Remaining Useful Life Estimation of Turbofan Engines
- A Recurrent Neural Network Approach -

last edited 08.05.2021

by Robin Hertel
(LinkedIn: linkedin.com/in/robin-hertel-0a1384190)

TLDR

The switch from conventional to predictive maintenance promises industrial companies significant efficiency gains. In practice, however, the successful use of predictive maintenance often fails due to great uncertainties in the assessment of a system's health status. This project shows how the current megatrend of the Internet of Things can be utilized to develop a viable and robust estimator of a system's remaining useful life. In particular, various filter and smoothing methods are implemented and tested to account for sensor noise. Additionally, a recurrent neural network model with a stacked LSTM architecture is defined which allows remaining useful life estimation without the necessity of an underlying structural fault propagation model.

---

1. Introduction

Presumably, we all have been there: We use a tool or a device and suddenly it breaks. If this tool is our pan filled with pasta sauce and its handle breaks on our way to the dinner table - well, that's just some wiping up and calling the delivery service. However, if this tool is a system of precisely coordinated components in an industrial production process, unexpected failures may cause major inefficiencies.

In order to minimize unexpected system failures in production, systems usually get serviced after a fixed period of time or after a fixed number of pieces produced. This in turn leads to inefficiencies, though: Either these preventive maintenance intervals are very short and many systems are needlessly serviced or unexpected failures still occur too often.

Trends such as the Internet of Things and the increasing use of sensors to monitor system health more and more facilitate the data-driven prognostics approach of predictive maintenance. With predictive maintenance, the goal is to estimate the systems' remaining useful life (RUL), for example based on sensor data, and to conduct maintenance actions only when some threshold is reached.

However, to succesfully switch from preventive to predictive maintenance, careful analysis of the data and the underlying degradation process is required: A first problem practitioners often face is that sensor data is usually contaminated with noise and then ways must first be found to separate the signal from the noise. Another common problem is that a great variety of potentially interacting factors affect the systems' degradation trajectories, rendering the examined process practically stochastic and difficult to model.

Based on the classic C-MAPSS data of simulated turbofan engine degradation from NASA (Saxena et al., 2008) this project aims to show how a viable and robust RUL estimator can be developed without having to specify a structural (process) model. The project's outline is as follows: Firstly, some data exploration is conducted to get to know the data. Secondly, multiple data preprocessing steps are applied; for example, various filter and smoothing methods are implemented and tested to account for sensor noise. Thirdly, a recurrent neural network with a stacked LSTM architecture is defined to circumvent the modeling problem. Additionally, the neural network is trained with a customized loss function that accounts for the asymmetric preferences of under- vs. overestimating a system's RUL.

The results of the experiment carried out in this project show that the proposed recurrent neural network model strictly outperforms corresponding feedforward neural networks which cannot factor in temporal dynamics. Furthermore, the results indicate that the optimal estimation strategy depends on the length of the available sensor time series: If only short sequences are available, the best performance is reached when the input data are first filtered with the Kalman filter. If at least moderately long sequences are available, the unfiltered time series can be used as input data to the proposed model without degradation of performance.

The following pages contain - a hopefully tolerable dose of - math, the overall discussion as well as the code, which is written in Python, its output and the associated comments. The code is most easily run from command line with the arguments

- *descriptives*,
- *preprocessing* and
- *estimation*

to execute only the specified subset of the entire project. If *all* is passed as an argument, the entire code is run. For each of these subsets, there is a separate module with subset specific functions in the project's folder in order to keep the code structured. Note that the project's master file is *tfed.py* and the code shown in this README is taken from this file unless explicitly stated otherwise. The project's folder does not contain the actual data. Instead, it is assumed that there is a text file *input_path.txt* in the folder, from which the path to the data can be extracted as a string.

Let's start working with the code: Code box 1 below shows the first lines to be executed in *tfed.py*. We can see that the first thing *tfed.py* does is to check whether it is indeed run as the main file (over the console). This test prevents *tfed.py* from being imported as a module and executed by another program and is required, for example, by the *multiprocessing* module which we will use later.

Code box 1:

```
[001] # main file check
[002] if __name__ == '__main__':
[003]
```

Admittedly, it would have been a pretty small project if we hadn't started *tfed.py* as the main file. So let's assume that the test in Code box 1 evaluates to *True* and we can move on to Code box 2. Lines 3 - 12 of Code box 2 below simply import standard Python modules and packages. As can be seen in lines 13 - 16, *tfed.py* opens a text file *working_directory.txt* which is assumed to be located in the current working directory. It is assumed that *working_directory.txt* contains the path to the project's folder. Line 37 adds the loss function, which we will define and discuss in section 4, to the built-in losses of the *keras* package.

And that's it: We can finally start working with the data.

Code box 2:

```
[001] ### Preliminaries
[002] # load external modules
[003] import os
[004] import sys
[005] import numpy as np
[006] import pandas as pd
[007] import matplotlib.pyplot as plt
[008] import math
[009] import keras.losses
[010] from copy import deepcopy
[011] from multiprocessing import Process as process
[012] import tensorflow as tf
[013] # change cwd
[014] with open(os.path.dirname(__file__) + '\\working_directory.txt') as wdir_file:
[015]     wdir = wdir_file.read()
[016]     os.chdir(wdir)
[017] # load modules from repository
[018] from tfed_descriptives import (
[019]         depvar_hists,
[020]         sensor_means_up_to_failure,
[021]         indepvar_cmap
[022] )
[023] from tfed_preprocessing import (
[024]         var_init_mean_adder,
[025]         kmeans_clustering,
[026]         target_clusterer,
[027]         rescale_data,
[028]         denoise_data,
[029]         standardizer,
[030]         feature_selector,
[031]         train_df_generator,
[032]         test_df_generator
[033] )
[034] from tfed_scoring import tf_wclf
[035] from tfed_models import RNN
[036] # add custom loss function to keras.losses
[037] keras.losses.tf_wclf = tf_wclf
[038]
```

2. Data exploration

2.1 Basic information about the data

Before we load the data, let's skim the readme file that comes with it: The data is provided as a collection of zip-compressed text files in which the values are delimited by spaces. The collection consists of a total of four different data sets, *FD001 - FD004*, each of which is subdivided into a train set and a test set. Each data set tracks multiple engines over time where time is given in cycles. For each engine, we have one observation per cycle. Apart from the engine id and the cycle number, we are provided with 24 additional variables per observation: The first three variables contain data on the operational setting and the remaining 21 variables are sensor data. The data sets differ in terms of the number of engines observed, the number of operational conditions and the number of fault modes.

While the training data sets are complete records of the engines' degradation trajectories, i.e. the last observation of an engine in the data indeed corresponds to the last operating cycle before failure, the test data are truncated some unknown number of cycles before failure. The aim of the project is to estimate the number of cycles remaining for each engine id in the test data set. To check the results, the vector of true cycles remaining is also included in the collection for each test data set. Additionally, the readme informs us that the sensor data are subject to noise and that the engines vary in their initial health; however none of the engines already starts with a fault condition.

Since the target measure of this project will be prediction accuracy, we cannot simply combine all training data sets into one due to the variation in operating conditions and fault modes. Therefore, we will focus on estimating the number of remaining operational cycles of one of the test data sets, namely the test data set of *FD002* which has six operational conditions and one fault mode.

2.2 Loading the data

Back to the code. Lines 3 and 4 of code box 2 below use the arguments with which the script was started to check whether it is necessary to load the data. We want to do just that, so let's continue with the indented part of the box. It is assumed that there is a file called *input_path.txt* in our current working directory which only contains the path to the location of the raw input data. In lines 6 and 7 this file is opened and the input path string is saved into the global environment.

Code box 2:

```
[001] ### load data
[002] # check if mode passed as system argument requires initial loading
[003] if 'all' in sys.argv or 'descriptives' in sys.argv \
[004] or 'preprocessing' in sys.argv:
[005]     # define input path
[006]     with open('input_path.txt', 'r') as path_file:
[007]         input_path = path_file.read()
[008]
```

From the readme file that comes with the data we can infer the data sets' column names. Lines 2 - 7 of code box 3 below save the list of all names to the global environment.

Code box 3:

```
[001] # define column names
[002] colnames = ['unit', 'cycle', 'op_setting_1', 'op_setting_2',
[003]             'op_setting_3', 'sensor_1', 'sensor_2', 'sensor_3', 'sensor_4',
[004]             'sensor_5', 'sensor_6', 'sensor_7', 'sensor_8', 'sensor_9',
[005]             'sensor_10', 'sensor_11', 'sensor_12', 'sensor_13',
[006]             'sensor_14', 'sensor_15', 'sensor_16', 'sensor_17',
[007]             'sensor_18', 'sensor_19', 'sensor_20', 'sensor_21']
[008]
[009] # load all tables needed
[010] data = {}
[011] for table in ['train_FD002', 'test_FD002', 'train_FD004']:
[012]     data[table] = pd.read_table(filepath_or_buffer=input_path+'\\'\
[013]                                 +table+'.txt', sep='\s', header=None,
[014]                                 names=colnames, engine='python')
[015]
[016] test_rul = np.array(pd.read_table(filepath_or_buffer=input_path+'\\'\
[017]                                     +'RUL_FD002.txt', sep='\s', header=None,
[018]                                     names=['RUL'], engine='python'))
[019]
```

When reading the data's readme, one might notice that *FD002* and *FD004* are related to some extent: Both data sets share a fault mode. Since we want to explore the data thoroughly in this chapter and since we are of course always highly motivated, we might find it worthwhile to load the train data set of *FD004* together with *FD002* and compare the two data sets.

In order to organize all these data sets in a way that is comfortable for us, we first initialize a dictionary in line 10 of code box 3. In lines 11 - 14, we iterate over the data sets and save them with their respective names as keys in our dictionary. Then, in lines 16 - 18, we save the vector of the true number of cycles remaining of the *FD002* test set as a numpy array.

2.3 Descriptive statistics

Now that we have loaded the data and learned its basic organizational structure, we are ready to statistically examine the data. So let's first output a table to the console that shows the number of observations, measures for central tendency and dispersion as well as the quartiles for each variable. A printout of these tables can be found in the appendix (see Ap1 and Ap2).

In code box 4 below we can ignore the system variable check again and directly go to the indented block of code. As can be seen in lines 7 and 8, we use the *describe* method of *pandas* to print out the descriptive statistics. Since we only saw a subset of all variables with the default settings of *pandas*, we temporarily adjust its printing settings in line 6 so that the statistics of all variables are printed to the console.

Code box 4:

```
[001] ### inspect and visualize data
[002] # check if mode passed as system argument requires descriptives
[003] if 'all' in sys.argv or 'descriptives' in sys.argv:
[004]
[005]     # get descriptive statistics tables
[006]     with pd.option_context('display.max_columns', len(colnames)):
[007]         print('FD002 descriptives:', '\n', data['train_FD002'].describe())
[008]         print('FD004 descriptives:', '\n', data['train_FD004'].describe())
[009]
```

Scrolling through the output, we notice a few things:

1. All variables are numeric.
2. We don't have a missing values problem. We can conclude this since a) we have not been given a numeric indicator for missing values, b) min and max values do not suggest that there are extreme outliers that represent missing values, c) no non-numeric characters have been used and d) all columns have the same length.
3. There are no variables with zero variance.

4. Operational condition variables appear to follow the same distributions across the two data sets.
5. There are considerable differences in central tendency and dispersion between the variables within each data set.

We will make use of - or account for, respectively - points 1, 4 and 5 later. Points 2 and 3 simply come in handy for us. If point 2 would not hold we would either have to impute missing values or remove observations with missing values. The latter meant that we threw away information in case the values would be missing only partially, like only a subset of a row's cells would be missing. Luckily, though, we don't have this problem here. Point 3 is important to us because variables with zero variance do not carry any additional information that can be exploited later by our model.

2.4 Data visualization

Of all the variables, *cycle* is of particular interest to us. RUL estimation requires some measure of time or something that can be thought of as time. However, there is no explicit *time* variable in our data set. We are going to use *cycle* instead of it. Note, however, that by doing so, we implicitly assume that average cycle lengths are equal across all engines.

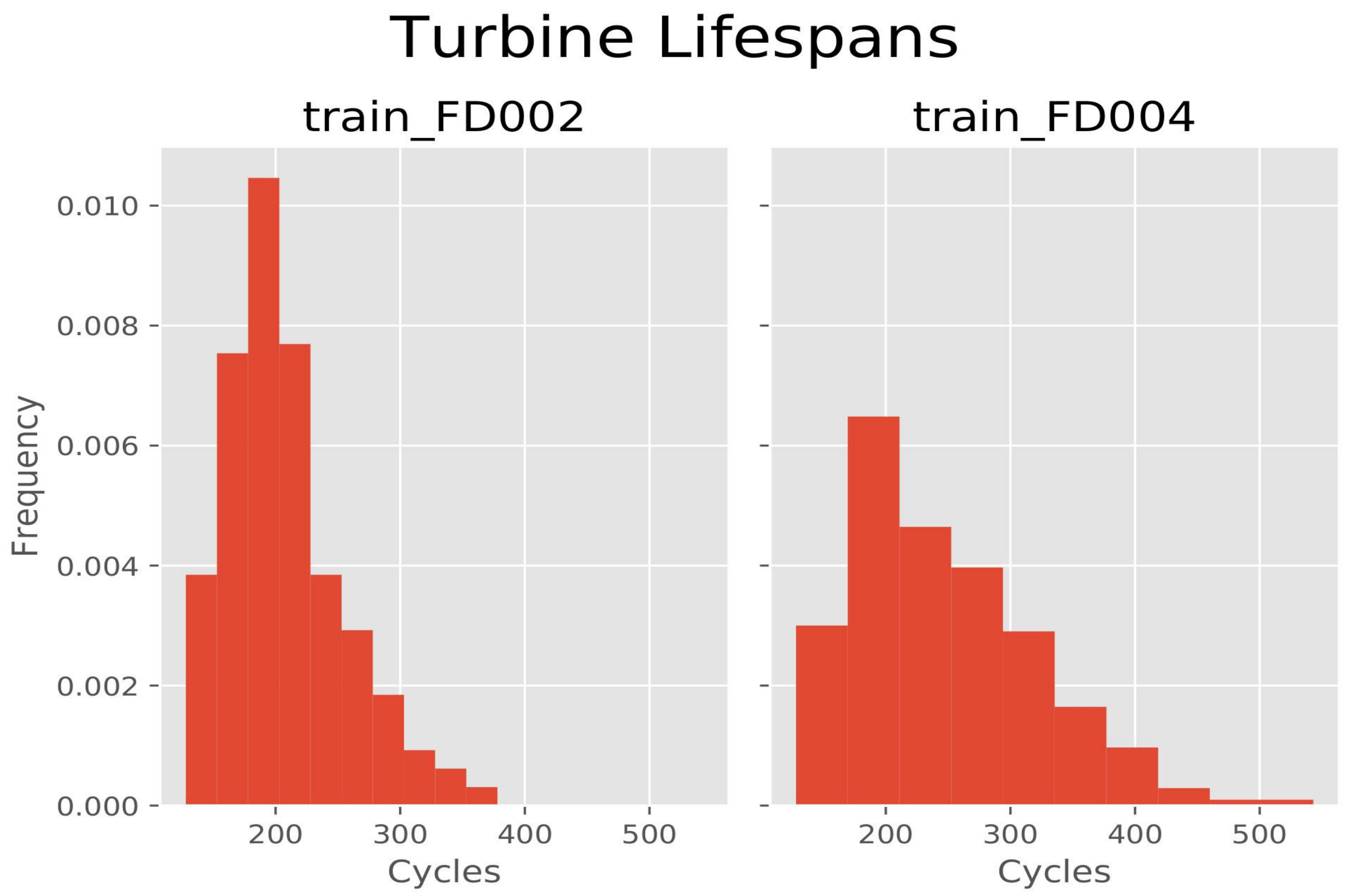
In order to understand *cycle* better, its distribution is now shown in a next step for both training data sets using histograms. As can be seen in code box 5 below, this is achieved simply by calling *depvar_hists* from the *tfed_descriptives* module which we previously loaded from our current working directory. In short, *depvar_hists* is a helper function that accepts a dictionary of data sets, *data* in our case, and the keys of those data sets that are to be plotted as parameters. Then, for each data set, the function appends each engine's maximum cycle number to a *max_cycles* list. Based on the data stored in *max_cycles*, a histogram is plotted and saved to the current working directory.

Code box 5:

```
[001] # plot histograms of dependent variables
[002] depvar_hists(data, keys = ['train_FD002', 'train_FD004'])
[003]
```

Line 2 of code box 5 produces figure 1 below.

Figure 1: Histograms of maximum number of cycles by training data set



A first quick look at the histograms reveals that *max_cycles* is distributed similarly in both data sets. Both histograms show an unimodal distribution where the mode is around 200 cycles. Furthermore, we can see that the engines' lifespans follow a right skewed distribution in both training data sets. We might recognize this pattern from other duration analysis projects. Right skewed distributions often emerge in processes where entire systems fail as soon as one critical component reaches its failure criterion.

However, one can also see differences: The histogram of *FD004* appears more spread out than the one of *FD002* which indicates a higher variability of the engines' lifespans in the *FD004* data. This might be due to the fact that we may actually see the overlay of two distinct histograms in *FD004*, namely one for each of the two fault modes.

Now that we have a basic understanding of our variables' unconditional distributions, we might be interested in the expected feature values conditional on the number of cycles until failure.

Code box 6:

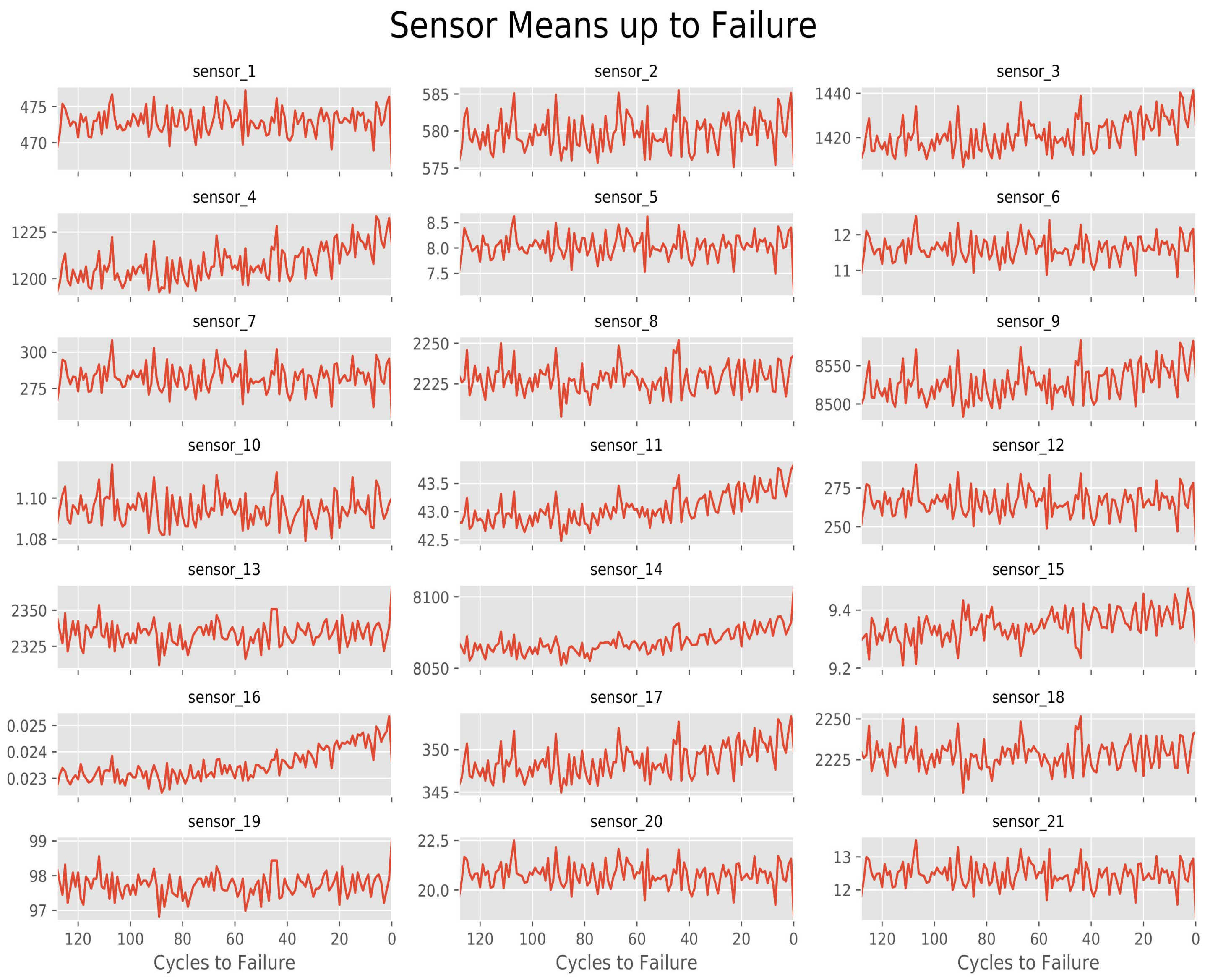
```
[001] # we first focus on tables FD002
[002] train_df = data['train_FD002']
[003] test_df = data['test_FD002']
[004]
[005] # plot sensor means conditional on how many cylces to failure
[006] sensor_means_up_to_failure(df=train_df)
[007]
```

For the sake of brevity in the following lines, in lines 2 and 3 of code box 6 we pull the train and the test set of *FD002* out of our *data* dictionary and store these two data sets directly in the global environment. Then, in line 6, we call *sensor_means_up_to_failure*, another little helper function we have already imported from *tfed_descriptives*.

To summarize *sensor_means_up_to_failure* briefly: First, it takes a data frame as input. Second, for every distinct unit (= every distinct engine) it gets the maximum number of cycles. Third, for every (unit, cycle) - pair it computes the difference *maximum number of cycles* - *current cycle* which translates to number of cycles until failure. Lastly, for each sensor the values are averaged over all units along the values of *number of cycles until failure*. The resulting line plots are then saved in one file to the current working directory. It has to be noted that only the last *x* cycles are plotted where *x* is the largest cycle number all engines have in common.

This call produces figure 2 below.

Figure 2: Sensor means up to failure



The most striking characteristic of figure 2 is probably the degree of oscillation in all graphs. On the one hand, this oscillation might be due to the noise that the sensor time series are contaminated with. On the other hand, this could be due to the fact that we average over all engines and all operating conditions in figure 2. Therefore, it might be worthwhile to look at a non-aggregated sensor time series.

Code box 7:

```
[001] # plot non-aggregated time series of a sensor
[002] fig = plt.figure()
[003] plt.title('Sensor 4 time series of unit 1', fontsize=22)
[004] plt.xlabel('Cycle')
[005] plt.ylabel('Sensor Value')
[006] plt.plot(train_df.loc[train_df['unit']==1, 'sensor_4'])
[007] fig.savefig('descriptives2_2_sensor4_unit1.pdf')
[008] plt.close()
[009]
```

Lines 2 - 8 of code box 7 generate an exemplary non-aggregated time series, namely the time series of sensor four of the engine with id one, and store the resulting figure to the current working directory.

Figure 3: Time series of unit 1, sensor 4

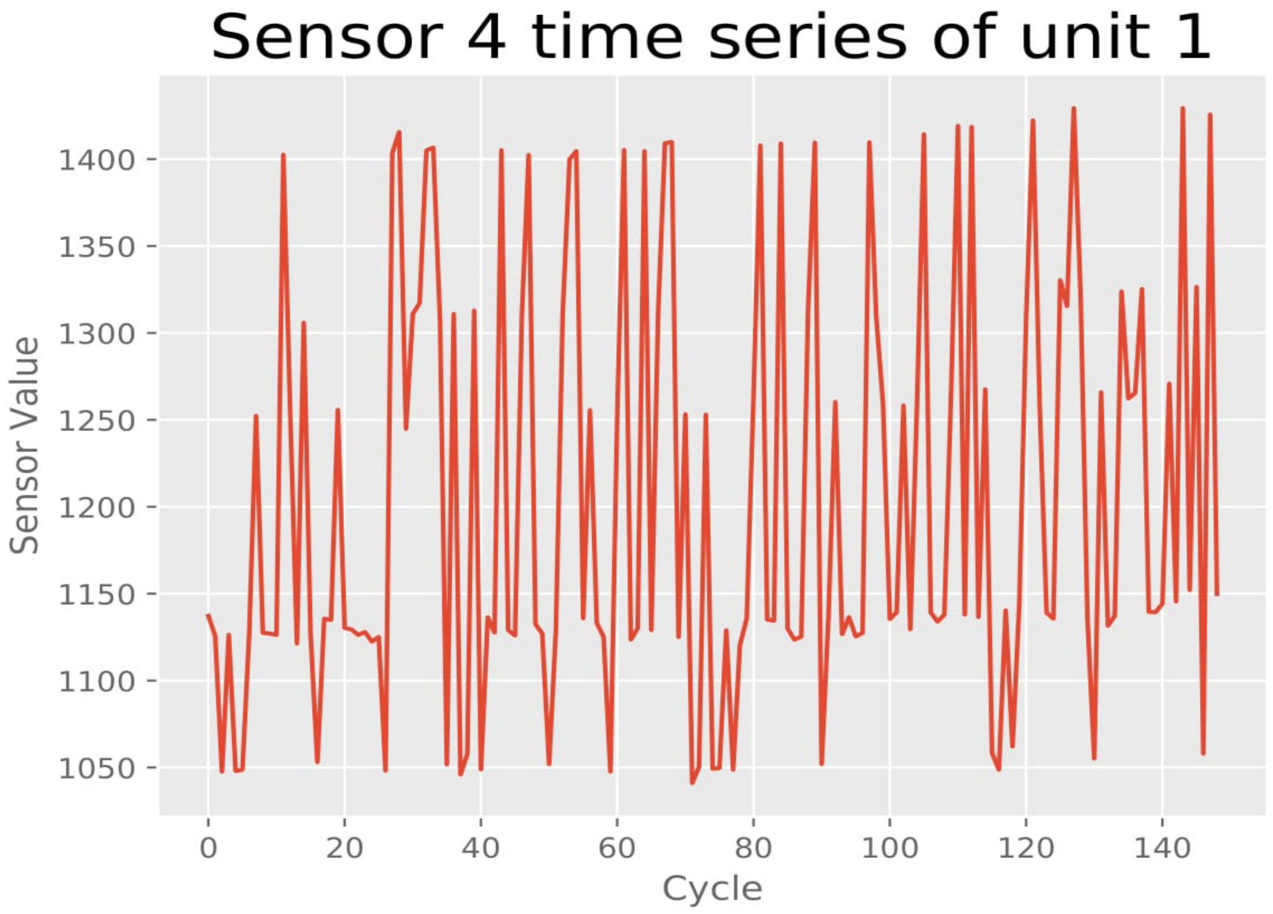


Figure 3 shows that oscillation is the predominant characteristic even with non-aggregated sensor time series. If one looks more closely, however, one can see that there are four to six different levels, all of which share a common upward trend. While these different levels may correspond to the different operating conditions present in *FD002,* the common upward trend may indicate the development of the fault condition. It should be noted, though, that the effect of a change of the operating condition appears to significantly outweigh the effect of the fault development. From this we can conclude that in the next steps we not only have to account for sensor noise but also for the different operating conditions.

Now we have already got to know our variables' unconditional distributions as well as their average values conditional on number of cycles to failure. A third aspect we might be interested in is how our feature variables are interrelated.

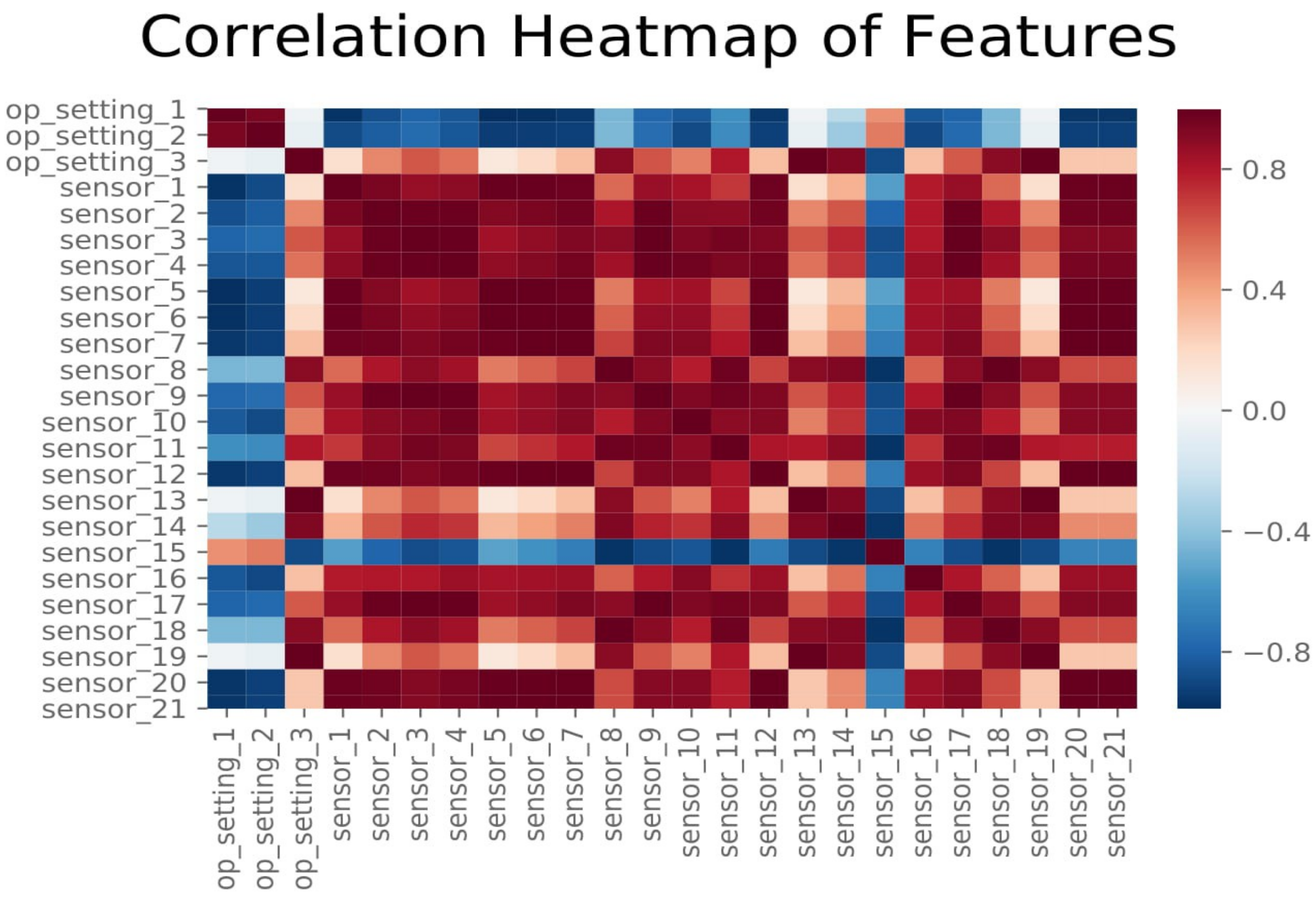
Code box 9:

```
# plot correlation heatmap of independent variables
indepvar_cmap(df=train_df)
```

Code box 9 calls the last function we have already imported from *tfed_descriptives*, *indepvar_cmap*. *indepvar_cmap* takes a data frame as input, selects all features with non-constant variance (all features in our case) and saves a correlation heatmap plot of all remaining features to the current working directory.

In figure 4 below we see a matrix where each cell represents the correlation between the variable in the cell's row and the variable in the cell's column. The darker the red colour, the closer the correlation in the cell is to +1, i.e. perfect positive correlation. The darker the blue colour, the closer the correlation in the cell is to -1, i.e. perfect negative correlation. White represents a correlation of zero. Logically, the main diagonal is dark red because the correlation of a variable with itself is +1. Interestingly, though, we see many other dark red cells in addition to those in the main diagonal. We also see many dark blue cells. This indicates that our data has a high level of multicollinearity. This suggests that some sensors might be organized in clusters, e.g. multiple sensors could measure the same quantity or groups of sensors could be placed in the same component. However, (almost) perfect multicollinearity leads to numerical instabilities during optimization in many machine learning algorithms. In order to remedy this, we will therefore also conduct feature selection in the data preprocessing step.

Figure 4: Correlation heatmap of features



2.5 Data exploration results

So now that we have explored our data thoroughly: What have we learned? What do we have to do in the data preprocessing step? Firstly, when analyzing the data's descriptive statistics, we have noted the different scales of the feature values. We therefore have to bring all variables to a common scale. Secondly, we should account for the various operating conditions. Thirdly, we are going to test some filter and smooting algorithms in order to correct for the sensor noise. Fourthly, we have noted the high level of multicollinearity among our feature variables. We therefore add a feature selection step to our data preprocessing agenda as well.

3. Data preprocessing

3.1 Data augmentation and scaling

In the previous chapter we got to know our data. Among other things, we have learned that the data is still in a pretty raw form and we therefore have to work through a few preprocessing steps before we can apply our model(s) to this data. This chapter covers everything that is needed so that in the end we have an estimation-ready data set.

Let's start with one of the most common questions in machine learning: Is there a way to get more data? Intuitively, since the subject of our analysis is practically a stochastic process, the data only provides a blurry description of the underlying true process. Adding relevant data points, i.e. adding relevant information, allows the model to get a clearer picture of the underlying process. Or a little more technically: Adding more data points is likely to reduce the risk of model overfitting during estimation and may therefore improve the model's prediction accuracy on unseen data.

So how can we get more data? If we knew the ground truth data generating process we could simply generate synthetic data. Unfortunately, though, we don't know this process. Yet there is still hope: From the last chapter's analysis we know that FD002, which is our current training data set, and FD004 are related. In fact, the only way they differ is that there is only one fault mode present in FD002 but two in FD004, where some trajectories of FD004 have the same fault mode as the FD002 trajectories. So here comes a hypothesis: If we could somehow label all trajectories in FD002 and FD004 with their fault mode, we could filter all FD004 trajectories with the same fault mode as in FD002. These observations could then be appended to our current training data set.

We have just described an unsupervised learning problem and - as often - there are multiple possible strategies to solve it, depending on the assumptions we are willing to make. In this project, the k-means clustering algorithm is applied to find the fault mode labels for all FD004 trajectories.

Excursus 1: k-means clustering

Basically, the algorithm works as follows: We are given a *d*-dimensional Euclidean feature space that contains all data points. Moreover, we are given some integer *k* which is the number of clusters we want to assign our data points to. Now, in a first step, the location of the cluster centers - or centroids - are initialized randomly somewhere in the feature space. Subsequently, each data point is assigned to the cluster whose centroid is closest to it. After the assignment of all points, the centroids are recalculated so that the sum of the distances between all points and their respective cluster center is minimized. These steps are then repeated until the algorithm converged, where convergence means that the sum of the distances between all points and their respective cluster center does not change anymore.

Formally, this algorithm can be described as follows: Let $\boldsymbol{x_i}$,with i = 1,...,N, be our data points that our d-dimensional feature space contains. Additionally, let $\boldsymbol{\mu_k}$, with k = 1,...,K be our vectors of cluster centroids. Furthermore, define a binary variable r_{ik} that evaulates to 1 if $\boldsymbol{x_i}$ is assigned to cluster k and 0 otherwise.

Now we can define a loss function L :

$$L \quad = \quad \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} \left\| \boldsymbol{x_i} - \boldsymbol{\mu_k} \right\|^2 \tag{E1.1}$$

where $\left\| \boldsymbol{x_i} - \boldsymbol{\mu_k} \right\|$ denotes the Euclidean distance between $\boldsymbol{x_i}$ and $\boldsymbol{\mu_k}$. Our goal is to minimize L. We try to accomplish this in two steps.

In the first step, we try to minimize L over r_{ik} , i.e.

$$r_{ik} \quad = \quad \begin{cases} 1 \text{ if } k = \underset{j}{arg\,min} \left\| \boldsymbol{x_i} - \boldsymbol{\mu_j} \right\|^2, \text{ where } j = 1,\dots,K \\ 0 \text{ otherwise} \end{cases} \tag{E1.2}$$

In the second step, we try to minimize L over the position of the cluster centroids.[1] This can be achieved by setting the first derivative of L with respect to $\boldsymbol{\mu_k}$ to zero:

$$\frac{\partial L}{\partial \mu_k} \quad = \quad 2\sum_{i=1}^{N} r_{ik} (\boldsymbol{x_i} - \boldsymbol{\mu_k}) \quad = \quad 0 \tag{E1.3}$$

1 Note that for each $\boldsymbol{\mu_k}$ we now only consider those $\boldsymbol{x_i}$ that have been assigned to cluster k and we hold this assignment fixed during optimization.

Solving (E1.3) for $\boldsymbol{\mu_k}$ yields:

$$\boldsymbol{\mu_k} \quad = \quad \sum_{i=1}^{N} r_{ik} \boldsymbol{x_i} \left(\sum_{i=1}^{N} r_{ik}\right)^{-1} \tag{E1.4}$$

From (E1.4) we can conclude that an optimal $\boldsymbol{\mu_k}$ is the mean of all $\boldsymbol{x_i}$ that are assigned to cluster *k*.

Steps (E1.2) - (E1.4) are repeated until L does not change anymore. As a result, for every $\boldsymbol{x_i}$ we have a cluster prediction $\hat{k}_i$.

In order to consistently find labels for all trajectories with k-means clustering, we need the following five assumptions:

A1: Our data is numeric.
We need this assumption since k-means clustering relies on the Euclidean metric as can be seen in (E1.1). Fortunately, from the previous chapter we already know that A1 holds.

A2: All features are on a common scale.
The choice of a variable's unit of measurement is almost always arbitrary: 1000 meters is identical to 1 kilometer; 0 degrees Celsius is identical to 273.15 Kelvin and so on. However, concepts like the Euclidean distance are solely based on raw numeric inputs: Changing the unit of measurement could change the raw numerical values, this led to a change of the locations of the points within the Euclidean space and this in turn changed the value of the resulting Euclidean distance measure. Rescaling all variables to a common scale helps to circumvent this inconsistency.

There are many ways to rescale variables. The choice of scaler always depends on the particular problem. In this project, standardization was chosen as rescaling method, i.e. the values of the variables were rescaled by subtracting the variable's mean from them and dividing them by the variable's standard deviation.

Note that this rescaling is also important for later optimization: Models that rely on gradient based optimization algorithms may not converge if the feature variables have not been brought to comparable scales. This can be attributed to the fact that the greater a variable's variance is, the more weight the optimization algorithm assigns to this variable. In extreme cases, this variable then dominates all others and the optimization algorithm only reaches a local minimum.

A3: The number of operating conditions in the training data set remains six after augmenting the FD002-based training data set by FD004-observations and the operating conditions are accounted for.

Following Pasa, Medeiros & Yoneyama (2019), we account for operating conditions by scaling for each operating condition separately: Let $\boldsymbol{X}_{,d}$ be the column vector of feature *d*. Further assume that for each operating condition *c* it holds that $c \in \{1,\ldots,C\}$ and that for each engine *i* at time *t* we have an operating condition cluster prediction $\hat{c}_{it}$. Then, the operating condition specific standardized column vector of feature *d*, $\widetilde{\boldsymbol{X}}_{,d}$, may be computed as:

$$\widetilde{\boldsymbol{X}}_{,d} \quad = \quad \sum_{k=1}^{C} \boldsymbol{\delta}_{\boldsymbol{k}} \odot \left(\frac{\boldsymbol{X}_{,d} - \overline{X}_{,dk}}{\sigma_{,dk}} \right) \tag{1}$$

with

$$\delta_{it,k} \quad = \quad \begin{cases} 1 \text{ if } k = \hat{c}_{it} \\ 0 \text{ otherwise} \end{cases} \tag{2}$$

However, how could we test the first part of A3? The predictions for the operating condition classes in FD002 should be the same regardless of whether the FD004-observations have already been added or not. At best, there would be a sensor whose values are only influenced by the operating conditions and the function between the set of operating conditions and the set of resulting sensor data is bijective. Such a sensor would be a perfect predictor of operating conditions. If the performance of the predictor sensor does not deteriorate after augmenting the data, we can assume that the first part of A3 holds.[2] Since sensor 5 exhibits such a bijective property, it is assumed in this project that it can serve as such a predictor sensor.

A4: The engines in FD002 and FD004 are structurally identical.

A5: Average cycle lengths in FD002 and FD004 do not differ.
We have not been given any information on A4 and A5. Therefore we have to continue by simply assuming them to hold.

Now we are ready to actually implement the data augmentation and scaling in Python. Lines 3 and 5 in code box 10 below act as a two layered flow control statement: Line 3 uses the system variables with which the master file was started to check whether data preprocessing has to be carried out at all. Since this step is to be introduced in this chapter, let's assume that this line is evaluated to *True*. Now, to see what happens in line 5, remember that we initially loaded all of the raw data sets into the *data* dictionary. In chapter 2, however, we decided to use FD002 as our data sets of interest and pulled them out of *data* for the sake of convenience. If the master file was triggered without the instruction to perform the steps from chapter 2, however, line 5 evaluates to true and the two FD002 data sets are pulled out of *data* in lines 7 and 8 and stored into the variables *train_df* and *test_df*.

Since we are going to append FD004 observations to our current FD002-based *train_df*, a data source identifier is added to *train_df* in line 9. Line 10 updates the unit identifiers accordingly. Lines 12 - 15 repeat lines 7, 9 and 10 for the FD004 training data set. Then, in line 16, the entire FD004 training data is appended to *train_df*.

However, the augmentation of our data has not yet been completed as 1) we don't know whether the number of distinct operating condition classes is still six in our training data, 2) our feature variables are still not brought to a common scale and 3) now those FD004-engines are also included that have a different failure mode than the engines we want to analyze.

2 We hard-coded the number of distinct operating condition classes to be six. But if there were actually more than six distinct operating condition classes present in the data after augmentation, the assignment between operating condition and sensor output could no longer be bijective.

Code box 10:

```python
### preprocess data
# check if mode passed as system argument requires preprocessing
if 'all' in sys.argv or 'preprocessing' in sys.argv:
    # create raw data df
    if not ('all' in sys.argv or 'descriptives' in sys.argv):
        # we first focus on tables FD002
        train_df = data['train_FD002']
        test_df = data['test_FD002']
    train_df['df'] = 'FD002'
    train_df['unit_id'] = train_df['df'] + '_' + train_df['unit'].astype(str)
    # add train_df_4 to train_df
    train_df_4 = data['train_FD004']
    train_df_4['df'] = 'FD004'
    train_df_4['unit_id'] = train_df_4['df'] + '_' \
                          + train_df_4['unit'].astype(str)
    train_df = train_df.append(train_df_4)

```

Code boxes 11 and 12 below address the first of these three points. Code box 11 uses two functions that have been imported from module *tfed_preprocessing.py*. *standardizer()* relies on the *sklearn.preprocessing* module *StandardScaler* and transforms a vector by centering it around zero and dividing it by its standard deviation. *kmeans_clustering()* relies on the *sklearn.cluster* module *KMeans* and offers a convenient implementation of (E1.1) - (E1.4). Note that we skipped lines 117 - 120 in *tfed.py* for now. We will come back to these lines in the next sub-section. All we currently need to know about these lines is that they generate a new feature variable *Ti*.

In line 2 of code box 11 we declare the columns based on which we want to get the operating condition classifications. Like all of our features, these three are not yet standardized and so their variances still vary considerably (as can be seen in Ap1 in the appendix, the standard deviation of *op_setting_1* is roughly 47.6 times that of *op_setting_2*). As discussed previously, this circumstance may bias class predictions. *standardizer()* is used to remedy this issue: It takes two arguments, the data frame and the list of columns (=variables) to standardize, and outputs the transformed (=standardized) columns as well as a *scaler* instance. *scaler* can be used later if the exact same transformation is to be applied to other data, i.e. if the same mean and standard deviation is to be used instead of the new data's mean and standard deviation. Hence we transform the three operating condition columns of *train_df* to their standardized versions in line 4 and repeat this transformation for *test_df* in line 6 with the values obtained from the transformation in line 4.

Code box 11:

```
[001] # cluster observations based on operating conditions
[002] cluster_cols=['op_setting_1', 'op_setting_2', 'op_setting_3']
[003] # standardize op-settings columns for train and test dfs
[004] train_df[cluster_cols], scaler = standardizer(data=train_df.copy(),
[005]         col_list=cluster_cols)
[006] test_df[cluster_cols] = scaler.transform(test_df[cluster_cols])
[007] # add op condition cluster prediction to train and test dfs
[008] train_df, kmeans_fit = kmeans_clustering(train_df.copy(),
[009]         cluster_cols=cluster_cols)
[010] test_df['op_condition'] = kmeans_fit.predict(test_df[cluster_cols])
[011]
```

Similar to *standardizer()*, *kmeans_clustering()* takes two arguments: A data frame and a list of columns that are to be used for clustering. The output of *kmeans_clustering()* is a tuple of length two: The first element of this tuple is the data frame that was passed to the function, but now with an additional column, *op_condition*, that contains the operating condition class predictions. The second element is a *kmeans_fit* instance which is the fitted k-means estimator. Hence in line 8 of code box 11, we add a column with the predicted operating condition classes to *train_df* and store the associated k-means estimator. In line 10 this estimator is then used to get the class predictions for *test_df*.

Let's now check whether our clustering approach worked. In a previous data analysis sensor 5 appeared to perfectly predict the operating condition classes for FD002 data. If the prediction performance of sensor 5 has not deteriorated after appending the FD004 data, we might conclude that FD002 and FD004 indeed contain the same set of operating conditions.

To see what happens in code box 12, first remember that we still only need to have six operating conditions. Therefore, we iterate through the integers from 1 to 6 and check whether the mean of sensor 5 is almost the same for both sub data sets; *almost the same* here means equal except for rounding errors.

Code box 12:

```python
# test whether op conditions are indeed the same
# sensor 5 is used to test this because it perfectly predicts op_condition
for i in range(6):
    print(math.isclose(\
          np.mean(train_df['sensor_5'].loc[(train_df['df']=='FD002') & \
                 (train_df['op_condition']==i)]),
          np.mean(train_df['sensor_5'].loc[(train_df['df']=='FD004') & \
                 (train_df['op_condition']==i)])))

```

Running the code of box 12, we see that it prints *True* for all six operating conditions. We can therefore conclude that our clustering approach worked and the number of distinct operating conditions in *train_df* indeed remained six.

So let's continue with point 2 of our current to do list and rescale the remaining feature variables. In lines 2 and 3 of Code box 13 below, we put all feature variables that still need to be rescaled in a list *scalevars_list*. Then we call the function *rescale_data()* which we loaded previously from *tfed_preprocessing.py* and apply it to our training and test data sets. *rescale_data()* is basically just an implementation of (1) and (2). It iterates through the set of operating conditions found in *data_train*. Then, for each distinct operating condition, it repeats the following tasks: It calls *standardizer()* with the arguments *data_train* and *scalevars_list*. As a result, a subset of *data_train* is formed, which only contains observations of the current operating condition. Subsequently, all variables in *scalevars_list* are standardized independently of the values they have under the other operating conditions. In each iteration the transformation applied to the current subset of *data_train* is reapplied to the corresponding subset in *data_test*. Note that this operating condition specific standardization of the features means that now the different operating conditions are accounted for.

Code box 13:

```python
# define list of variables that are to be rescaled
scalevars_list = [x for x in train_df.columns if x.startswith('sensor') \
                  or x.startswith('Ti')]
# call rescaling management function and get standardized features
data_train, data_test = rescale_data(data_train=train_df.copy(),
                                     data_test=test_df.copy(),
                                     scalevars=scalevars_list)

```

Now there's only one thing left on our current to do list: We have to drop all those engines from *data_train* that have a different failure mode than the FD002 observations. To achive this, we first need to identify each engine's last operating cycle. However, this information is only implicitly contained in our current data by *cycle*. Essentially, *cycle* just counts the number of observations per unit upwards. Since the total number of observations varies across units, the information contained in *cycle* alone can't serve as an indicator for failure. Therefore, an auxiliary variable *max_cycle* is created and added to *data_train* in lines 2 - 4 of code box 14 below. For each unit, *max_cycle* stores the value of *cycle* in the last operating cycle. Then, our new target variable *cycles_to_failure* can easily be created in lines 5 and 6: For each observation, *cycle* just needs to be subtracted from *max_cycle*. Since the training data contain the full degradation trajectories of all units, for each unit we have exactly one observation where *cycles_to_failure* equals to zero which corresponds to the unit's last operating cycle before failure.

Based on the sensor values in the last operating cycle, we can now divide the engines into two clusters, where each cluster represents a failure mode. Ideally, all FD002 observations are assigned to one cluster, and while some of the FD004 observations are assigned to the same cluster, some FD004 observations are assigned to the other cluster. Those observations that are assigned to the other cluster can then be dropped from the training data set. Line 9 of code box 14 calls *target_clusterer()* which has been imported from *tfed_preprocessing.py*. *target_clusterer()* is basically just a wrapper function for *KMeans* from the *sklearn.cluster* module: It takes a data frame and the columns that are to be used for clustering as input, creates a subset of the data frame where each row corresponds to an engine's last observation and then calls *KMeans* on the resulting data frame. It returns a data frame with two columns: The unit identifier and a failure mode prediction *target_cluster*.

Code box 14:

```
[001] # create target variable for training data
[002] data_train =\
[003] data_train.assign(max_cycle=data_train.groupby('unit_id')['cycle'].\
[004]                   transform('max'))
[005] data_train['cycles_to_failure'] = data_train['max_cycle'] \
[006]                                   - data_train['cycle']
[007]
[008] # get all observations with fd002 fault mode (including those from fd004)
[009] dtrain_targeted = target_clusterer(data_train.copy(), filter_list)
[010] fd002_target_cluster = np.mean(dtrain_targeted['target_cluster'].\
[011]                                loc[dtrain_targeted['unit_id'].str.\
[012]                                    contains('FD002')])
[013] dtrain_targeted = dtrain_targeted[dtrain_targeted['target_cluster']\
[014]                                   ==fd002_target_cluster]
[015] data_train = data_train[data_train['unit_id'].\
[016]                         isin(dtrain_targeted['unit_id'])]
[017]
```

Consistent clustering would assign all FD002 observations to the same failure mode cluster. Examining *dtrain_targeted* from line 9, we can see that this is indeed the case; all FD002 observations are labeled with the same cluster integer. Therefore, we can simply store the mean of all FD002 predictions as the cluster we are interested in, which is implemented in lines 10 - 12. Lines 13 and 14 then declare that only those units are to be kept that are assigned to the cluster to which all FD002 observations were assigned. The resulting list is stored in *dtrain_targeted*. Finally, lines 15 and 16 subset *data_train* so that it only contains those units that are also included in *dtrain_targeted*.

And that's it. We have increased the number of observed trajectories in our training data set by about 57% and all our features have been brought to a common scale. By the way: With *cycles_to_failure*, we even created the target variable for this project. But more on this in the next sub-section.

3.2 Feature engineering

Adding more observations is not all we can do to increase the information content of our training data set. We can also try to increase the information content per observation for our later model. This can be achieved by engineering (new) features from existing variables.

In principle, creating new variables based on existing ones is nothing new to us anymore. For example, we just created our later model's target variable *cycles_to_failure* based on *cycle* (see Code box 14). However, if we stopped at this point, some information in our data might be left untapped. Imagine a situation where we only see random snapshots of the trajectories, say only eight consecutive periods. If we disregarded the information in *cycle* we wouldn't know whether the snapshot was taken from a relatively young, a middle aged or relatively old engine. Some would argue, however, that an engine's age could be an important piece of information in run to failure analysis. But here comes the dilemma: On the one hand, all of our feature variables have to be brought to a common scale, on the other hand, we will later need a non-transformed *cycle* for our sampling method. So let's come back to lines 117 - 120 of *tfed.py*, which we skipped above. As can be seen in code box 15 below, we copy *cycle* and store this copy as a new variable, *Ti*, which we can preprocess and later use as a feature.

Code box 15:

```
# add column Ti which is the cycles information but used as a feature
train_df['Ti'] = train_df['cycle']
test_df['Ti'] = test_df['cycle']

```

Admittedly, the identity transformation in code box 15 doesn't really knock anyone's socks off. So let's quickly move on.

Our later model requires a slight variation of A1 which we defined in sub-section 3.1: All features should be numeric. In Code box 11, however, we have created the variable *op_condition*, which does not meet this assumption. *op_condition* is a categorical variable in disguise: Although it is integer-coded, its values have no natural ordering. Each value represents a distinct operating condition, like operation at sea level. So "*op_condition* equal to two" does not imply "twice as much as *op_condition* equal to one". Since we want to provide the information stored in *op_condition* as input to our later model, we need to find an actual numerical representation of it.

One of the most common approaches to solving this problem is one hot encoding. With one hot encoding, for each distinct value of a categorical variable a new binary variable is defined. So for each realization of the categorical variable, only the corresponding binary variable "switches on" and all others remain "switched off".

As can be seen in code box 16 below, the *pandas* package offers a convenient way to one hot encode categorical variables. In line 2 the data frame column in which *op_condition* is stored is passed to *get_dummies()*. Additionally, the string *op_condition* is passed as the prefix for the names of the new binary variables. The resulting data frame *one_hots* has the same number of rows as *data_train* and contains one binary valued column for each distinct operating condition, i.e. *op_condition_0*, *op_condition_1*, ..., *op_condition_5*. In line 4, *one_hots* is added to *data_train* along axis one, i.e. the *one_hots* data is added column-wise. Since now the new set of binary variables renders *op_condition* itself redundant, *op_condition* is deleted from *data_train* in line 5. Lines 6 - 9 simply repeat this one hot encoding procedure for the test data set *data_test*.

Code box 16:

```
[001] # one hot encode operational conditions
[002] one_hots = pd.get_dummies(data_train['op_condition'],
[003]                           prefix='op_condition')
[004] data_train = pd.concat([data_train,one_hots],axis=1)
[005] data_train.drop(['op_condition'],axis=1, inplace=True)
[006] one_hots = pd.get_dummies(data_test['op_condition'],
[007]                           prefix='op_condition')
[008] data_test = pd.concat([data_test,one_hots],axis=1)
[009] data_test.drop(['op_condition'],axis=1, inplace=True)
[010]
```

The readme file that comes with the data informs us that there is variation in the initial wear of the engines. Even though none of the engines already starts with a fault condition, initial wear might have an effect on the engines' degradation trajectories (Saxena et al., 2008). Usually, if we had domain knowledge in the field of turbofan engines, we would add initial flow and efficiency values of critical components to our list of features (Saxena et al., 2008). Unfortunately, though, we lack exactly this knowledge.

But again: We are not lost. We can circumvent this issue in a purely data-driven way: For each engine and for each sensor, we simply store the average values over the first periods as a new feature variable. Later, a feature selection algorithm could be applied to keep only the most relevant initial sensor means. Note that the sensor values are averaged over the first *x* periods in order to reduce the effect of sensor noise.

Code box 17:

```
[001] # define remaining sensors list
[002] sensors_list = [x for x in data_train.columns if x.startswith('sensor')]
[003]
[004] # add each variables mean over first 11 cycles as initial health control
[005] data_train = var_init_mean_adder(df=data_train.copy(),
[006]                                 columns=sensors_list)
[007] data_test = var_init_mean_adder(df=data_test.copy(),
[008]                                columns=sensors_list)
[009]
```

In line 2 of code box 17, we define the list of sensors. Then, the function *var_init_mean_adder*, which we imported previously from *tfed_preprocessing.py*, is called for our training and test data. To summarize *var_init_mean_adder*: It takes a data frame, *df*, and a list of columns (our just defined list of sensors), *columns*, as parameters. Then it iterates over two nested loops where the outer loop is the iteration over all sensors and the inner loop is the iteration over all unique unit ids. For each unit and each sensor, it computes the average over the first eleven observations. Finally, for each sensor it adds a column with the vectors of these unit-specific averages to *df* and returns the resulting data frame.

3.3 Denoising

During data exploration, we have noted that our sensor data is subject to noise. This sub-section is intended to analyze possible solutions to this problem and show their implementation in Python.

Excursus 2: Smoothing and filter methods

Suppose a process that generates a sequence of data $\boldsymbol{x_t}$, where t = 1,2,..., T. Suppose further we observe this process; however we actually only observe $\boldsymbol{x^{\sim}_t}$, where $\boldsymbol{x^{\sim}_t = x_t + \varepsilon}$ with $\boldsymbol{\varepsilon}$ being a normally distributed random variable. In our case, one possible source of $\boldsymbol{\varepsilon}$ could be measurement noise. We assume that if we had a "good enough" estimate of $\boldsymbol{x_t}$, $\boldsymbol{\hat{x}_t}$, replacing $\boldsymbol{x^{\sim}_t}$ with $\boldsymbol{\hat{x}_t}$ in our feature set would increase our model's efficiency.

Excursus 2.1: Savitzky-Golay Filter

One possible way to get an estimate of $\boldsymbol{x_t}$ is the filter method proposed by Savitzky and Golay (1964). Intuitively, the idea of this filter is to replace each $x^{\sim}_t$ with a weighted average of all observations within a certain range around $x^{\sim}_t$. Note that if we replaced each $x^{\sim}_t$ with a simple average of all observations within a certain range around it, we would apply the moving average technique. However, the Savitzky-Golay filter is often the preferred choice of filtering since it preserves data attributes that other filter techniques like moving average do not.

So how is this favourable weighting achieved? First we reorganize our given data into the set of sequences $(\boldsymbol{x^{\sim}_t}, \boldsymbol{d_t})$, where, d_1 = 1, d_2 = 2, ..., d_T = T. Next, we choose a window size m, where m is a positive and odd integer. Then, for each $x^{\sim}_t$, we only consider the interval $\left[x^{\sim}_{t-(m-1)/2}, x^{\sim}_{t+(m-1)/2}\right]$ and change $\boldsymbol{d}$ to $\boldsymbol{z}$, where $\boldsymbol{z} = \boldsymbol{d} - d_t$. Due to the evenly spaced nature of $\boldsymbol{d}$ it holds that for any $\left[x^{\sim}_{t-(m-1)/2}, x^{\sim}_{t+(m-1)/2}\right]$, $\boldsymbol{z} = -(m-1)/2, \ldots, -2, -1, 0, 1, 2, \ldots, (m-1)/2$.

Next, imagine we want to fit a low degree polynomial to all observations that are in $\left[x^{\sim}_{t-(m-1)/2}, x^{\sim}_{t+(m-1)/2}\right]$. Let k denote the degree of our polynomial and let our polynomial be defined by $X = \beta_0 + \beta_1 \boldsymbol{z}^1 + \ldots + \beta_k \boldsymbol{z}^k$. We then obtain the unknown parameters β_0, ..., β_k by applying the least squares method:

$$\boldsymbol{\hat{\beta}_{SG}} = (\boldsymbol{J'J})^{-1}\boldsymbol{J'x^{\sim}_t}, \qquad \text{(E2.1.1)}$$

where row i of matrix $\boldsymbol{J}$ is equal to 1, z_i, z_i^2, ..., z_i^k and $\boldsymbol{J'}$ indicates the transpose of matrix $\boldsymbol{J}$. Since $\boldsymbol{z}$ does not change as we slide $\left[x^{\sim}_{t-(m-1)/2}, x^{\sim}_{t+(m-1)/2}\right]$ along $\boldsymbol{x^{\sim}_t}$, $(\boldsymbol{J'J})^{-1}\boldsymbol{J'}$ does not change either. Consequently, it suffices to compute $(\boldsymbol{J}'\boldsymbol{J})^{-1}\boldsymbol{J}'$ only once and then apply convolution. Therefore, let $\boldsymbol{c}$ denote the first row of $(\boldsymbol{J}'\boldsymbol{J})^{-1}\boldsymbol{J}'$. Then our estimated x_t is defined by:

$$\hat{x}_t = \sum_{i=-(m-1)/2}^{m-1/2} c_i x\tilde{}_{t+i} \qquad \text{(E2.1.2)}$$

Closely related to the method proposed by Savitzky and Golay is *locally weighted scatterplot smoothing* (LWSS)[3]. LWSS can be viewed as a generalization of the Savitzky-Golay filter since it replaces (E2.1.1) with its weighted least squares version:

$$\hat{\boldsymbol{\beta}}_{LWSS} = (\boldsymbol{J}'\boldsymbol{W}\boldsymbol{J})^{-1}\boldsymbol{J}'\boldsymbol{W}\boldsymbol{x}\tilde{}_{\boldsymbol{t}} \qquad \text{(E2.1.3)}$$

where the entries of weight matrix $\boldsymbol{W}$ are commonly (but not necessarily) defined by:

$$W_{ij} = \left(1-\left|\frac{d_j-d_i}{s_i}\right|^3\right)^3 \qquad \text{(E2.1.4)}$$

where s_i is the maximum distance between between d_i and any d_j associated with the current interval $\left[x\tilde{}_{t-(m-1)/2}, x\tilde{}_{t+(m-1)/2}\right]$. From (E2.1.4) it can be seen that the weight of a data point $x\tilde{}_j$ decreases with its distance to $x\tilde{}_i$.

The generalization aspect results here from the fact that LWSS also accomodates situations in which the data points are not equally spaced. Variations in distance are accounted for by changes in $\boldsymbol{W}$. Each time the weights are recalculated, a new $\hat{\boldsymbol{\beta}}_{LWSS}$ is to be computed. However, in this project *cycles* is used as $\boldsymbol{d}_t$ and so for all *i* and *j* it holds that:

$$d_i - d_{i-1} = d_{i+j} - d_{i+j-1},$$

3 Depending on the order of the polynomial and the number of predictor variables, locally weighted scatterplot smoothing is mostly known in the literature as either LOWESS or LOESS. In practice, however, these two terms are often used synonymously. In order to avoid potential naming discussions, locally weighted scatterplot smoothing as defined in this work is henceforth abbreviated LWSS.

i.e. $\boldsymbol{d}_t$ is equally spaced and hence weights never change. Even though this generalization is not required in this work, it might nonetheless be of interest to see whether the mere reweighting of the data, as in (E2.1.4), leads to an improved filter performance. So let's assume evenly spaced data points again. Due to the even spacing it again holds that weights don't change. Hence we only need to compute $\boldsymbol{W}$ once and then we can apply convolution again, now only with weight-adjusted convolution coefficients:

$$\hat{x}_t \quad = \quad \sum_{i=-(m-1)/2}^{m-1/2} \tau_i x^{\sim}_{t+i} \qquad \text{(E2.1.5)}$$

where τ is the first row of $(\boldsymbol{J}'\boldsymbol{W}\boldsymbol{J})^{-1}\boldsymbol{J}'\boldsymbol{W}$.

Excursus 2.2: Kalman Filter

If you're working on a signal processing problem with sensor data from NASA, most colleagues will probably nod and ask (in a friendly advisory manner), "Have you tried the Kalman filter yet?" So let's better do that now.

Again, we assume that we only observe $\boldsymbol{x}_t^{\sim}$ instead of $\boldsymbol{x}_t$. However, instead of computing a weighted average over the interval $\left[x^{\sim}_{t-(m-1)/2}, x^{\sim}_{t+(m-1)/2}\right]$ to get $\hat{x}_t$, Kalman (1960) proposed to combine a prediction of x_t which is based on $\hat{x}_{t-1}$, $\hat{x}_t^{-}$, with a function of $x_t^{\sim}$ to get $\hat{x}_t$. For every *t*, this algorithm is then repeated recursively.

At least two questions arise at this point:
1) How do we get $\hat{x}_t^{-}$?
2) What is the function of $x_t^{\sim}$ which we want to combine with $\hat{x}_t^{-}$?

First, we have to assume that we can model the underlying process as:

$$\boldsymbol{x}_t \quad = \quad \boldsymbol{F}\boldsymbol{x}_{t-1} + \boldsymbol{w}_t, \qquad \text{(E2.2.1)}$$

and

$$\boldsymbol{x}_t^{\sim} \quad = \quad \boldsymbol{H}\boldsymbol{x}_t + \boldsymbol{v}_t, \qquad \text{(E2.2.2)}$$

where, in the Kalman filter context, $\boldsymbol{x}_t$ is called the *state* and is allowed to be a vector of length *n*, $\boldsymbol{F}$ is the $n \times n$ matrix that contains the practitioner-supplied information on how the state evolves from the previous to the current (time-)index, hence $\boldsymbol{F}$ is often called the *transition matrix*, and $\boldsymbol{w}_t$ is the $n \times 1$ process noise vector that follows $\boldsymbol{w}_t \sim N(0, \boldsymbol{Q}_t)$, where $\boldsymbol{Q}_t$ is a known $n \times n$ covariance matrix.

However, as mentioned above, we only observe $\boldsymbol{x}_t^{\sim}$. Therefore, we need (E2.2.2) which provides the mapping from the true state to the observed state, where $\boldsymbol{H}$ is the $m \times n$ matrix containing the practitioner-supplied information on the association between the true state and the observed state and $\boldsymbol{v}_t$ is the error term of the measurement process and is assumed $\boldsymbol{v}_t \sim N(0, \boldsymbol{R}_t)$, where $\boldsymbol{R}_t$ is a known $m \times m$ covariance matrix.

Admittedly, that was quite a bit of theory. But this immediately helps us to answer question 1: We only have to replace $\boldsymbol{x}_{t-1}$, which we don't observe, with our "best guess" of it, $\hat{\boldsymbol{x}}_{t-1}$, in (E2.2.1):

$$\hat{\boldsymbol{x}}_t^- = \boldsymbol{F}\hat{\boldsymbol{x}}_{t-1} \qquad \text{(E2.2.3)}$$

Because the underlying process is stochastic, $\hat{\boldsymbol{x}}_t^-$ will (almost certainly) be not equal to $\boldsymbol{x}_t$, i.e. there will be some residual error $\boldsymbol{\epsilon}_t \equiv \boldsymbol{x}_t - \hat{\boldsymbol{x}}_t^-$. Luckily, though, we still have our measurement $\boldsymbol{x}_t^{\sim}$. Let us now assume that our sensors work (to some extent) and there is at least some useful information in $\boldsymbol{x}_t^{\sim}$. We can then incorporate $\boldsymbol{x}_t^{\sim}$ in such a way that our residual error is minimized. Therefore, let

$$\hat{\boldsymbol{x}}_t = \hat{\boldsymbol{x}}_t^- + \boldsymbol{K}_t(\boldsymbol{x}_t^{\sim} - \hat{\boldsymbol{x}}_t^-) \qquad \text{(E2.2.4)}$$

where $\boldsymbol{K}_t$ is an $n \times m$ matrix that gives weights to $\hat{x}_t^-$ and $x_t^{\sim}$ so that the residual error is minimized. To illustrate (E2.2.4): The greater the uncertainty in our measurements, the more weight would be put to $\hat{x}_t^-$ and hence the entries in $\boldsymbol{K}_t$ would approach zero. Conversely, the less noise there would be in the measurement, the more "we could trust" the measurement and hence the entries in $\boldsymbol{K}_t$ would approach one.

As we can see, a crucial part of the "Kalman filter magic" lies in $\boldsymbol{K}_t$. Therefore we will now take a closer look at it. Let our objective function be the mean squared prediction error which we want to minimize:

$$l_t \quad = \quad E(\boldsymbol{e}_t^2) \quad = \quad E[(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t)^2] \qquad \text{(E2.2.5)}$$

The error covariance matrix is then given by:

$$\boldsymbol{P}_t \quad = \quad E(\boldsymbol{e}_t \boldsymbol{e}_t{}') \quad = \quad E\left[(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t)(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t)'\right] \qquad \text{(E2.2.6)}$$

Note that the main diagonal of $\boldsymbol{P}_t$ contains the mean squared errors. Therefore, l_t is minimized if the sum of the main diagonal of $\boldsymbol{P}_t$ is minimized. The sum of the main diagonal of some matrix ***M*** is often referred to as the trace of ***M***, *tr(**M**)*. In our case, we want to minimize *tr(* $\boldsymbol{P}_t$ *)* with respect to $\boldsymbol{K}_t$:

$$\frac{\partial\, tr(\boldsymbol{P}_t)}{\partial \boldsymbol{K}_t} \quad = \quad 0 \qquad \text{(E2.2.7)}$$

To achieve (E2.2.7), we first substitute (E2.2.4) into (E2.2.6):

$$\boldsymbol{P}_t \quad = \quad E\left[(\boldsymbol{x}_t - (\hat{\boldsymbol{x}}_t^- + \boldsymbol{K}_t(x_t^{\sim} - \boldsymbol{H}\hat{\boldsymbol{x}}_t^-)))(\boldsymbol{x}_t - (\hat{\boldsymbol{x}}_t^- + \boldsymbol{K}_t(x_t^{\sim} - \boldsymbol{H}\hat{\boldsymbol{x}}_t^-)))'\right] \qquad \text{(E2.2.8)}$$

Next, substituting (E2.2.2) into (E2.2.8) yields:

$$\boldsymbol{P}_t \quad = \quad E\left[(\boldsymbol{x}_t - (\hat{\boldsymbol{x}}_t^- + \boldsymbol{K}_t(\boldsymbol{H}\boldsymbol{x}_t + \boldsymbol{v}_t - \boldsymbol{H}\hat{\boldsymbol{x}}_t^-)))(\boldsymbol{x}_t - (\hat{\boldsymbol{x}}_t^- + \boldsymbol{K}_t(\boldsymbol{H}\boldsymbol{x}_t + \boldsymbol{v}_t - \boldsymbol{H}\hat{\boldsymbol{x}}_t^-)))'\right] \qquad \text{(E2.2.9)}$$

Rearranging (E2.2.9) then yields:

$$\boldsymbol{P}_t \quad = \quad E\left[((\boldsymbol{I} - \boldsymbol{K}_t\boldsymbol{H})(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t^-) - \boldsymbol{K}_t\boldsymbol{v}_t)((\boldsymbol{I} - \boldsymbol{K}_t\boldsymbol{H})(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t^-) - \boldsymbol{K}_t\boldsymbol{v}_t)'\right] \qquad \text{(E2.2.10)}$$

where ***I*** is the $n \times n$ identity matrix. Now, for the sake of readability, let $(\boldsymbol{I} - \boldsymbol{K}_t\boldsymbol{H})(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t^-) = \boldsymbol{s}_t$. Then

$$\begin{aligned} \boldsymbol{P}_t &= E\left[(\boldsymbol{s}_t - \boldsymbol{K}_t \boldsymbol{v}_t)(\boldsymbol{s}_t - \boldsymbol{K}_t \boldsymbol{v}_t)'\right] \\ &= E\left[(\boldsymbol{s}_t \boldsymbol{s}_t') - 2\boldsymbol{K}_t \boldsymbol{v}_t \boldsymbol{s}_t + ((\boldsymbol{K}_t \boldsymbol{v}_t)(\boldsymbol{K}_t \boldsymbol{v}_t)')\right] \end{aligned} \qquad \text{(E2.2.11)}$$

Since, generally, the expectation of a sum of random variables equals the sum of their expectations, we can rewrite (E2.2.11) to:

$$\boldsymbol{P}_t = E\left[(\boldsymbol{s}_t \boldsymbol{s}_t')\right] - E\left[2\boldsymbol{K}_t \boldsymbol{v}_t \boldsymbol{s}_t\right] + E\left[(\boldsymbol{K}_t \boldsymbol{v}_t)(\boldsymbol{K}_t \boldsymbol{v}_t)'\right] \qquad \text{(E2.2.12)}$$

Note that the second expectation in (E2.2.12) is equal to zero since $(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t^-)$ and $\boldsymbol{v}_t$ are uncorrelated. Therefore, dropping this term, expanding $\boldsymbol{s}_t$ and some minor rearranging yields:

$$\boldsymbol{P}_t = (\boldsymbol{I} - \boldsymbol{K}_t \boldsymbol{H}) E\left[(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t^-)(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t^-)'\right] (\boldsymbol{I} - \boldsymbol{K}_t \boldsymbol{H})' + \boldsymbol{K}_t E\left[\boldsymbol{v}_t \boldsymbol{v}_t'\right] \boldsymbol{K}_t' \qquad \text{(E2.2.13)}$$

We are almost there. Let's use the fact that $\boldsymbol{R}_t = E\left[\boldsymbol{v}_t \boldsymbol{v}_t'\right]$ and further denote $E\left[(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t^-)(\boldsymbol{x}_t - \hat{\boldsymbol{x}}_t^-)'\right]$, the error covariance of the a priori estimate, $\boldsymbol{P}_t^-$. Then (E2.2.13) can be simplified to:

$$\boldsymbol{P}_t = (\boldsymbol{I} - \boldsymbol{K}_t \boldsymbol{H}) \boldsymbol{P}_t^- (\boldsymbol{I} - \boldsymbol{K}_t \boldsymbol{H})' + \boldsymbol{K}_t \boldsymbol{R}_t \boldsymbol{K}_t' \qquad \text{(E2.2.14)}$$

Now we can easily differentiate $\boldsymbol{P}_t$ with respect to $\boldsymbol{K}_t$, which is just what we wanted in (E2.2.7):

$$\frac{\partial\, tr(\boldsymbol{P}_t)}{\partial \boldsymbol{K}_t} = -2(\boldsymbol{H}\boldsymbol{P}_t^-)' + 2\boldsymbol{K}_t(\boldsymbol{H}\boldsymbol{P}_t^- \boldsymbol{H}' + \boldsymbol{R}_t) \qquad \text{(E2.2.15)}$$

Setting to zero and solving for $\boldsymbol{K}_t$ finally yields:

$$\boldsymbol{K}_t = \frac{\boldsymbol{P}_t^- \boldsymbol{H}'}{(\boldsymbol{H}\boldsymbol{P}_t^- \boldsymbol{H}' + \boldsymbol{R}_t)} \qquad \text{(E2.2.16)}$$

In theory, we are now ready to implement the filter in Python. In practice, however, we will quickly find that we have so far assumed away a non-trivial aspect: To implement the filter, the practitioner needs detailed process knowledge. $\boldsymbol{F}$, $\boldsymbol{H}$ and even the noise covariance matrices $\boldsymbol{Q}_t$ and $\boldsymbol{R}_t$ are to be supplied by the practitioner. As long as these practitioner-supplied values deviate from the actual values, the performance of the filter is not optimal. Since such detailed process knowledge is not available in most cases, though, these parameters, especially $\boldsymbol{Q}_t$ and $\boldsymbol{R}_t$, are often viewed as hyperparameters over which the performance of the filter is optimized. And that's not all: Another caveat is that (E2.2.1) - (E2.2.16) do not control for nonlinearities. If, e.g., damage increases exponentially with time, the Kalman filter would systematically underestimate the actual accumulated damage and therefore overestimate RUL during the cylces shortly before failure.

Before we finally come to look at the implementation of the denoising algorithms in Python, we have to briefly discuss multiprocessing. Up to this point only one instance of our program is run. The commands in our code are executed serially, i.e. instruction *x+1* is only executed when instruction *x* finished. However, the following situation often occurs: A problem can be broken down into several sub-problems, with the results of the sub-problems not depending on the results of the other sub-problems. If the CPU computing power is decisive for the execution time of the entire problem and not external processes such as waiting for user input, the runtime of the program can be reduced significantly by distributing the processing of the sub-problems over multiple memory locations. Let's say instruction *x* is calling some function *func()* with argument *a1* and instruction *x+1* is calling *func()* with argument *a2* and the output of *func(a2)* does not depend on the output of *func(a1)*. With multiprocessing, we could then start two instances of our program (i.e. we now run two processes) and solve *func(a1)* and *func(a2)* concurrently.

But how does this help us for our current project? We can formulate the problem of denoising the sensor time series with our various denoising algorithms in such a way that it fits exactly into the situation just described: We define a function *denoise_data()*, which is supposed to solve our fundamental problem of denoising the sensor time series. Among others, *denoise_data()* expects a string argument that specifies which denoising algorithm should be applied. Since the results of the algorithms do not depend on the results of the others and since denoising the sensor time series is a pure number crunching task, we can expect a considerable speed-up of the execution by running all denoising algorithms in parallel.

In Python, running multiple instances of a program in parallel can conveniently be implemented with the *multiprocessing* module. Code box 18 below shows how *multiprocessing* is used to concurrently run all denoising algorithms: Line 4 of Code box 18 defines a list that contains the names of the algorithms as strings. In addition to the algorithms' names, this list also contains *'none'*, which will later serve as an indicator that no denoising should be carried out. Now, the idea is that each denoising algorithm will be assigned its own process. To achieve this, a list *processes* is first initialized in line 7. Subsequently, in lines 8 - 14, a for-loop is used to start all four processes. *multiprocessing spawns* processes by first creating a *Process* object. Note that we imported *Process* as *process* from *multiprocessing* at the beginning of the script. In line 12 the *process* instance *p* is generated with the arguments *target*, which declares which function or task is to be carried out, and *kwargs* which are the keyworded arguments of the *target* function. As can be seen in lines 9 - 11, in the *i*th iteration of the for-loop the *i*th element of *denoising_algos* is passed as an argument, *dm*, to the *target* function. In line 14 the process of the current denoising algorithm is started by calling the *start()*-method of *p*. After calling the *start()*-method of instance *p* from iteration *x*, the next iteration is executed immediately, even if the *x*th iteration is not yet finished. Each *p* is appended to the *processes* list in line 13. This enables us to define another for-loop in lines 15 and 16. This loop now iterates over all *p*s in *processes* and calls the *join()*-method of each *p*. Essentially, calling the *join()*-method means "wait until the process has finished". Therefore, once the last process has finished, our program moves on in the usual sequential way.

Code box 18:

```
[001] # denoising
[002] # choose denoising algorithms
[004] denoising_algos = ['none', 'savgol', 'lwss', 'kalman']
[005] # multiprocessing: start a process and call denoise_data function for each
[006] # denoising algorithm
[007] processes = []
[008] for i in denoising_algos:
[009]     prc_kwargs = {'data_train':data_train.copy(),
[010]                   'data_test':data_test.copy(),
[011]                   'sensors_list':sensors_list, 'dm':i}
[012]     p = process(target=denoise_data, kwargs=prc_kwargs)
[013]     processes.append(p)
[014]     p.start()
[015] for prc in processes:
[016]     prc.join()
[017]
```

Now we have everything together to finally take a closer look at the implementation of the filter methods. As can be seen in Code box 18, the target function that is parallelized is *denoise_data()*. *denoise_data()* manages two sub-tasks: Firstly, if the denoising method passed to it is either *'savgol'*, *'lwss'* or *'kalman'*, it calls the sub-function *signal_smoother()*. Secondly, it catches the filtered time series returned by *signal_smoother()* and stores a plot of an exemplary filtered time series to the current working directory. If *denoise_data()* receives *'none'* instead, it simply skips these tasks.

So let's go down one level of abstraction and take a look at *signal_smoother()*. *signal_smoother()* also resides in the *tfed_preprocessing* module. It expects a data frame, a group identifier, i.e. one column that specifies the engines' ids, a list of sensors to be smoothed/filtered and an indicator of the filter method to be used. Regardless of which filter method indicator was passed to *signal_smoother()*, it first executes a nested for loop where the outer loop defines the iteration over the sensor time series and the inner loop defines the iteration over the engine ids.

If the filter method passed to *signal_smoother()* is either *'savgol'*, which is short for "Savitzky-Golay", or *'lwss'*, which is the weighted Savitzky-Golay filter version of locally weighted scatterplot smoothing, *signal_smoother()* calls the *savgol()* function from *tfed_savgol.py*. The function definition of *savgol()* is shown in Code box 19 below.

As can be seen in line 1 of Code box 19, *savgol()* expects at least three parameters: *y* is the time series selected by the nested loop of *signal_smoother()*, *window_size* is the length of the interval around each point to be smoothed (see m in Excursus 2.1) and *poly_order* is the degree of the polynomial that is fit to the points within this interval (see k in Excursus 2.1). Unless otherwise specified by the user, *savgol()* receives *window_size* = 9 and *poly_order* = 2 as default values from *signal_smoother()*. In addition, *savgol()* has two optional arguments: When discussing the algorithm in E2.1, it was not mentioned that it can provide derivatives of the filtered time series. Via *deriv* the user can choose the order of the derivative of the time series. Throughout the rest of this project we will work with the default "*deriv* = 0", i.e. we will work with the actual filtered time series. Via *weighted* the user can choose whether the original Savitzky-Golay filter as in (E2.1.2) or its weighted version as in (E2.1.5) should be applied.

In line 3, the number of points to the left and right of the data point to be smoothed is saved as a variable, since this value is needed multiple times during a single function call. Then, lines 5 and 6 initialize matrix $\boldsymbol{J}$ from E2.1 by means of a nested list comprehension.

Lines 7 - 16 prove my laziness. If one looks at E2.1 superficially, one would have to implement two different sets of equations, one set for the unweighted version and one set for the weighted version of the Savitzky-Golay filter. In fact, however, it is sufficient to just implement the set of equations for the weighted version and $\boldsymbol{W}$ will take care of the rest: If $\boldsymbol{W}$ is the identity matrix where the number of rows and the number of columns are equal to the number of rows of $\boldsymbol{J}$, (E2.1.3) reduces to (E2.1.1) and (E2.1.5) reduces to (E2.1.2). Hence, if *weighted* = *False*, line 8 is executed and $\boldsymbol{W}$ is generated as the identity matrix. If, on the other hand, *weighted* = *True*, lines 10 - 13 are executed and the main diagonal of $\boldsymbol{W}$ is filled according to (E2.1.4). Lines 14 - 16 then only define the remaining matrix multiplication steps. Line 16 finally obtains the convolution coefficients by selecting the *deriv*th row of $(\boldsymbol{J}'\boldsymbol{W}\boldsymbol{J})^{-1}\boldsymbol{J}'\boldsymbol{W}$. Note that by default, this selects the zeroth row of $(\boldsymbol{J}'\boldsymbol{W}\boldsymbol{J})^{-1}\boldsymbol{J}'\boldsymbol{W}$.

Applying convolution as in (E2.1.2) and (E2.1.5) raises the question of how to treat the endpoints. While the first elements of the time series do not have enough preceding data points, the last elements do not have enough subsequent data points to smooth with the given window size. There are at least two ways to fix this problem: Either smoothing starts at index *half_window* and ends *half_window* elements before the last element, or the time series is extrapolated, i.e. we concatenate the actual observations with computed observations at the endpoints. Both options have their respective drawbacks. The latter option has a theoretical drawback: The data points at the right end of the time series represent the sensor readings shortly before failure and are therefore of particular interest to us. However, the predictions of their noise-free values will be subject to a larger standard error since they will be partly based on the extrapolated elements of the time series. The former option, on the contrary, has a practical drawback: The smoothed time series will be shorter, i.e. the length of the smoothed time series won't match the other series in the data set. Sequences of unequal lengths cannot constitute a valid data frame for our later model, though. Therefore, the longer sequences would have to be truncated at the edges. Since this would result in a loss of information, extrapolation is used in this work.

The choice of algorithm used for extrapolation can be considered a hyperparameter over which we can optimize performance. Lines 18 and 19 of Code box 19 define how the sequences are extrapolated in this work: First, the *half_window* closest neighbors of the current endpoint are selected. Second, the sequence of the absolute changes between the neighboring values and the value of the endpoint is mirrored at the endpoint. Thus, the *half_window* extrapolated elements at the beginning of the sensor time series are obtained by subtracting the corresponding sequence of deviations from the first observed value. Analogously, the *half_window* extrapolated elements at the end of the time series are obtained by adding the corresponding sequence of deviations to the last observed value. Line 20 then concatenates the sequence of observed values with the extrapolated values. Line 22 finally applies convolution as in (E2.1.2) and (E2.1.5) and returns the resulting smoothed sensor times series.

Code box 19:

```
[001] def savgol(y, window_size, poly_order, deriv=0, weighted=False):
[002]     # how many data points to consider to the left and right
[003]     half_window= int((window_size -1 )/2)
[004]     # compute convolution coefficients
[005]     J = np.mat([[k**i for i in range(poly_order+1)] \
[006]                  for k in range(-half_window, half_window+1)])
[007]     if not weighted:
[008]         W = np.eye(J.shape[0])
[009]     else:
[010]         xjxi = np.arange(-half_window,half_window+1)
[011]         xjxidi3 = (np.abs(xjxi/(half_window)))**3
[012]         wi = (1 - xjxidi3)**3
[013]         W = np.diagflat(wi)
[014]     JW = np.matmul(J.T,W)
[015]     JWJ = np.linalg.inv(np.matmul(JW,J))
[016]     m = np.matmul(JWJ,JW).A[deriv]
[017]     # padding
[018]     firstvals = y[0] - np.abs( y[1:half_window+1][::-1] - y[0] )
[019]     lastvals = y[-1] + np.abs(y[-half_window-1:-1][::-1] - y[-1])
[020]     y = np.concatenate((firstvals, y, lastvals))
[021]     # convolve and return results
[022]     return(np.convolve( m, y, mode='valid'))
[023]
```

signal_smoother() can also be used to filter a time series with the Kalman filter. If *'kalman'* is passed as the filter method argument, *signal_smoother()* calls *kalman_filter()* from *tfed_preprocessing.py*. The *kalman_filter()* function definition is shown in Code box 20 below. *kalman_filter()* relies on *KalmanFilter()* from the external module *pykalman*. It expects the user to provide the following inputs:

- *initial_state_mean* and *initial_state_covariance*. The very first state cannot be described by (E2.2.1) since there are no lagged values available. Therefore, *KalmanFilter()* expects the user to provide values that are used for the first period.
- *observation_covariance* corresponds to $\boldsymbol{R}_t$ in E2.2.
- *observation_matrices* corresponds to $\boldsymbol{H}_t$ in E2.2.
- *transition_covariance* corresponds to $\boldsymbol{Q}_t$ in E2.2.
- *transition_matrices* corresponds to $\boldsymbol{F}_t$ in E2.2.

In E2.2 it has been noted that the state can be described by a vector of length n. But since each time series is filtered individually in this project, the state at each cycle is described by a single sensor value. Thus, $\boldsymbol{R}_t$, $\boldsymbol{H}_t$, $\boldsymbol{Q}_t$ and $\boldsymbol{F}_t$ reduce to scalar values. Nonetheless, the practical disadvantages of this method come into play at this point: Which values to choose for $\boldsymbol{R}_t$, $\boldsymbol{H}_t$, $\boldsymbol{Q}_t$ and $\boldsymbol{F}_t$? In contrast to other methods, these "structural" parameters are not estimated; they are assumed to be known. As can be seen in Code box 20, instances of the *KalmanFilter*-class are generated with standard values. It must be emphasized that these values were not obtained through expert process knowledge. Therefore, these values are most likely not equal to the true structural parameters and, as a result, filter performance is not optimal. At best, "best guesses" of these parameters could be obtained, e.g. by expectation maximization algorithms. However, this is intended for future work on this project. Once the *KalmanFilter*-instance is generated, all that is left to do is to pass the list of observations (our sensor time series we want to filter) to its *smooth()*-method. The first element of the returned tuple is then the smoothed sensor time series.

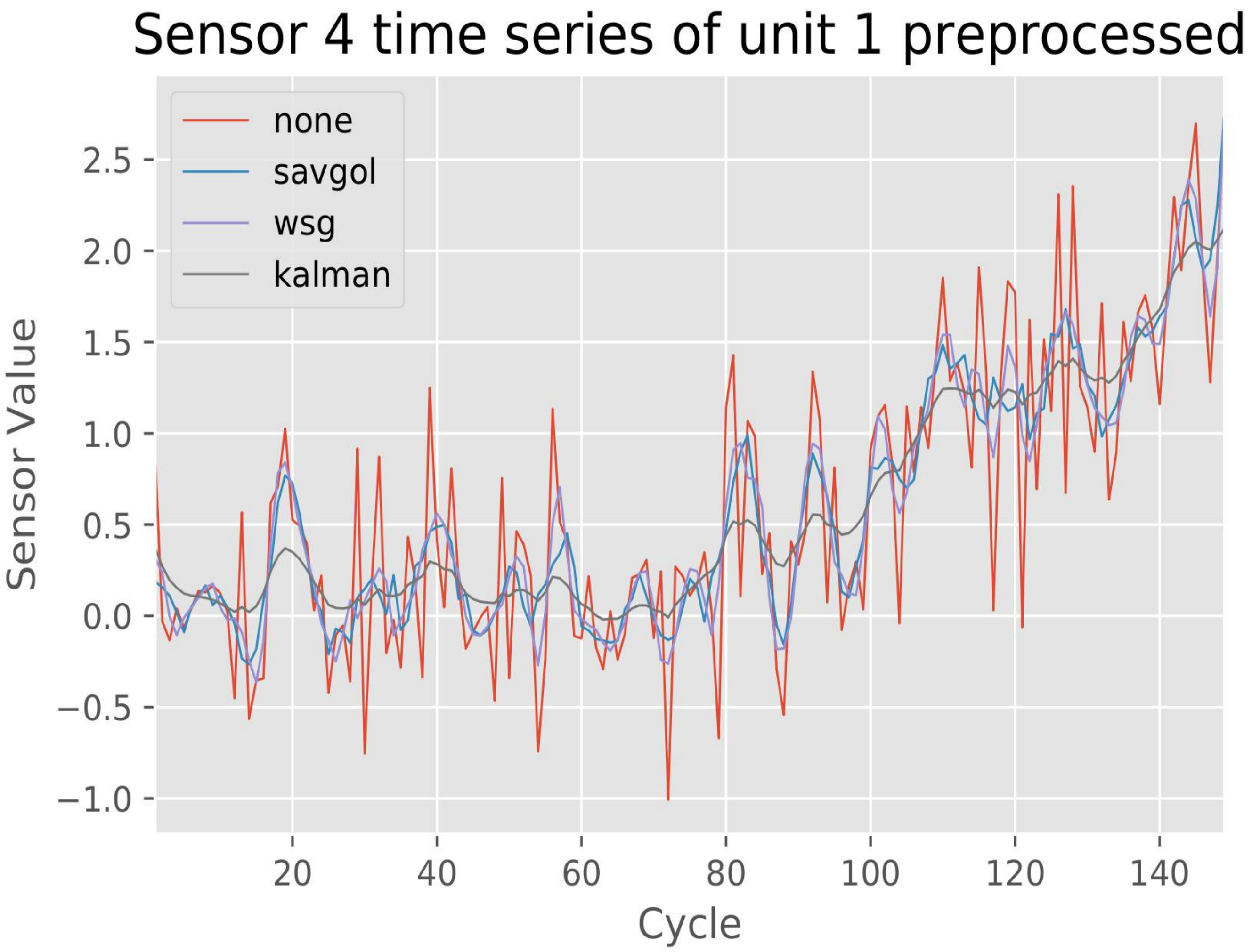
Code box 20:

```
# kalman filter function
def kalman_filter(obs,obs_cov=1,trans_cov=0.1):
    obs = list(obs)
    kf = KalmanFilter(
            initial_state_mean=obs[0],
            initial_state_covariance=obs_cov,
            observation_covariance=obs_cov,
            observation_matrices=1,
            transition_covariance=trans_cov,
            transition_matrices=1
        )
    pred_state, state_cov = kf.smooth(obs)
    return(pred_state)
```

Finally, *denoise_data()* receives the data frames with the filtered time series from *signal_smoother()* and saves them as csv files in the current working directory.

Before we move on to the last preprocessing step, feature selection, we probably want to take a look at the results of our preprocessing efforts so far. Let's take Figure 3 as a starting point. This figure shows the unprocessed time series of sensor 4 of unit 1 from data set FD002. When we first looked at this figure, we noticed the high degree of oscillation. Aside from this feature, we could only conjecture an upward trend, which is blurred by the dominant effects of the various operating conditions. Now let's plot the same time series, but after preprocessing, i.e. the time series is now standardized and operating conditions adjusted. Additionally, we use *denoise_data()* to obtain the plots of the corresponding time series for each filter method. The four resulting graphs are collected in Figure 5 below.

Figure 5: Preprocessed time series of unit 1, sensor 4



none = preprocessed unfiltered, savgol = preprocessed and filtered by standard Savitzky-Golay, wsg = preprocessed and filtered by weighted Savitzky-Golay, kalman = preprocessed and filtered by Kalman filter

All time series now show a clear upward trend. Furthermore, we can see that all three filter methods actually reduced the degree of oscillation. The Savitzky-Golay filter and the weighted Savitzky-Golay filter removed slightly less oscillation than the Kalman filter; however, it should be noted that this observation strongly depends on the selected hyperparameters. Nevertheless, we can quite safely conclude that preprocessing worked as we wanted: While Figure 3 seems to show almost nothing but noise, Figure 5 appears to show informative time series.

3.4 Feature selection

tbc.

The following chapter describes the sampling method, defines the model and employs our preprocessed data for estimation.

4. Estimation

4.1 Difficulties in RUL modeling

When designing remaining useful life models, practitioners often cannot revert to conventional structural modeling approaches. For many of the processes to be examined, formal knowledge about the underlying causal relationships is incomplete (Jensen & Abonyi, 2005). In most cases, this can simply be attributed to missing data. But even if a process has been observed over a period of time, the multitude of potentially interacting factors, the high degree of nonlinearity and/or temporal dynamics prevent the complete uncovering of causal relationships in these processes.

To make matters worse, this project just throws the data at us with minimal accompanying information. We know that the data are organized as multivariate time series and we know they include three variables describing the operational condition and 21 variables that represent sensor recordings. However, the readme doesn't give us a crash course on fault propagation in turbines. It doesn't even tell us which sensor measures which quantity.

So we cannot postulate a structural model and analyze causal relationships. However, this chapter is intended to show that we can nevertheless formulate an algorithm with which predictive maintenance can be used for the use case presented in this project.

As a starting point: At least we have data. For each engine-cycle pair *i*, our training data set contains the true number of cycles until failure, y_i, as well as a feature vector $\boldsymbol{x}_i$. Ideally, we would want to find a function f^+ which exactly maps all possible $\boldsymbol{x}_i$ to their corresponding y_i. Moreover, f^+ should yield a correct mapping for new $\boldsymbol{x}_i$ that are not included in our training data. But even if $\boldsymbol{x}_i$ contains all relevant data, f^+ does not exist in most cases due to the stochastic component introduced by random errors in measuring the process. For most practical applications, however, it suffices to find a function f^* which maps $\boldsymbol{x}_i$ to $\hat{y}_i$, where $\hat{y}_i$ is a "good enough" estimate of y_i.

Now the question is: How can we find f^*? Suppose $F^{\sim}$ is the set of all possible functions that map $\boldsymbol{X}$ to any $\hat{Y}$ and F is the set of functions we search through, $F \subset F^{\sim}$. For the reasons described above, we don't want to impose strong assumptions on the data generating process just by restricting f^* to be of any specific functional form. Therefore we need a function approximator that can search through a very large F, i.e. a "universal function approximator". Once we defined our function approximator, we need a mechanism to choose f^* among all possible f in F. To achieve this, let's assume a loss function $L: \hat{Y} \times Y \rightarrow \mathbb{R}_{\geq 0}$, where $\hat{y}_i = f(\boldsymbol{x}_i)$; i.e. L assigns each pair $(\hat{y}_i, y_i)$ a positive real value. The less $\hat{Y}$ and Y differ, the smaller this *loss* should be. Now we can use Statistical Learning Theory and define the so-called empirical risk as

$$R(f) \quad = \quad \int L(f(x), y) \, dP(x, y),$$

i.e. the empirical risk associated with f is the expectation of the loss function. Finally, we choose the f that minimizes $R(f)$, i.e.

$$f^* \quad = \quad \underset{f \in F}{arg\,min} \; R(f) \tag{3}$$

The following sub-sections discuss these steps in detail: Sub-sections 4.2 and 4.3 discuss how we can determine the architecture of our function approximator. Sub-section 4.4 then defines and discusses L, and sub-section 4.5 discusses the optimization step of (3). Finally, sub-section 4.6 briefly summarizes the experimental design.

4.2 Sampling method

It's not that we can't assume anything about the data generating process. Currently, our training data provide 408 full degradation trajectories; i.e. we observed 408 engines until they reached an RUL of zero. However, we cannot define full trajectories as observations, since we couldn't exploit variation within our dependent variable in the optimization step. A rather problematic solution would be to define each of the 85338 engine-cycle pairs in our training data set as an independent observation. Here's why this would be problematic: By doing so we assumed away the panel data structure of our training data (we observe engines repeatedly over time). Due to this structure, the engine-cycle pairs are grouped. Hence we can suspect that there exists some kind of structural dependency among different engine-cycle pairs. That shouldn't seem too far-fetched: Engines might slightly vary in their characteristics, e.g. manufacturing inefficiencies might result in varying degrees of initial wear.

So the option we are left with is to define sections of the trajectories as observations: Consequently, a window of *x* cycles of an engine constitutes an observation. This preserves panel information within observations as well as variation within the dependent variable. *x* can be seen as a hyperparameter where the choice of *x* may itself be subject of a study or be based on cross validation. Once *x* has been chosen, we randomly pick *s* such windows from the set of our 85338 engine-cycle pairs. Like *x*, *s* can be viewed as a hyperparameter. The choice of *s* depends on *x*. Note that we allow overlapping observations by randomly selecting these windows and the larger *x* and *s*, the larger the overlap.

Code box 21 shows the Python implementation of the just described sampling method. *train_df_generator* expects the arguments *df*, which corresponds to the training data frame from the previous chapter, and *use_cols*, which is the list of features. Furthermore, it expects the number of samples, *n_sampling*, which corresponds to *s* above and the window size, *time_series_length*, which corresponds to *x* above. As a first step, *train_df_generator* obtains a list of *n_sampling* randomly chosen engine-cycle pairs. Note that only pairs can be chosen where *cycle* is at least as large as *time_series_length*, since we have defined the entire period from the current cycle to the (*time_series_length*-1)-th cycle before it as one observation. Then *x_train* needs to be initialized: *x_train* is the input data frame for our later model and is a cuboid with dimensions [*N_samples*, *window_size*, *N_features*], where *N_features* is the length of *use_cols*. The nested for-loop in lines

19 - 22 finally fills *x_train* with the corresponding data from *df*. Line 24 then returns *x_train* with its associated vector of labels.

Code box 21:

```
[001] def train_df_generator(df, n_sampling, time_series_length, use_cols):
[002]
[003]     # randomly shuffle indices of unit-cycle pairs where cycle >=
[004]     time_series_length
[005]     df['idx'] = df.index
[006]     idxs = list(df['idx'].loc[df['cycle']>=time_series_length])
[007]     idxs = random.sample(idxs, n_sampling)
[008]     idxs = random.choices(list(df['idx'].loc[df['cycle']>=time_series_length]),
[009]                              k=n_sampling)
[010]
[011]     # get y_train
[012]     y_train = np.array(df.loc[idxs, 'cycles_to_failure'])
[013]     y_train = np.float32(y_train)
[014]
[015]     # initialize X_train
[016]     X_train = np.zeros((n_sampling, time_series_length, len(use_cols)))
[017]
[018]     # fill return array with reshuffled time series
[019]     for d1 in range(0, n_sampling):
[020]         idx = idxs[d1]
[021]         for d2 in range(0, time_series_length):
[022]             X_train[d1,d2,:] = df.loc[idx-(time_series_length-1)+d2, use_cols]
[023]
[024]     return((X_train, y_train))
[025]
```

Note that we now already determined an important aspect about the function approximator: We account for temporal dynamics with a finite number of lagged feature values. In the next sub-section, we will therefore have to craft a universal function approximator that can incorporate time domain information.

4.3 Recurrent Neural Networks

tbc.

4.4 Loss function

In the previous sub-section we laid the groundwork so that we can approximate our unknown target function in a very flexible way. In this sub-section we define the loss function, our mechanism that tells us how good an approximation is. As long as the loss function, L, is of the form $L:\hat{Y}\times Y\rightarrow \mathbb{R}_{\geq 0}$, there are hardly any practical restrictions on it. Classic examples in the context of continuous dependent variables include the L1 norm loss function

$$l_{l1} = \sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{4}$$

and the L2 norm loss function

$$l_{l2} = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{5}$$

Both functions are symmetric about zero. While (4) is less sensitive to outliers, (5) is still often preferred due to the fact that it is differentiable on its whole domain, including zero.

Even if there are hardly any practical restrictions, we should be aware that we use the loss function to steer our approximation of the target function towards a desired performance. Consequently, possible preferences of the decision maker with regard to prediction errors should be taken into account at this point. This also applies to the present project: While early predictions may lead to economic inefficiencies, late predictions may be fatal. It can therefore be concluded that both (4) and (5) are unsuitable for the present use case. Instead, we seek an asymmetric loss function that penalizes late predictions more than early predictions. For the present project, we start with the function provided by the project organizers (see Saxena et al., 2008):

$$l_u = \begin{cases} \sum_{i=1}^{n} e^{-\left(\frac{d}{a_1}\right)} - 1, & \text{for } d < 0 \\ \sum_{i=1}^{n} e^{\left(\frac{d}{a_2}\right)} - 1, & \text{for } d \geq 0 \end{cases} \tag{6}$$

where l_u is our loss, n corresponds to the number of samples, d corresponds to the difference between predicted RUL and true RUL, $d_i = \hat{y}_i - y_i$, and $a_1 = 13$ and $a_2 = 10$. Only when the predicted RUL equals the true RUL does l_u reach its minimum at zero. As the difference between the predicted and the true value increases, l_u grows exponentially. Note that the scalar values a_1 and a_2 control the degree of asymmetry in (6): If a_1 was equal to a_2, (6) would be symmetric. If $a_1 > a_2$, (6) indicates a preference for early predictions over late predictions, as the slope of the loss is steeper for late predictions.

Even though the organizers provided (6) as loss function, they noted that (6) does not yet reflect all possible preferences. For example, (6) does not yet show that the importance of predictions shortly before failure is greater than that of predictions with a long remaining RUL. For this reason, the loss function was adjusted as follows for the present project:

$$l \quad = \quad w \circ l_u \tag{7}$$

where $\circ$ denotes the Hadamard product and w is the vector of weights. Each entry in w is defined as

$$w_i \quad = \quad \frac{1}{g_i}, \quad \text{with } g_i = y_i + 10, \tag{8}$$

i.e. the closer an observation is to failure, the more weight is assigned to it.

So much for the theoretical consideration of our loss function. Let's take a look at its implementation in Python.

Code box 22:

```
[001] # tensorflow weighted custom loss function
[002] def tf_wclf(y_true, y_hat):
[003]     y_plus = tf.add(10., y_true)
[004]     weights = tf.truediv(1., y_plus)
[005]     loss = tf.where(tf.less(y_hat - y_true, 0), \
[006]                     tf.math.exp(tf.math.negative(y_hat - y_true)\
[007]                                 /tf.constant([13.])) - 1,
[008]                     tf.math.exp((y_hat - y_true)\
[009]                                 /tf.constant([10.])) - 1)
[010]     w_loss = weights * loss
[011]     return(tf.reduce_sum(w_loss))
[012]
```

In sub-section 4.2.1 we defined the architecture of our recurrent neural network (RNN) using *tensorflow*. Since we want to use that RNN together with our loss function L to approximate our unknown objective function in the next sub-section, we have to implement L in such a way that it can work with *tensorflow* tensors.

As can be seen in Code box 22, this is not that complicated: In line 2 of Code box 22, we declare L as *tf_wclf* in Python (*tf_wclf* is short for tensorflow weighted customized loss function) and say that it expects two inputs, where *y_true* represents $\boldsymbol{y}$ and *y_hat* represents $\hat{\boldsymbol{y}}$. Note that *y_true* and *y_hat* will be *tensorflow* tensors. Lines 3 and 4 compute the vector of weights, $\boldsymbol{w}$, from (8); lines 5 - 9 are simply the implementation of (6). Line 10 is the implementation of (7) and yields the vector, or rather first order tensor, *w_loss*. Each element of *w_loss* corresponds to the loss value of one $(y_i, \hat{y}_i)$ pair. Line 11 finally computes the sum of all elements of *w_loss* and returns the result. We have thus fulfilled our goal of this sub-section: We assign $(\boldsymbol{y}, \hat{\boldsymbol{y}})$ a non-negative real number that, taking into account our preferences regarding weighting and sign of our prediction errors, represents a measure of "how good" an approximation is.

4.5 Optimization

(tbc.) The adam optimizer is used in this project.

4.6 Experimental design

Four different pairs of training and test data sets were used for estimation. Each of these pairs represents one filter method set-up: In one pair all sensor time series were filtered with the Savitzky-Golay filter, in another with the weighted Savitzky-Golay filter and in a third with the Kalman filter. In one pair, the sensor time series remained unfiltered. Apart from that, all sensor time series experienced the same preprocessing as described in chapter 3.

For each filter method set-up, the feature selection algorithm from 3.4 was applied to the initial health control variables after loading the data. This feature selector only keeps the seven most significant initial health control variables. Since the statistical significance of a variable may differ across the various filter method set-ups, the final feature sets may also differ slightly across the filter method set-ups.

The initialization of the model parameters of an artificial neural network and the sampling method from 4.2 add a stochastic component to the estimation procedure. That is, optimization results will likely differ (slightly) when training the same model multiple times. Therefore, for each filter method set-up and each time series length of interest, the performance results reported in the next chapter are all based on 30 replications of the sampling and the optimization step.

Regardless of the time series length, 6000 samples are always selected. Consequently, the overlap of the samples increases with increasing time series length. However, we only analyze relatively short time series with a maximum length of eight periods. The effect of the overlap should therefore be negligible.

In addition to the adam-optimizer and the customized loss function, the following hyperparameters are chosen: The models are trained over 100 epochs with a learning rate equal to 0.0001 and a batch size equal to 16. 75% of the samples are used for training, 25% for validation.

Once the model parameter estimates are obtained, they are used to predict the RUL in the test data set. Unless explicitly stated otherwise, the results of the experiment presented in the next chapter are always based on the models' performance in the test data set.

5. Results

Basically, this project attempts to answer a two-dimensional research question: First, it is to be shown that the proposed recurrent neural network model with its ability to "learn" temporal dynamics and interactions indeed outperforms simple feedforward neural networks which are mainly used for RUL estimation with the C-MAPSS data up to now. Second, it is to be examined what influence the various filter methods have on the performance of our proposed model. To answer the first part of the research question, we need a feedforward "null" model; a model with which we can compare our proposed model performance-wise.

For better comparability, the architecture of the null model follows that of our proposed model whenever possible. The number of hidden layers and their units as well as the choice of activation functions are the same in both models. The main difference between the two models is that the null model only uses the current period's data to compute the weights. In contrast, the proposed model is an LSTM-RNN, i.e. the LSTM cells additionally receive the previous periods' cell outputs as inputs. Therefore the input data shapes differ as well: While the null model's input shape is *number of samples* times *number of features*, the proposed model's input shape is a cuboid with *number of samples* times *number of periods in the observation window* times *number of features*.

To compare the performance, we also need an appropriate metric. In this project, the output of the loss function across all test units at risk was selected as the main metric. Here, "at risk" refers to all units whose actual RUL is less than or equal to 50 cycles. This choice can be justified by the fact that none of the engines already starts with a fault condition. Furthermore, predictions for engines with a large RUL are of little practical relevance. For the sake of completeness, loss function outputs for the entire test sets can be found in the appendix (see Ap3 and Ap4).

The following two tables compare the results of the models after 30 iterations of the experiment for each filter method. Since the proposed model also has a time dimension, the experiments were carried out for different window sizes, *T*.

Table 1: Mean weighted asymmetric error of test units at risk (variance in parentheses) - proposed model

T	unfiltered	Savitzky-Golay	Weighted Savitzky-Golay	Kalman
3	2.7976 (0.6228)	2.1199 (0.0822)	2.5653 (0.2732)	2.3578 (0.5836)
4	2.3464 (0.1772)	2.2731 (0.4826)	2.1193 (0.2768)	1.9760 (0.4510)
8	1.7976 (0.1659)	1.8757 (0.1520)	2.0305 (0.3345)	2.3568 (10.4611)

Table 2: Mean weighted asymmetric error of test units at risk (variance in parentheses) - null model

T; epochs	unfiltered	Savitzky-Golay	Weighted Savitzky-Golay	Kalman
1; 100	21.9658 (165.5817)	6.9579 (3.4127)	8.0982 (6.9777)	7.3758 (3.9333)

When comparing tables 1 and 2, one of the most striking aspects is that the loss values for the null model are strictly larger than for the proposed model. From this it can be concluded that the proposed model strictly outperforms the null model. The hypothetical decision maker would prefer the proposed model to the null model in all specifications.

Furthermore, table 1 indicates that the proposed model's loss values tend to decrease with increasing window size. With the unfiltered input data, the proposed model reduces the loss value by 87.26% for *T* = 3 and by 91.28% for *T* = 8 compared to the null model; correspondingly, with Savitzky-Golay filtered input data, the proposed model reduces the loss by 69.53% and 73.04%, respectively; with the weighted Savitzky-Golay filtered input data, the proposed model reduces the loss by 68.32% and 74.93%, respectively; with the Kalman filtered input data, the proposed model reduces the loss by 68.03% and 68.05%, respectively. This suggests that the proposed model learns from higher order lags.

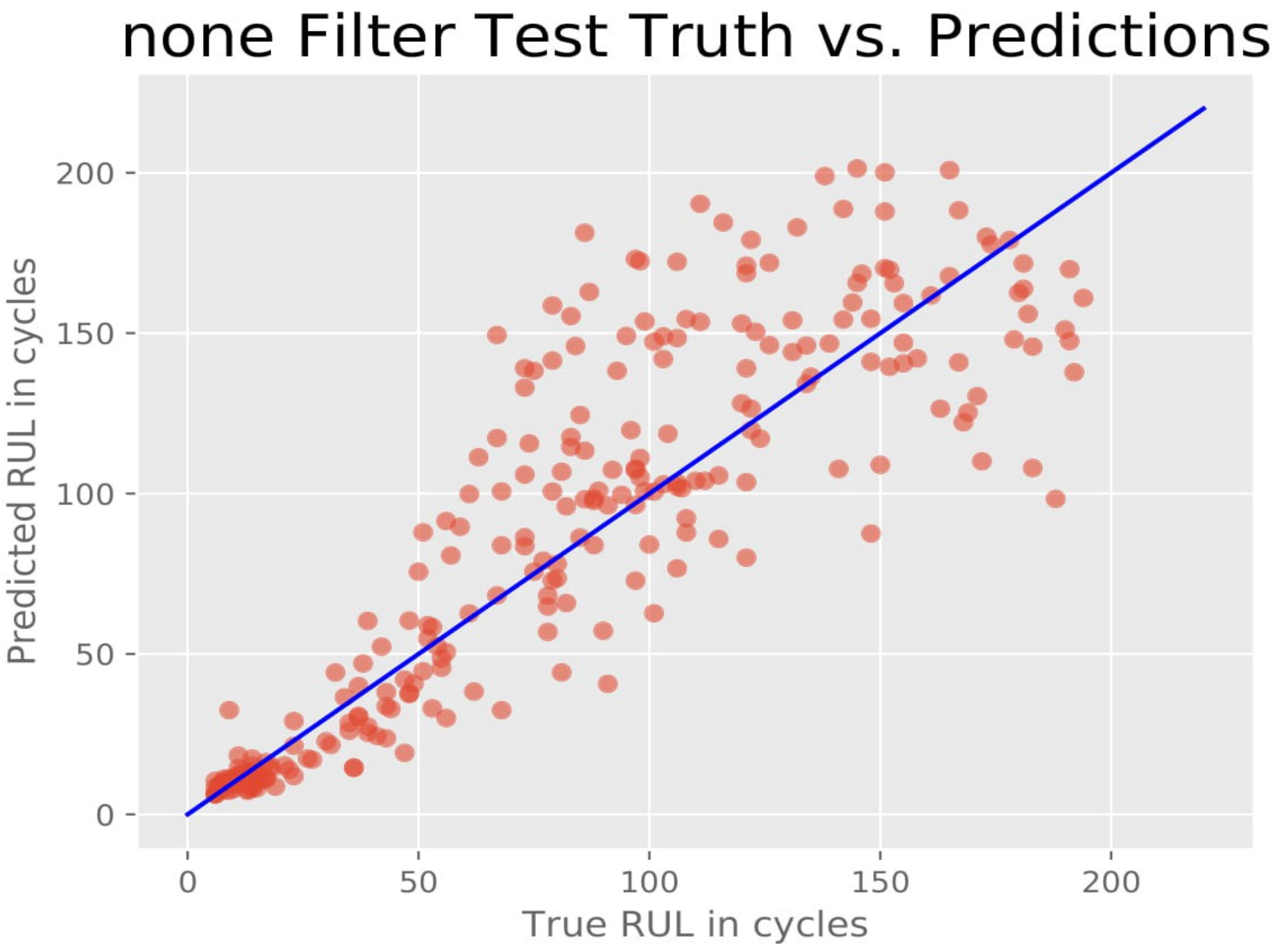
To identify the dominant process that causes this learning, we need to take a closer look at how the various filter methods affect the models' performance: If we only look at the results of the null model, we can see that all three filter methods already reduce the loss value significantly (Savitzky-Golay reduces the loss by 68.32%, weighted Saviztky-Golay reduces the loss by 63.16% and Kalman reduces the loss by 66.42%). If we now take a look at the performance of the proposed model, we can see that for short observation periods, all three filter methods result in a further performance increase. For moderately long observation periods however, this performance advantage vanishes. For $T = 8$, the proposed model with unfiltered input data achieves a loss reduction of 91.82% compared to the null model - a larger reduction than its filtered counterparts. This indicates that the performance advantage of longer observation periods is not solely due to the fact that higher order lags provide additional structural information. Rather, it indicates that the model can learn to filter out sensor noise.

Even if it is not immediately apparent from tables 1 and 2 whether one filter method strictly dominates the others, these results yield a clear recommendation for practical use: Note that the Savitzky-Golay filter and the weighted Savitzky-Golay filter require at least a moderate window size to obtain a filtered sensor value. For moderately long window sizes, however, both filter methods don't provide any performance advantage anymore. In contrast, the Kalman filter only requires the current and the previous sensor value to obtain the corresponding filtered value.

Hence: If long (continuous) sequences of sensor values are only available to the model at large cost or if storing long sequences is costly, feeding short Kalman filtered sequences to the proposed model appears to be the practitioner's preferred choice. If, on the other hand, moderately long or long (continuous) sequences of sensor values are (cheaply) available, the unfiltered data can be used as input data to the proposed model without any degradation of performance.

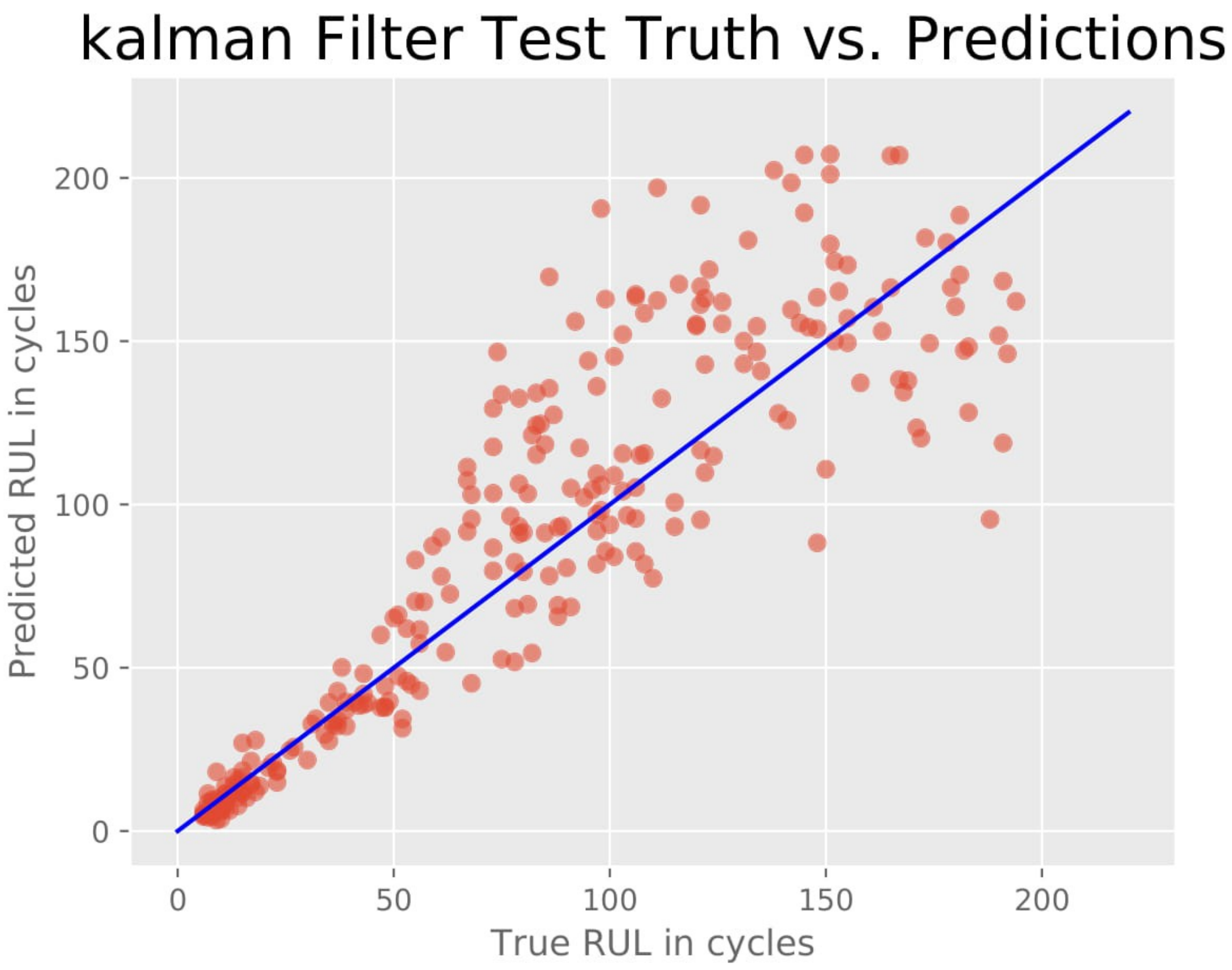
Figures 6 and 7 below summarize the performance of the proposed model in scatter plots. Each scatter plot compares the test data's true values with the correspondig prediction of the proposed model for a window size of two periods. Figure 6 shows the proposed model's performance with unfiltered input data. Figure 7 shows the proposed model's performance with Kalman filtered input data.

Figure 6: True values vs. proposed model's predicted values with unfiltered input data



In both figures we can see that the dispersion of the points increases as the true RUL increases. This can be attributed to the fact that we weighted the observations during training: The smaller the true RUL, the more weight was assigned to the observation. Moreover, both figures confirm our preference to underestimate an observation's RUL: Of the observations that we have defined as "at risk", i,e, observations with an RUL less than or equal to 50 cycles, there are obviously more dots below than above the blue diagonal line. Finally, we can see that the dots in Figure 7 are tighter around the blue line than they are in Figure 6. This illustrates that for short time series the proposed model exhibits a better performance if the input data was preprocessed with the Kalman filter.

Figure 7: True values vs. proposed model's predicted values with Kalman filtered input data



6. Conclusions

This project was intended to develop a viable and robust remaining useful life estimator for use in predictive maintenance. In particular, this estimator accounts for sensor noise and does not require a structural model of the underlying fault propagation process. Additionally, an asymmetric loss function ensures that the preferences regarding under- vs. overestimating a system's remaining useful life are taken into account.

The results of the experiment carried out in this project show that the dominant estimation strategy depends on the length of the available sensor time series. For short sensor time series, the optimal strategy is to feed Kalman filtered input data to the proposed model. For longer time series, the filtering step can be dispensed with as the proposed model learns to filter out the sensor noise. To sum up, the results of the experiment carried out in this project suggest that the proposed estimator outperforms current standard estimators. The proposed model could reduce the time needed for maintainance without increasing the systems' downtime.

Several points remain open for future work. The first question that arises is whether these results also hold for other fault propagation processes. Second, it could be investigated whether extensions to the Kalman filter provide significant performance improvements; for example, the Kalman filter could be adjusted so that it accounts for a nonlinear fault propagation process. Third, in future work the loss function could be adjusted so that is assigns zero weight to non-risk subjects. Fourth, it could be investigated whether an optimized feature selection algorithm provides significant performance improvements.

References

Jensen, B.A. and Abonyi, J. (2005): "Neural networks for process modeling" (https://www.researchgate.net/publication/329530893_Neural_networks_for_process_modeling, last accessed 08.05.2021)

Kalman, R. (1960): "A New Approach to Linear Filtering and Prediction Problems", ASME Journal of Basic Engineering, 82, 35-45.

Pasa, G., Medeiros, I. and Yoneyama T. (2019): "Operating Condition-Invariant Neural Network-based Prognostics Methods applied on Turbofan Aircraft Engines", Proceedings of the Annual Conference of the PHM Society, 11(1), Sep 2019.

Savitzky, A. and Golay, M.J.E. (1964): "Smoothing and Differentiation of Data by Simplified Least-Squares Procedures", Analytical Chemistry, 36, 1627-1639.

Saxena, A. and Goebel K. (2008): "Turbofan Engine Degradation Simulation Data Set", NASA Ames Prognostics Data Repository (http://ti.arc.nasa.gov/project/prognostic-data-repository), NASA Ames Research Center, Moffett Field, CA.

Appendix

Ap1: Descriptive statistics table of data set *train_FD002*

	cycle	*op_setting_1*	*op_setting_2*	*op_setting_3*	*sensor_1*
count	53759	53759	53759	53759	53759
mean	109.154746	23.998407	0.572056	94.046020	472.910207
std	69.180569	14.747376	0.310016	14.237735	26.389707
min	1.000000	0.000000	0.000000	60.000000	445.000000
25%	52.000000	10.004600	0.250700	100.000000	445.000000
50%	104.000000	25.001300	0.700000	100.000000	462.540000
75%	157.000000	41.998000	0.840000	100.000000	491.190000
max	378.000000	42.008000	0.842000	100.000000	518.670000

	sensor_2	*sensor_3*	*sensor_4*	*sensor_5*	*sensor_6*
count	53759	53759	53759	53759	53759
mean	579.672399	1419.971013	1205.442024	8.031986	11.600746
std	37.289399	105.946341	119.123428	3.613839	5.431802
min	535.530000	1243.730000	1023.770000	3.910000	5.710000
25%	549.570000	1352.760000	1123.655000	3.910000	5.720000
50%	555.980000	1369.180000	1138.890000	7.050000	9.030000
75%	607.340000	1499.370000	1306.850000	10.520000	15.490000
max	644.520000	1612.880000	1439.230000	14.620000	21.610000

	sensor_7	*sensor_8*	*sensor_9*	*sensor_10*	*sensor_11*
count	53759	53759	53759	53759	53759
mean	282.606787	2228.879188	8525.200837	1.094962	42.985172
std	146.005306	145.209816	335.812013	0.127469	3.232372
min	136.800000	1914.770000	7985.560000	0.930000	36.230000
25%	139.935000	2211.880000	8321.660000	1.020000	41.910000
50%	194.660000	2223.070000	8361.200000	1.020000	42.390000
75%	394.080000	2323.960000	8778.030000	1.260000	45.350000
max	555.820000	2388.390000	9215.660000	1.300000	48.510000

	sensor_12	*sensor_13*	*sensor_14*	*sensor_15*	*sensor_16*
count	53759	53759	53759	53759	53759
mean	266.069034	2334.557253	8066.597682	9.329654	0.023326
std	137.659507	128.068271	84.837950	0.749335	0.004711
min	129.120000	2027.610000	7848.360000	8.335700	0.020000
25%	131.520000	2387.900000	8062.140000	8.677800	0.020000
50%	183.200000	2388.080000	8082.540000	9.310900	0.020000
75%	371.260000	2388.170000	8127.195000	9.386900	0.030000
max	523.370000	2390.480000	8268.500000	11.066900	0.030000

	sensor_17	*sensor_18*	*sensor_19*	*sensor_20*	*sensor_21*
count	53759	53759	53759	53759	53759
mean	348.309511	2228.806358	97.756838	20.789296	12.473423
std	27.754515	145.327980	5.364067	9.869331	5.921615
min	303.000000	1915.000000	84.930000	10.180000	6.010500
25%	331.000000	2212.000000	100.000000	10.910000	6.546300
50%	335.000000	2223.000000	100.000000	14.880000	8.929200
75%	369.000000	2324.000000	100.000000	28.470000	17.083200
max	399.000000	2388.000000	100.000000	39.340000	23.590100

Ap2: Descriptive statistics table of data set *train_FD004*

	cycle	*op_setting_1*	*op_setting_2*	*op_setting_3*	*sensor_1*
count	61249	61249	61249	61249	61249
mean	134.311417	23.999823	0.571347	94.031576	472.882435
std	89.783389	14.780722	0.310703	14.251954	26.436832
min	1.000000	0.000000	0.000000	60.000000	445.000000
25%	62.000000	10.004600	0.250700	100.000000	445.000000
50%	123.000000	25.001400	0.700000	100.000000	462.540000
75%	191.000000	41.998100	0.840000	100.000000	491.190000
max	543.000000	42.008000	0.842000	100.000000	518.670000
	sensor_2	*sensor_3*	*sensor_4*	*sensor_5*	*sensor_6*
count	61249	61249	61249	61249	61249
mean	579.420056	1417.896600	1201.915359	8.031626	11.589457
std	37.342647	106.167598	119.327591	3.622872	5.444017
min	535.480000	1242.670000	1024.420000	3.910000	5.670000
25%	549.330000	1350.550000	1119.490000	3.910000	5.720000
50%	555.740000	1367.680000	1136.920000	7.050000	9.030000
75%	607.070000	1497.420000	1302.620000	10.520000	15.480000
max	644.420000	1613.000000	1440.770000	14.620000	21.610000
	sensor_7	*sensor_8*	*sensor_9*	*sensor_10*	*sensor_11*
count	61249	61249	61249	61249	61249
mean	283.328633	2228.686034	8524.673301	1.096445	42.874529
std	146.880210	145.348243	336.927547	0.127681	3.243492
min	136.170000	1914.720000	7984.510000	0.930000	36.040000
25%	142.920000	2211.950000	8320.590000	1.020000	41.760000
50%	194.960000	2223.070000	8362.760000	1.030000	42.330000
75%	394.280000	2323.930000	8777.250000	1.260000	45.220000
max	570.810000	2388.640000	9196.810000	1.320000	48.360000
	sensor_12	*sensor_13*	*sensor_14*	*sensor_15*	*sensor_16*
count	61249	61249	61249	61249	61249
mean	266.735665	2334.427590	8067.811812	9.285604	0.023252
std	138.479109	128.197859	85.670543	0.750374	0.004685
min	128.310000	2027.570000	7845.780000	8.175700	0.020000
25%	134.520000	2387.910000	8062.630000	8.648000	0.020000
50%	183.450000	2388.060000	8083.810000	9.255600	0.020000
75%	371.400000	2388.170000	8128.350000	9.365800	0.030000
max	537.490000	2390.490000	8261.650000	11.066300	0.030000

	sensor_17	*sensor_18*	*sensor_19*	*sensor_20*	*sensor_21*
count	61249	61249	61249	61249	61249
mean	347.760029	2228.613283	97.751396	20.864333	12.518995
std	27.808283	145.472491	5.369424	9.936396	5.962697
min	302.000000	1915.000000	84.930000	10.160000	6.084300
25%	330.000000	2212.000000	100.000000	10.940000	6.566100
50%	334.000000	2223.000000	100.000000	14.930000	8.960100
75%	368.000000	2324.000000	100.000000	28.560000	17.135500
max	399.000000	2388.000000	100.000000	39.890000	23.885200

Ap3: Mean weighted asymmetric error over all test units (variance in parentheses) - proposed model

T	unfiltered	Savitzky-Golay	Weighted Savitzky-Golay	Kalman
3	9878.5060 (2751887789)	287.3675 (239311.7)	294.6659 (37948.4)	511.3457 (78886.1)
4	514.4008 (683961.3)	379.2622 (48837.6)	404.7780 (84179.7)	421.4987 (75867.6)
8	516.0288 (89575.7)	401.6064 (135931.5)	477.8868 (87834.6)	510.483 (90906.8)

Ap4: Mean weighted asymmetric error over all test units (variance in parentheses) - null model

T; epochs	unfiltered	Savitzky-Golay	Weighted Savitzky-Golay	Kalman
1; 100	301.0728 (10179.2)	156.8994 (6552.9)	136.7631 (1707.3)	500.5139 (74822.1)