # Add extracted values to pandas dataframe
    discharge_data['Voltage_measured'] = New_Voltage_measured
    discharge_data['Current_measured'] = New_Current_measured
    discharge_data['Temperature_measured'] = New_Temperature_measured
    discharge_data['Current_load'] = New_Current_load
    discharge_data['Voltage_load'] = New_Voltage_load
    discharge_data['Time'] = New_Time
    discharge_data['Cycles'] = Cycles
    discharge_data['RUL'] = New_Label_Time

    # Set index
    discharge_data.set_index(['Cycles', 'Time'], inplace=True)

    # Add values for capacity
    for i in range(1,len(Capacity)+1):
        discharge_data['Capacity'].loc[i] = Capacity[i-1]
```

```
    return discharge_data, Capacity


# Get data from charge cycles
def get_charge_data(filepath):
    # Load File
    raw_data = loadmat(filepath)
    raw_data = raw_data[filepath.split('.')[0]]

    # Create pandas dataframe with the correct columns
    charge_data = pd.DataFrame(
        columns=['Cycles', 'Voltage_measured', 'Current_measured', 'Temperature_measured', 'Current_charge',
                 'Voltage_charge',
                 'Time'])

    # Initialise vectors for each parameter
    Voltage_measured = []; Current_measured = []; Temperature_measured = []; Current_charge = [];
    Voltage_charge = []; Time = []; Label_Time = []; cycle_number = [];

    # Extract values and add to the vectors
    cyc = 0
    for i in range(len(raw_data[0][0][0][0])):
        if raw_data[0][0][0][0][i][0][0] == 'charge':
            cyc += 1
            cycle_number.append(cyc)
            Voltage_measured.append(raw_data[0][0][0][0][i][3][0][0][0][0])
            Current_measured.append(raw_data[0][0][0][0][i][3][0][0][1][0])
            Temperature_measured.append(raw_data[0][0][0][0][i][3][0][0][2][0])
            Current_charge.append(raw_data[0][0][0][0][i][3][0][0][3][0])
            Voltage_charge.append(raw_data[0][0][0][0][i][3][0][0][4][0])
            Time.append(raw_data[0][0][0][0][i][3][0][0][-1][0])
            Label_Time.append(raw_data[0][0][0][0][i][3][0][0][-1][0][::-1])

    # Initialise new vectors for each parameter
    sykluser = []; New_Voltage_measured = []; New_Current_measured = []; New_Temperature_measured = []
    New_Current_charge = []; New_Voltage_charge = []; New_Time = []; New_Label_Time = []

    # Extract values for each parameter
    for j in range(len(cycle_number)):
        test = np.ones(len(Voltage_measured[j])) * (j + 1)
        for vals in test:
            sykluser.append(vals)

        for n in range(len(Voltage_measured[j])):
            New_Voltage_measured.append(Voltage_measured[j][n])
            New_Current_measured.append(Current_measured[j][n])
            New_Temperature_measured.append(Temperature_measured[j][n])
            New_Current_charge.append(Current_charge[j][n])
            New_Voltage_charge.append(Voltage_charge[j][n])
            New_Time.append(Time[j][n])
            New_Label_Time.append(Label_Time[j][n])

    # Convert lists/vectors to numpy arrays
    Cycles = np.array(sykluser)
    New_Voltage_measured = np.array(New_Voltage_measured)
    New_Current_measured = np.array(New_Current_measured)
    New_Temperature_measured = np.array(New_Temperature_measured)
    New_Current_charge = np.array(New_Current_charge)
    New_Voltage_charge = np.array(New_Voltage_charge)
    New_Time = np.array(New_Time)
    New_Label_Time = np.array(New_Label_Time)

    # Add extracted values to pandas dataframe
    charge_data['Voltage_measured'] = New_Voltage_measured
    charge_data['Current_measured'] = New_Current_measured
    charge_data['Temperature_measured'] = New_Temperature_measured
    charge_data['Current_charge'] = New_Current_charge
    charge_data['Voltage_charge'] = New_Voltage_charge
    charge_data['Time'] = New_Time
    charge_data['Cycles'] = Cycles
    charge_data['RUL'] = New_Label_Time

    # Set index
    charge_data.set_index(['Cycles', 'Time'], inplace=True)

    return charge_data


# Get data from impedance cycles
def get_impedance_data(filepath):
    # Load file
    raw_data = loadmat(filepath)
    raw_data = raw_data[filepath.split('.')[0]]

    # Create pandas dataframe with the correct columns
    imp_data = pd.DataFrame(
        columns=['Cycles','Time', 'Sense_current', 'Battery_current', 'Current_ratio', 'Re', 'Rct'])

    # Initialise vectors for each parameter
    Sense_current = []; Battery_current = []; Current_ratio = []; Time = []; Re = []; Rct = []; cycle_number = []

    # Extract values and add to the vectors
    cyc = 0
    for i in range(len(raw_data[0][0][0][0])):
        if raw_data[0][0][0][0][i][0][0] == 'impedance':
            cyc += 1
            cycle_number.append(cyc)
            Sense_current.append(raw_data[0][0][0][0][i][3][0][0][0][0])
            Battery_current.append(raw_data[0][0][0][0][i][3][0][0][1][0])
            Current_ratio.append(raw_data[0][0][0][0][i][3][0][0][2][0])
            Time.append(range(1,len(raw_data[0][0][0][0][i][3][0][0][0][0])+1))
            #Battery_impedance.append(raw_data[0][0][0][0][i][3][0][0][3][0])
            #Rectified_impedance.append(raw_data[0][0][0][0][i][3][0][0][4][0])
            Re.append(raw_data[0][0][0][0][i][3][0][0][-2][0][0])
```

```
            Rct.append(raw_data[0][0][0][0][i][3][0][0][-1][0][0])

    # Initialise new vectors for each parameter
    sykluser = []; New_Sense_current = []; New_Battery_current = []; New_Current_ratio = []
    New_Time = []; New_Battery_impedance = []

    # Extract values for each parameter
    for j in range(len(cycle_number)):
        test = np.ones(len(Sense_current[j])) * (j + 1)
        for vals in test:
            sykluser.append(vals)

        for n in range(len(Sense_current[j])):
            New_Sense_current.append(Sense_current[j][n])
            New_Battery_current.append(Battery_current[j][n])
            New_Current_ratio.append(Current_ratio[j][n])
            New_Time.append(Time[j][n])

    # Convert lists/vectors to numpy arrays
    Cycles = np.array(sykluser)
    New_Sense_current = np.array(New_Sense_current)
    New_Battery_current = np.array(New_Battery_current)
    New_Current_ratio = np.array(New_Current_ratio)
    New_Time = np.array(New_Time)

    # Add extracted values to pandas dataframe
    imp_data['Sense_current'] = New_Sense_current
    imp_data['Battery_current'] = New_Battery_current
    imp_data['Current_ratio'] = New_Current_ratio
    imp_data['Cycles'] = Cycles
    imp_data['Time'] = New_Time

    # Set index
    imp_data.set_index(['Cycles', 'Time'], inplace=True)

    # Add values for capacity
    for i in range(1, len(Re) + 1):
        imp_data['Re'].loc[i] = Re[i - 1]
        imp_data['Rct'].loc[i] = Rct[i - 1]

    return imp_data, Re, Rct

##########################################################################################################################
########################################### Extract features ##############################################################
##########################################################################################################################


# Extract features from charge and discharge cycles. Extracts features that amplify battery life degradation.
def extract_features(data, mode):
    features = []

    # Extract features from charge cycles
    if mode == 'Charge':
        print('Extracting features from Charge cycles')

        volt_mea_time_ref = data['Voltage_measured'].loc[2].loc[data['Voltage_measured'].loc[2] > 4.2].index.min()
        volt_mea_time_change = []
        current_mea_drop_ref = (data['Current_measured'].loc[2].loc[data['Current_measured'].loc[2] > 0.2].index.max() -
                                data['Current_measured'].loc[2].loc[data['Current_measured'].loc[2] > 1.5].index.max())
        current_mea_drop_change = []
        temp_peak_ref = data['Temperature_measured'].loc[2].loc[data.loc[2].index>500].idxmax()
        temp_peak_change = []

        for i in range(1, len(data.index.levels[0]) + 1):
            volt_mea_time_change.append(volt_mea_time_ref -
                                        data['Voltage_measured'].loc[i].loc[data['Voltage_measured'].loc[i]
                                                                            > 4.2].index.min())
            temp_cur_drop = (data['Current_measured'].loc[i].loc[data['Current_measured'].loc[i] > 0.2].index.max() -
                             data['Current_measured'].loc[i].loc[data['Current_measured'].loc[i] > 1.5].index.max())
            current_mea_drop_change.append(temp_cur_drop - current_mea_drop_ref)
            if i != 170:
                temp_peak_change.append(temp_peak_ref - data['Temperature_measured'].loc[i].loc[data.loc[i].index>200].idxmax())
            else:
                temp_peak_change.append(temp_peak_change[-1])
        features.append(np.array(volt_mea_time_change))
        features.append(np.array(current_mea_drop_change))
        features.append(np.array(temp_peak_change))


    # Extract features from discharge cycles
    if mode == 'Discharge':
        print('Extracting features from Discharge cycles')

        volt_mea_discharge_ref = data['Voltage_measured'].loc[1].loc[data.loc[1].index.max()]
        volt_mea_discharge_change = []
        cycle_discharge_ref = data.loc[1].index.max()
        cycle_discharge_change = []
        temp_discharge_ref = data['Temperature_measured'].loc[1].max()
        temp_discharge_change = []
        volt_load_ref = data['Voltage_load'].loc[1].iloc[round(len(data.loc[1].index)/1.8)]
        volt_load_change = []
        for i in range(1, len(data.index.levels[0])+1):
            volt_mea_discharge_change.append(data['Voltage_measured'].loc[i].loc[data.loc[i].index.max()] - volt_mea_discharge_ref) # Max sluttverdi
            cycle_discharge_change.append(cycle_discharge_ref-data.loc[i].index.max())
            temp_discharge_change.append(data['Temperature_measured'].loc[i].max()-temp_discharge_ref)
            volt_load_change.append(volt_load_ref-data['Voltage_load'].loc[i].iloc[round(len(data.loc[i].index)/1.8)])
        features.append(np.array(volt_mea_discharge_change))
        features.append(np.array(cycle_discharge_change))
        features.append(np.array(temp_discharge_change))
        features.append(np.array(volt_load_change))

    return features
```

```
# Generate dataset with available features
def generate_EOL_dataset(disch_data, charge_data, capacity, re, rct):
    re = np.array(re)
    rct = np.array(rct)
    tmp = np.linspace(1,len(re)-1,len(disch_data[0]))
    tmp = tmp.astype(int)

    re = re[tmp]
    rct = rct[tmp]

    cols = ['capacity', 'Re', 'Rc']
    df = pd.DataFrame(columns=cols, index=range(1, len(disch_data[0])+1))
    df['volt_mea_discharge_change'] = disch_data[0]
    df['cycle_discharge_change'] = disch_data[1]
    df['temp_discharge_change'] = disch_data[2]
    #df['volt_load_change'] = disch_data[3]
    df['capacity'] = capacity
    #df['volt_mea_time_change'] = charge_data[0][1:-1]
    df['current_mea_drop_change'] = charge_data[1][1:-1]
    df['temp_peak_change'] = charge_data[2][1:-1]
    df['Re'] = re
    df['Rct'] = rct

    df.fillna(0, inplace=True)

    return df


# Convert datetime array format to datetime format
def convert_array_to_datetime(timeArray):
    timestamps = []
    for i in range(len(timeArray)):
        year = timeArray[i][0]
        month = timeArray[i][1]
        day = timeArray[i][2]
        hour = timeArray[i][3]
        minute = timeArray[i][4]
        second = timeArray[i][5]

        timestamp = datetime.datetime(int(year), int(month), int(day), int(hour), int(minute), int(second))
        timestamps.append(timestamp)
    return timestamps

# Lable dataset - Convert from charge and discharge cycles until combined cycles.
def label(batteries):
    rul = []
    for i in range(len(batteries)):
        rul.append(range(len(batteries[i].index),0,-1))

    return rul


# Moving average filtering for predictions. Returns filtered predictions
def moving_average(predicted_rul, window_size):
    ma_predicted_rul = pd.DataFrame(data=predicted_rul, index=range(1,len(predicted_rul)+1))
    ma_predicted_rul2 = ma_predicted_rul.rolling(window_size).mean().copy()
    ma_predicted_rul2.loc[0:window_size]=ma_predicted_rul.loc[0:window_size]

    return ma_predicted_rul2


# Corrected moving average function for predictions. Returns filtered predictions
def corr_moving_average(predicted_rul, window_size):
    corr_ma2 = predicted_rul - (window_size-window_size/2)

    return corr_ma2



if __name__ == '__main__':
    # Battery dataset can be downloaded from:
    #                  https://ti.arc.nasa.gov/tech/dash/pcoe/prognostic-data-repository/

    # Defines the paths to the battery files.
    files = ['B0005.mat', 'B0006.mat', 'B0018.mat', 'B0007.mat']

    # Initialise list for battery data
    Batteries = []

    # Load and generate dataset for each battery file
    for f in files:
        # Print current file
        print(str(f))

        # Get data from discharge cycles
        disch_data, Capacity = get_discharge_data(f)

        # Get data from charge cycles
        charge_data = get_charge_data(f)

        # Get data from impedance cycles
        imp_data, Re, Rct = get_impedance_data(f)

        # Extract features from charge and discharge cycles
        disch_features = extract_features(disch_data, 'Discharge')
        charge_features = extract_features(charge_data, 'Charge')

        # Generate dataset and add to battery list
        Batteries.append(generate_EOL_dataset(disch_features, charge_features, Capacity, Re, Rct))

    disch_data, Capacity = get_discharge_data(files[0])
    charge_data = get_charge_data(files[0])
    imp_data, Re, Rct = get_impedance_data(files[0])
```

```
# Create scaler for normalisation
scaler = MinMaxScaler()

# Create list for normalised data
Norm_Batteries = []

# Normalise data and add to list. Scaler is fit with first battery,
# the additional batteries are transformed with the same scaler.
Norm_Batteries.append(pd.DataFrame(data = scaler.fit_transform(Batteries[0]),
                                   columns=Batteries[0].columns, index=Batteries[0].index))
Norm_Batteries.append(pd.DataFrame(data = scaler.transform(Batteries[1]),
                                   columns=Batteries[1].columns, index=Batteries[1].index))
Norm_Batteries.append(pd.DataFrame(data = scaler.transform(Batteries[2]),
                                   columns=Batteries[2].columns, index=Batteries[2].index))
Norm_Batteries.append(pd.DataFrame(data=scaler.transform(Batteries[3]),
                                   columns=Batteries[3].columns, index=Batteries[3].index))

# Lable the dataset with remaining life
RUL = label(Norm_Batteries)

# Choose which batteries to use for training and testing
train_bat1 = 0; train_bat2 = 3
test_bat1 = 1; test_bat2 = 2

# Create test dataset
test = pd.concat([Norm_Batteries[train_bat1],Norm_Batteries[train_bat2]])

# Create target to test dataset
test_Rul = np.concatenate((RUL[train_bat1],RUL[train_bat2]), axis=0)

#########################################################################################################################
#                                       Machine learning
#########################################################################################################################

# MLP Regressor - Create and fit
mlp = MLPRegressor(hidden_layer_sizes=15, warm_start=False, activation='identity', solver='lbfgs', momentum=0.95)
mlp.fit(test, test_Rul)

# Gradient boosting regressor - Create and fit
gbr = sklearn.ensemble.GradientBoostingRegressor(n_estimators=20, max_depth=15, warm_start=False, loss='ls')
gbr.fit(test, test_Rul)

# Predict on test batteries with both MLP and Gradient Boosting Regressor
predicted7 = mlp.predict(Norm_Batteries[test_bat1])
predicted18 = mlp.predict(Norm_Batteries[test_bat2])
predicted7B = gbr.predict(Norm_Batteries[test_bat1])
predicted18B = gbr.predict(Norm_Batteries[test_bat2])

print('------------------------------------------------------------' + '\n')
print('MLP on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(predicted7, RUL[test_bat1])))
print('GBR on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(predicted7B, RUL[test_bat1])))
print('MLP on B0018 - MSE: ' + str(sklearn.metrics.mean_squared_error(predicted18, RUL[test_bat2])))
print('GBR on B0018 - MSE: ' + str(sklearn.metrics.mean_squared_error(predicted18B, RUL[test_bat2])))

predMLP = [predicted7] # List for predictions with MLP on test battery 1
predGBR = [predicted7B] # List for predictions with GBR on test battery 1

# Filter predictions with moving average and corrected moving average with different window sizes
for i in [2,3,4,7,10]:
    # Predictions filtered with moving average
    ma_predA = moving_average(predicted7,i)
    ma_predB = moving_average(predicted7B, i)

    # Predictions filtered with corrected moving average
    corr_ma_predA = corr_moving_average(ma_predA,i)
    corr_ma_predB = corr_moving_average(ma_predB, i)

    # Append filtered predictions to list of predictions
    predMLP.append(ma_predA); predGBR.append(ma_predB)
    predMLP.append(corr_ma_predA); predGBR.append(corr_ma_predB)

    # Print results
    print('\n')
    print('-------------- Window size: ' + str(i) + ' ------------------')
    print('MLP on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(predicted7, RUL[test_bat1])))
    print('MA MLP on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(ma_predA, RUL[test_bat1])))
    print('Corr MA MLP on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(corr_ma_predA, RUL[test_bat1])))
    print('GBR on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(predicted7B, RUL[test_bat1])))
    print('MA GBR on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(ma_predB, RUL[test_bat1])))
    print('Corr GBR MLP on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(corr_ma_predB, RUL[test_bat1])))



predMLPB = [predicted18] # List for predictions with MLP on test battery 2
predGBRB = [predicted18B] # List for predictions with MLP on test battery 2

# Filter predictions with moving average and corrected moving average with different window sizes
for i in [2, 3, 4, 7, 10]:
    # Predictions filtered with moving average
    ma_predA = moving_average(predicted18, i)
    ma_predB = moving_average(predicted18B, i)

    # Append filtered predictions to list of predictions
    corr_ma_predA = corr_moving_average(ma_predA, i)
    corr_ma_predB = corr_moving_average(ma_predB, i)

    # Append filtered predictions to list of predictions
    predMLPB.append(ma_predA); predGBRB.append(ma_predB)
    predMLPB.append(corr_ma_predA); predGBRB.append(corr_ma_predB)

    # Print results
    print('\n')
    print('-------------- Window size: ' + str(i) + ' ------------------')
    print('MLP on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(predicted18, RUL[test_bat2])))
```

```
        print('MA MLP on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(ma_predA, RUL[test_bat2])))
        print('Corr MA MLP on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(corr_ma_predA, RUL[test_bat2])))
        print('GBR on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(predicted18B, RUL[test_bat2])))
        print('MA GBR on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(ma_predB, RUL[test_bat2])))
        print('Corr GBR MLP on B0006 - MSE: ' + str(sklearn.metrics.mean_squared_error(corr_ma_predB, RUL[test_bat2])))

##########################################################################################################################
#                                          Visualisation
##########################################################################################################################

    ##########################################
    # Visualise degradation in discharge cycles
    ##########################################
    cycles = [1, 20, 50, 75, 100, 125, 168]
    legend = []
    column = 'Voltage_measured'
    plt.figure(1)
    plt.suptitle('Discharge cycle - Measurement degradation')
    subnum = 231
    plt.subplot(subnum)
    for cyc in cycles:
        plt.plot(disch_data[column].loc[cyc].index, disch_data[column].loc[cyc])
        legend.append(str(cyc))
    plt.legend(legend)
    plt.xlabel('Time [sec]')
    plt.ylabel('Voltage [V]')
    plt.title('Discharge - ' + column)

    legend = []
    column = 'Current_measured'
    plt.subplot(subnum + 1)  # plt.figure(2)
    for cyc in cycles:
        plt.plot(disch_data[column].loc[cyc].index, disch_data[column].loc[cyc])
        legend.append(str(cyc))
    plt.legend(legend)
    plt.xlabel('Time [sec]')
    plt.ylabel('Current [A]')
    plt.title('Discharge - ' + column)

    legend = []
    column = 'Temperature_measured'
    plt.subplot(subnum + 3)  # plt.figure(3)
    for cyc in cycles:
        plt.plot(disch_data[column].loc[cyc].index, disch_data[column].loc[cyc])
        legend.append(str(cyc))
    plt.legend(legend)
    plt.xlabel('Time [sec]')
    plt.ylabel('Temperature [C]')
    plt.title('Discharge - ' + column)

    legend = []
    column = 'Current_load'
    plt.subplot(subnum + 4)  # plt.figure(4)
    for cyc in cycles:
        plt.plot(disch_data[column].loc[cyc].index, disch_data[column].loc[cyc])
        legend.append(str(cyc))
    plt.legend(legend)
    plt.xlabel('Time [sec]')
    plt.ylabel('Current [A]')
    plt.title('Discharge - ' + column)

    legend = []
    column = 'Voltage_load'
    plt.subplot(subnum + 5)  # plt.figure(5)
    for cyc in cycles:
        plt.plot(disch_data[column].loc[cyc].index, disch_data[column].loc[cyc])
        legend.append(str(cyc))
    plt.legend(legend)
    plt.xlabel('Time [sec]')
    plt.ylabel('Voltage [V]')
    plt.title('Discharge - ' + column)


    ##########################################
    # Visualise degradation in charge cycles
    ##########################################
    cycles = [5, 20, 50, 75, 100, 125, 168]
    # cycles = [1,2,3,4,5,6]
    legend = []
    column = 'Voltage_measured'
    plt.figure(2)
    plt.suptitle('Charge cycle - Measurement degradation')
    subnum = 231
    plt.subplot(subnum)
    for cyc in cycles:
        plt.plot(charge_data[column].loc[cyc].index, charge_data[column].loc[cyc])
        legend.append(str(cyc))
    plt.legend(legend)
    plt.xlabel('Time [sec]')
    plt.ylabel('Voltage [V]')
    plt.title('Charge - ' + column)

    legend = []
    column = 'Current_measured'
    plt.subplot(subnum + 1)  # plt.figure(7)
    for cyc in cycles:
        plt.plot(charge_data[column].loc[cyc].index, charge_data[column].loc[cyc])
        legend.append(str(cyc))
    plt.legend(legend)
    plt.xlabel('Time [sec]')
    plt.ylabel('Current [A]')
    plt.title('Charge - ' + column)

    legend = []
```

```
column = 'Temperature_measured'
plt.subplot(subnum + 3)  # plt.figure(8)
for cyc in cycles:
    plt.plot(charge_data[column].loc[cyc].index, charge_data[column].loc[cyc])
    legend.append(str(cyc))
plt.legend(legend)
plt.xlabel('Time [sec]')
plt.ylabel('Temperature [C]')
plt.title('Charge - ' + column)

legend = []
column = 'Current_charge'
plt.subplot(subnum + 4)  # plt.figure(9)
for cyc in cycles:
    plt.plot(charge_data[column].loc[cyc].index, charge_data[column].loc[cyc])
    legend.append(str(cyc))
plt.legend(legend)
plt.xlabel('Time [sec]')
plt.ylabel('Current [A]')
plt.title('Charge - ' + column)

legend = []
column = 'Voltage_charge'
plt.subplot(subnum + 5)  # plt.figure(10)
for cyc in cycles:
    plt.plot(charge_data[column].loc[cyc].index, charge_data[column].loc[cyc])
    legend.append(str(cyc))
plt.legend(legend)
plt.xlabel('Time [sec]')
plt.ylabel('Voltage [V]')
plt.title('Charge - ' + column)


##########################################
# Visualise degradation in impedance cycles
##########################################
cycles = [1, 50, 100, 150, 200, 250, 270]
legend = []
column = 'Sense_current'
plt.figure(3)
plt.suptitle('Impedance cycle - Measurement degradation')
subnum = 231
plt.subplot(subnum)
for cyc in cycles:
    plt.plot(imp_data[column].loc[cyc].index, imp_data[column].loc[cyc])
    legend.append(str(cyc))
plt.legend(legend)
plt.xlabel('Time [sec]')
plt.ylabel('Current [A]')
plt.title('Impedance measure - ' + column)

legend = []
column = 'Battery_current'
plt.subplot(subnum + 1)
for cyc in cycles:
    plt.plot(imp_data[column].loc[cyc].index, imp_data[column].loc[cyc])
    legend.append(str(cyc))
plt.legend(legend)
plt.xlabel('Time [sec]')
plt.ylabel('Current [A]')
plt.title('Impedance measure - ' + column)

legend = []
column = 'Current_ratio'
plt.subplot(subnum + 1)
for cyc in cycles:
    plt.plot(imp_data[column].loc[cyc].index, imp_data[column].loc[cyc])
    legend.append(str(cyc))
plt.legend(legend)
plt.title('Impedance measure - ' + column)

##########################################################
# Visualise degradation in impedance cycles - Re and Rct
##########################################################
plt.figure(4)
plt.suptitle('Impedance Measures - Re and Rct')
subnum = 231
plt.subplot(subnum)
plt.plot(Batteries[0].index, Batteries[0]['Re'])
plt.plot(Batteries[1].index, Batteries[1]['Re'])
plt.plot(Batteries[2].index, Batteries[2]['Re'])
plt.plot(Batteries[3].index, Batteries[3]['Re'])
plt.xlabel('Cycles')
plt.ylabel('Re (Ohm)')
plt.legend(['B0005', 'B0006', 'B0018', 'B0007'])
plt.title('Impedance measure - Re')

plt.subplot(subnum + 1)
plt.plot(Batteries[0].index, Batteries[0]['Rct'])
plt.plot(Batteries[1].index, Batteries[1]['Rct'])
plt.plot(Batteries[2].index, Batteries[2]['Rct'])
plt.plot(Batteries[3].index, Batteries[3]['Rct'])
plt.xlabel('Cycles')
plt.ylabel('Rct')
plt.legend(['B0005', 'B0006', 'B0018', 'B0007'])
plt.title('Impedance measure - Rct')


#####################################################################
# Visualise degradation in capacity - Capacity threshold as red line
#####################################################################
plt.figure(5)
plt.plot(Batteries[0].index, Batteries[0]['capacity'])
plt.plot(Batteries[1].index, Batteries[1]['capacity'])
```

```
plt.plot(Batteries[2].index, Batteries[2]['capacity'])
plt.plot(Batteries[3].index, Batteries[3]['capacity'], 'black')
plt.plot(Batteries[3].index, np.ones(len(Batteries[0])) * 1.4, 'red')
plt.title('Capacity degradation')
plt.xlabel('Cycles')
plt.ylabel('Capacity (Ahr)')
plt.legend(['B0005', 'B0006', 'B0018', 'B0007'])


######################################################################
#                  Pretty prediction plot
######################################################################
plt.figure(6)
plt.suptitle('Battery life prediction')

plt.plot(range(1, len(predMLP[0]) + 1), RUL[test_bat1], 'blue')
plt.plot(range(1, len(predMLP[6]) + 1), predMLP[6], 'red')
plt.plot(range(1, len(predGBR[6]) + 1), predGBR[6], 'green')
plt.legend(['Actual remaining life', 'Prediction with ANN', 'Prediction with GBR'], loc='upper right',
           prop={'size': 14})
plt.xlabel('Cycle number', fontsize='large')
plt.ylabel('Remaining life [Cycles]', fontsize='large')
ax = plt.gcf().gca()
ax.tick_params(labelsize='large')

######################################################################
#        Predictions and filtered predictions (Window size=4)
######################################################################
plt.figure(7)
plt.suptitle('RUL Prediction')

subnum = 221
plt.subplot(subnum)
plt.plot(range(1, len(predMLP[0]) + 1), RUL[test_bat1])
plt.plot(range(1, len(predMLP[0]) + 1), predMLP[0])
plt.plot(range(1, len(predMLP[5]) + 1), predMLP[5])
plt.plot(range(1, len(predMLP[6]) + 1), predMLP[6])
# plt.plot(range(1, len(predMLP[9]) + 1), predMLP[9])
# plt.plot(range(1, len(predMLP[10]) + 1), predMLP[10])
plt.title('B0006')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'MLP Prediction', 'Filter W=4', 'Corrected W=4'])
plt.subplot(subnum + 1)
# plt.figure(17)
plt.plot(range(1, len(predGBR[0]) + 1), RUL[test_bat1])
plt.plot(range(1, len(predGBR[0]) + 1), predGBR[0])
plt.plot(range(1, len(predGBR[5]) + 1), predGBR[5])
plt.plot(range(1, len(predGBR[6]) + 1), predGBR[6])
# plt.plot(range(1, len(predGBRB[5]) + 1), predGBRB[5])
# plt.plot(range(1, len(predGBRB[7]) + 1), predGBRB[7])
# plt.plot(range(1, len(predGBRB[9]) + 1), predGBRB[9])
plt.title('B0006')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'GBR Prediction', 'Filter W=4', 'Corrected W=4'])
plt.subplot(subnum + 2)
plt.plot(range(1, len(predMLPB[0]) + 1), RUL[test_bat2])
plt.plot(range(1, len(predMLPB[0]) + 1), predMLPB[0])
plt.plot(range(1, len(predMLPB[5]) + 1), predMLPB[5])
plt.plot(range(1, len(predMLPB[6]) + 1), predMLPB[6])
# plt.plot(range(1, len(predMLPB[6]) + 1), predMLPB[6])
# plt.plot(range(1, len(predMLPB[8]) + 1), predMLPB[8])
# plt.plot(range(1, len(predMLPB[10]) + 1), predMLPB[10])
plt.title('B0018')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'MLP Prediction', 'Filter W=4', 'Corrected W=4'])
plt.subplot(subnum + 3)
# plt.figure(17)
plt.plot(range(1, len(predGBRB[0]) + 1), RUL[test_bat2])
plt.plot(range(1, len(predGBRB[0]) + 1), predGBRB[0])
plt.plot(range(1, len(predGBRB[5]) + 1), predGBRB[5])
plt.plot(range(1, len(predGBRB[6]) + 1), predGBRB[6])
# plt.plot(range(1, len(predGBRB[6]) + 1), predGBRB[6])
# plt.plot(range(1, len(predGBRB[8]) + 1), predGBRB[8])
# plt.plot(range(1, len(predGBRB[10]) + 1), predGBRB[10])
plt.title('B0018')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'GBR Prediction', 'Filter W=4', 'Corrected W=4'])

######################################################################
#             Predictions without filtering
######################################################################
plt.figure(8)
plt.suptitle('RUL Prediction')
subnum = 231
plt.subplot(subnum)
plt.plot(range(1, len(predicted7) + 1), RUL[test_bat1])
plt.plot(range(1, len(predicted7) + 1), predicted7)
plt.plot(range(1, len(predicted7B) + 1), predicted7B)
plt.title('B0006')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'MLP Prediction', 'GBR Prediction'])

plt.subplot(subnum+1)
plt.plot(range(1, len(predicted18) + 1), RUL[test_bat2])
plt.plot(range(1, len(predicted18) + 1), predicted18)
plt.plot(range(1, len(predicted18B) + 1), predicted18B)
plt.title('B0018')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
```

```
plt.legend(['Actual RUL', 'MLP Prediction', 'GBR Prediction'])


#####################################################################
#  Predictions with 5 different window sizes for filtering - B0006
#####################################################################
plt.figure(9)
plt.suptitle('RUL Prediction')

subnum = 231
plt.subplot(subnum)
plt.plot(range(1, len(predMLP[0]) + 1), RUL[test_bat1])
plt.plot(range(1, len(predMLP[0]) + 1), predMLP[0])
plt.plot(range(1, len(predMLP[1]) + 1), predMLP[1])
plt.plot(range(1, len(predMLP[3]) + 1), predMLP[3])
plt.plot(range(1, len(predMLP[5]) + 1), predMLP[5])
plt.plot(range(1, len(predMLP[7]) + 1), predMLP[7])
plt.plot(range(1, len(predMLP[9]) + 1), predMLP[9])
plt.title('B0006')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'MLP Prediction', 'W=2', 'W=3', "W=4", "W=7", "W=10"])
plt.title('MLP Predictions on B0006 - Moving Average')

plt.subplot(subnum + 1)
plt.plot(range(1, len(predGBR[0]) + 1), RUL[test_bat1])
plt.plot(range(1, len(predGBR[0]) + 1), predGBR[0])
plt.plot(range(1, len(predGBR[1]) + 1), predGBR[1])
plt.plot(range(1, len(predGBR[3]) + 1), predGBR[3])
plt.plot(range(1, len(predGBR[5]) + 1), predGBR[5])
plt.plot(range(1, len(predGBR[7]) + 1), predGBR[7])
plt.plot(range(1, len(predGBR[9]) + 1), predGBR[9])
plt.title('B0006')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'GBR Prediction', 'W=2', 'W=3', "W=4", "W=7", "W=10"])
plt.title('GBR Predictions on B0006 - Moving Average')

plt.subplot(subnum+2)
plt.plot(range(1, len(predMLP[0]) + 1), RUL[test_bat1])
plt.plot(range(1, len(predMLP[0]) + 1), predMLP[0])
plt.plot(range(1, len(predMLP[2]) + 1), predMLP[2])
plt.plot(range(1, len(predMLP[4]) + 1), predMLP[4])
plt.plot(range(1, len(predMLP[6]) + 1), predMLP[6])
plt.plot(range(1, len(predMLP[8]) + 1), predMLP[8])
plt.plot(range(1, len(predMLP[10]) + 1), predMLP[10])
plt.title('B0006')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'MLP Prediction', 'W=2', 'W=3', "W=4", "W=7", "W=10"])
plt.title('MLP Predictions on B0006 - Corrected Moving Average')

plt.subplot(subnum + 3)
plt.plot(range(1, len(predGBR[0]) + 1), RUL[test_bat1])
plt.plot(range(1, len(predGBR[0]) + 1), predGBR[0])
plt.plot(range(1, len(predGBR[2]) + 1), predGBR[2])
plt.plot(range(1, len(predGBR[4]) + 1), predGBR[4])
plt.plot(range(1, len(predGBR[6]) + 1), predGBR[6])
plt.plot(range(1, len(predGBR[8]) + 1), predGBR[8])
plt.plot(range(1, len(predGBR[10]) + 1), predGBR[10])
plt.title('B0006')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'GBR Prediction', 'W=2', 'W=3', "W=4", "W=7", "W=10"])
plt.title('GBR Predictions on B0006 - Corrected Moving Average')

#####################################################################
#   Predictions with 5 different window sizes for filtering - B0018
#####################################################################
plt.figure(10)
plt.suptitle('RUL Prediction')

subnum = 231
plt.subplot(subnum)
plt.plot(range(1, len(predMLPB[0]) + 1), RUL[test_bat2])
plt.plot(range(1, len(predMLPB[0]) + 1), predMLPB[0])
plt.plot(range(1, len(predMLPB[1]) + 1), predMLPB[1])
plt.plot(range(1, len(predMLPB[3]) + 1), predMLPB[3])
plt.plot(range(1, len(predMLPB[5]) + 1), predMLPB[5])
plt.plot(range(1, len(predMLPB[7]) + 1), predMLPB[7])
plt.plot(range(1, len(predMLPB[9]) + 1), predMLPB[9])
plt.title('B0018')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'MLP Prediction', 'W=2', 'W=3', "W=4", "W=7", "W=10"])
plt.title('MLP Predictions on B0018 - Moving Average')

plt.subplot(subnum + 1)
plt.plot(range(1, len(predGBRB[0]) + 1), RUL[test_bat2])
plt.plot(range(1, len(predGBRB[0]) + 1), predGBRB[0])
plt.plot(range(1, len(predGBRB[1]) + 1), predGBRB[1])
plt.plot(range(1, len(predGBRB[3]) + 1), predGBRB[3])
plt.plot(range(1, len(predGBRB[5]) + 1), predGBRB[5])
plt.plot(range(1, len(predGBRB[7]) + 1), predGBRB[7])
plt.plot(range(1, len(predGBRB[9]) + 1), predGBRB[9])
plt.title('B0018')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'GBR Prediction', 'W=2', 'W=3', "W=4", "W=7", "W=10"])
plt.title('GBR Predictions on B0018 - Moving Average')

plt.subplot(subnum + 2)
plt.plot(range(1, len(predMLPB[0]) + 1), RUL[test_bat2])
plt.plot(range(1, len(predMLPB[0]) + 1), predMLPB[0])
```

```
plt.plot(range(1, len(predMLPB[2]) + 1), predMLPB[2])
plt.plot(range(1, len(predMLPB[4]) + 1), predMLPB[4])
plt.plot(range(1, len(predMLPB[6]) + 1), predMLPB[6])
plt.plot(range(1, len(predMLPB[8]) + 1), predMLPB[8])
plt.plot(range(1, len(predMLPB[10]) + 1), predMLPB[10])
plt.title('B0018')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'MLP Prediction', 'W=2', 'W=3', "W=4", "W=7", "W=10"])
plt.title('MLP Predictions on B0006 - Corrected Moving Average')

plt.subplot(subnum + 3)
plt.plot(range(1, len(predGBRB[0]) + 1), RUL[test_bat2])
plt.plot(range(1, len(predGBRB[0]) + 1), predGBRB[0])
plt.plot(range(1, len(predGBRB[2]) + 1), predGBRB[2])
plt.plot(range(1, len(predGBRB[4]) + 1), predGBRB[4])
plt.plot(range(1, len(predGBRB[6]) + 1), predGBRB[6])
plt.plot(range(1, len(predGBRB[8]) + 1), predGBRB[8])
plt.plot(range(1, len(predGBRB[10]) + 1), predGBRB[10])
plt.title('B0018')
plt.xlabel('Cycle')
plt.ylabel('RUL [Cycles]')
plt.legend(['Actual RUL', 'GBR Prediction', 'W=2', 'W=3', "W=4", "W=7", "W=10"])
plt.title('GBR Predictions on B0018 - Corrected Moving Average')

# Show figures
plt.show()
```

Benchmark

- Benchmark

Benchmark.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import time
import random

from sklearn.cluster import KMeans
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, mean_absolute_error

# Adds columns for time (nr. of cycles) spent in each of the 6 modes
def add_mode_cycles(data):
    data['Mode 1'] = 0; data['Mode 2'] = 0; data['Mode 3'] = 0
    data['Mode 4'] = 0; data['Mode 5'] = 0; data['Mode 6'] = 0

    ind = data.index.get_level_values(0)
    units = len(ind.unique())
    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()
        column = np.zeros((unit_len, 6), int)

        for index, row in data.loc[unit].iterrows():
            mode = int(row['Operational Mode'])
            if index != 1:
                column[index - 1] = column[index - 2]
            column[index - 1][mode] = column[index - 1][mode] + 1

        data.loc[unit].loc[:, data.columns[-6::]] = column

    return data

# Transforms the data from one range to another (0 to 1 or based on mean and standard deviation in each feature)
def normalize(df, zero2one=False):
    result = df.copy()
    for feature_name in df.columns:
        if zero2one == True:
            min = df[feature_name].min()
            max = df[feature_name].max()
            result[feature_name] = (df[feature_name] - min) / (max - min)
        else:
            mean = df[feature_name].mean()
            std = df[feature_name].std()
            result[feature_name] = (df[feature_name] - mean) / (std)

    return result

# Adds a linear labeled RUL column
def linear_label_rul(data):
    data['RUL'] = 0

    ind = data.index.get_level_values(0)
    units = len(ind.unique())
    for unit_num in range(units):
        unit = unit_num + 1
        unit_len = data.loc[unit].index.max()
        data.loc[unit, 'RUL'] = range(unit_len, 0, -1)

    return data

if __name__ == '__main__':

    start_time = time.clock()

    # Load Dataset
    start = time.clock()
    data = pd.read_csv('train.txt', sep=" ", header=None, index_col=[0, 1],
                       names=["operational_setting 1",
                              "operational_setting 2", "operational_setting 3", "sensor 1", "sensor 2", "sensor 3",
                              "sensor 4", "sensor 5", "sensor 6", "sensor 7", "sensor 8", "sensor 9", "sensor 10",
                              "sensor 11", "sensor 12", "sensor 13", "sensor 14", "sensor 15", "sensor 16", "sensor 17",
                              "sensor 18", "sensor 19", "sensor 20", "sensor 21", "", ""])
    data = data[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]]
    data.to_csv('DatasetForBenchmark.csv')

    # Reading .csv fil can be used after first run. Then the code above can be commented
    # data = pd.read_csv('DatasetForBenchmark.csv', sep=',', header=0, index_col=[0, 1])
    load_data_time = format((time.clock() - start), '.2f')

    # Clusters data into 6 clusters and add it to dataset as operational mode
    start = time.clock()
```

```
kmeans = KMeans(n_clusters=6).fit(data[[0, 1, 2]])
operational_mode = kmeans.predict(data[[0, 1, 2]])
clustering_time = format((time.clock() - start), '.2f')
data['Operational Mode'] = operational_mode

# Adds mode cycles to data
start = time.clock()
data = add_mode_cycles(data)
add_mode_cycles_time = format((time.clock() - start), '.2f')

# Labels the dataset with linear RUL
start = time.clock()
norm_data = linear_label_rul(data)
rul_label_time = format((time.clock() - start), '.2f')

# Separate Train and Test
start = time.clock()
random.seed(a=5)
rnd_vector = random.sample(range(1, 219), 218)
training_units = rnd_vector[0:145]
testing_units = rnd_vector[145::]

# Splits the dataset into training and test sets
test_set_norm = norm_data[norm_data.columns[3::]].loc[testing_units].copy()
train_set_norm = norm_data[norm_data.columns[3::]].loc[training_units].copy()

# Further split the train and test dataset to not contain the RUL
x_train = train_set_norm[train_set_norm.columns[0:-1]]
x_test = test_set_norm[test_set_norm.columns[0:-1]]

# Split train set to target values (RUL)
y_train = train_set_norm.RUL
y_test = test_set_norm.RUL
split_train_test_time = format((time.clock() - start), '.2f')

# Normalise data from 0 to 1
start = time.clock()
x_train = normalize(x_train, zero2one=True)
x_test = normalize(x_test, zero2one=True)
normalize_time = format((time.clock() - start), '.2f')

################################### Machine learning ############################################################
linReg = linear_model.LinearRegression(normalize=True)

# Fit the ML model to the train set
start = time.clock()
linReg.fit(x_train, y_train)
training_time = format((time.clock() - start), '.2f')

# Dataframe for predicted results (Preres = predicted results)
preres = pd.DataFrame(y_test.values, index=y_test.index, columns=['Actual RUL'])

# Predicts with the ML model
start = time.clock()
prediction = linReg.predict(x_test)
prediction_time = format((time.clock() - start), '.2f')
preres['LinReg'] = prediction

# Evaluation of MSE and MAE
start = time.clock()
mse = mean_squared_error(y_test, prediction)
mae = mean_absolute_error(y_test, prediction)
evaluation_time = format((time.clock() - start), '.2f')

# Visualise the predictions
start = time.clock()
fig = plt.figure(1)
plt.suptitle('')
subnum = 231
for i in range(30, 36):
    plt.subplot(subnum)
    j = testing_units[i]
    plt.plot(y_test.loc[j].index, preres.loc[j]['Actual RUL'], 'blue')
    plt.plot(y_test.loc[j].index, preres.loc[j]['LinReg'], 'red')
    plt.xlabel('Cycles')
    plt.ylabel('RUL [Cycles]')
    plt.legend(['Actual RUL', 'LinReg Predicted'], loc='lower left')
    plt.title('Engine ' + str(j))
    subnum = subnum + 1
visualization_time = format((time.clock() - start), '.2f')

total_time = format((time.clock() - start_time), '.2f')

# Printing all results
```

```
print('Load dataset: ' + str(load_data_time) + 'sec')
print('Clustering: ' + str(clustering_time) + 'sec')
print('Add mode cycles: ' + str(add_mode_cycles_time) + 'sec')
print('RUL Label: ' + str(rul_label_time) + 'sec')
print('Split train and test dataset: ' + str(split_train_test_time) + 'sec')
print('Normalizing: ' + str(normalize_time) + 'sec')
print('LinReg Training time: ' + str(training_time) + 'sec')
print('LinReg Prediction: ' + str(prediction_time) + 'sec')
print('Evaluation: ' + str(evaluation_time) + 'sec' + ' || MSE = ' + str(mse) + ' || MAE = ' + str(mae))
print('Visualization: ' + str(visualization_time) + 'sec')
print('Total time: ' + str(total_time) + 'sec')

plt.show()
```