Robust Feature Engineering

Module 1 | Chapter 5 | Notebook 3

To generate predictions with machine learning models from a time series, you often use special characteristic numbers for each cycle. These numbers then take on the role of new features. In this exercise, you'll get to know some of these characteristic numbers and analyze how they are influenced by outliers. By the end of this lesson you will be able to:

- What robust procedures are.
- How to calculate the *median absolute deviation* as an alternative to standard deviation.

Summarizing cycles

Scenario: You work for a company that manufactures hydraulic pumps. A pilot project will investigate how data science could be used to help optimize pump development. In a first step, you will attempt to predict the required cooling capacity based on the temperature values of the machine. You have been provided with a database containing machine data for this purpose: *hydro.db*.

After we got an overview of the data in the last lesson, we now want to decide on a prediction model. We'll select linear regression because we want to predict a continuous variable, the cooling capacity.

As usual, we'll start by preparing the data. In our case, this only affects feature engineering.

But we will start by reading the temperature data from the database. The values in the four DataFrames `df_temp1`, `df_temp2`, `df_temp3` and `df_temp4` each represent temperatures in degrees Celsius (°C). Each row represents a machine cycle. Each column represents a point in time during the cycle. At the end of this cell, we'll take a look at the first 5 rows of the first temperature sensor as an example.

In [1]:
```
# module import
import sqlalchemy as sa
import pandas as pd

# connection to database
engine = sa.create_engine('sqlite:///hydro.db')
connection = engine.connect()

# read out temperature data from sensors
sql_query = '''
SELECT *
FROM temperature_sensor_{}
'''
```

```
df_temp1 = pd.read_sql(sql_query.format(1), con=connection)
df_temp2 = pd.read_sql(sql_query.format(2), con=connection)
df_temp3 = pd.read_sql(sql_query.format(3), con=connection)
df_temp4 = pd.read_sql(sql_query.format(4), con=connection)

df_temp1.head()
```

Out[1]:

	0	1	2	3	4	5	6	7	8	9	...	51	52	
0	46.055	45.949	45.953	45.875	45.801	45.762	45.723	45.723	45.617	45.629	...	46.125	46.125	46
1	46.457	46.363	46.379	46.301	46.203	46.152	46.152	46.125	46.062	45.980	...	46.551	46.562	46
2	44.117	44.031	44.031	43.957	43.859	43.871	43.770	43.691	43.613	43.621	...	44.199	44.141	44
3	45.871	45.793	45.727	45.715	45.621	45.539	45.488	45.445	45.379	45.375	...	45.953	45.871	45
4	46.047	45.969	45.969	45.883	45.793	45.727	45.734	45.637	45.621	45.543	...	46.055	46.062	46

5 rows × 61 columns

Apart from the `cycle_id` column, which is the cycle's identification number, we seem to have 60 seconds per cycle and temperature sensor. The 60 seconds per cycle represent the features. However, it is a fallacy to assume that since all features describe the temperature, our real feature is the temperature. This is a typical property of a time series.

Time series can be analyzed in a variety of ways. Even to give an overview of the methods would go beyond the scope of this chapter. So let's keep this brief: Let's decide to look at the cooling capacity regardless of time. This is an assumption which we would have to justify plausibly in the case of any doubt.

Since we are now looking at the cooling capacity independently of the time and we want to predict it from the temperature data, the time component is of very little use to us. We want to break down the temporal developments into characteristic numbers and then use these as independent temperature features. So to put it another way, we want to calculate the time out of the time series. There's also a wide range of options for how you can do this. We'll use the following two characteristic numbers:

1. The central value of a cycle.
2. The dispersion of a cycle.

In mathematics there are a lot options for these kinds of relationships. In this lesson you'll get to know two of them. We want to embed these numbers into a new `DataFrame` called `df`, which should indicate for each cycle which central value the sensors have measured and what the dispersion of the sensor values is. So we'll create two new features that we can use later for linear regression.

Now let's start creating `df` step by step. For now, let's focus on the first temperature sensor. Initialize `df` just by using the `'cycle_id'` column from `df_temp1`. Then print the first five rows of `df`. Make sure that `df` is really a `DataFrame` at the end.

In [2]:
```
df = df_temp1.loc[:,['cycle_id']]
df.head()
```

Out[2]:

	cycle_id
0	C_312
1	C_313
2	C_314
3	C_315
4	C_316

The best known example of the central value of a data series is the **arithmetic mean** (reference link). You can use the `my_df.mean()` method to summarize the values in a `DataFrame` with the arithmetic mean. If you set its `axis` parameter to `0`, you calculate one average per **column**. However, if you set it to `1`, you get one average value per **row**. We want the latter so that we get a mean value for each cycle.

Calculate the average temperature values per cycle for the first temperature sensor and add them to the `DataFrame` `df` as a new column named `'temp1_central'`. Then print the first five rows of `df` and make sure that everything worked.

In [3]:
```
df.loc[:, 'temp1_central'] = df_temp1.mean(axis=1)
df_temp1.mean(axis=1).head()
```

Out[3]:
```
0    45.979467
1    46.394750
2    44.019700
3    45.951341
4    45.954200
dtype: float64
```

For the mean deviation from the arithmetic mean we want to use the **standard deviation** (reference link). It tells us how large the average distance of any data point is from the mean value.

Calculate the standard deviation of the temperature values per cycle for the first temperature sensor and add them to the `DataFrame` `df` as a new column named `'temp1_dispersion'`. Use the `my_df.std()` method with `axis=1` for this. Then print the first 5 rows of `df`.

In [4]:
```
df.loc[:,'temp1_dispersion'] = df_temp1.std(axis=1)
df.head()
```

Out[4]:

	cycle_id	temp1_central	temp1_dispersion
0	C_312	45.979467	0.177846
1	C_313	46.394750	0.204112
2	C_314	44.019700	0.199925
3	C_315	45.951341	1.451052
4	C_316	45.954200	0.183352

In the next cell, we'll do the same for the other three temperature sensors:

In [5]:
```
# temperature sensor 2 (mean and standard deviation)
df.loc[:, 'temp2_central'] = df_temp2.mean(axis=1)
df.loc[:, 'temp2_dispersion'] = df_temp2.std(axis=1)

# temperature sensor 3 (mean and standard deviation)
df.loc[:, 'temp3_central'] = df_temp3.mean(axis=1)
df.loc[:, 'temp3_dispersion'] = df_temp3.std(axis=1)

# temperature sensor 4 (mean and standard deviation)
df.loc[:, 'temp4_central'] = df_temp4.mean(axis=1)
df.loc[:, 'temp4_dispersion'] = df_temp4.std(axis=1)

df.head()
```

Out[5]:

	cycle_id	temp1_central	temp1_dispersion	temp2_central	temp2_dispersion	temp3_central	temp3_
0	C_312	45.979467	0.177846	50.818033	0.101144	48.190283	
1	C_313	46.394750	0.204112	51.338450	0.111588	48.571650	
2	C_314	44.019700	0.199925	48.729419	3.559435	46.651089	
3	C_315	45.951341	1.451052	50.643050	0.093302	47.977117	
4	C_316	45.954200	0.183352	50.785733	0.109040	48.454464	

Now let's look at how the presence of outliers affects these key figures. First let's consider the temperature values of the fourth sensor for two cycles: `'C_461'` and `'C_462'`. They have the row indices `149` and `150`.

In [6]:
```
df_temp4.iloc[[149, 150], :]
```

Out[6]:

	0	1	2	3	4	5	6	7	8	9	...	51	52
149	41.156	41.078	41.074	41.086	41.078	41.164	41.156	41.156	41.152	41.102	...	41.113	41.086
150	41.074	41.078	41.078	41.078	41.090	41.090	41.066	41.090	41.086	41.102	...	41.172	41.180

2 rows × 61 columns

Visualize the temperature curve of sensor 4 in these two cycles in a line chart. Each line should represent one cycle.

Tip: Make sure that you only use the numerical columns for the visualization. You should also swap around rows and columns with `my_df.T` so that one line is plotted for each cycle.

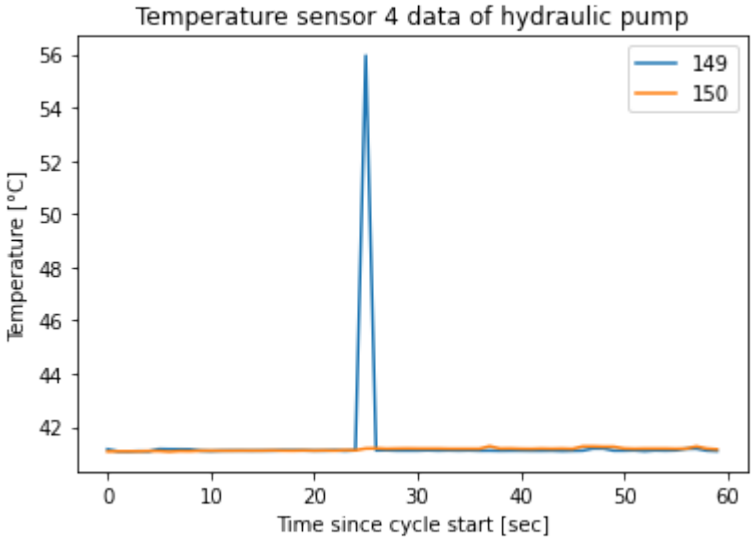
In [7]:
```
# import matplotlib
import matplotlib.pyplot as plt

#drawing figure
fig, ax = plt.subplots()

# plot line chart
df_temp4.iloc[[149, 150], :-1].T.plot(legend=True, ax=ax)

# optimise line chart
ax.set(xlabel='Time since cycle start [sec]',
       ylabel='Temperature [°C]',
       title='Temperature sensor 4 data of hydraulic pump')
ax.set_xticklabels(range(-10, 71, 10))
```

Out[7]:
```
[Text(-10.0, 0, '-10'),
 Text(0.0, 0, '0'),
 Text(10.0, 0, '10'),
 Text(20.0, 0, '20'),
 Text(30.0, 0, '30'),
 Text(40.0, 0, '40'),
 Text(50.0, 0, '50'),
 Text(60.0, 0, '60'),
 Text(70.0, 0, '70')]
```



You should get a graph with the following lines

Except for an outlier at around 25°C in the `'C_461'` cycle (the blue line), the two temperature curves are almost identical. Now print the arithmetic mean for the temperature values for these two cycles.

In [8]:
```
print(df_temp4.iloc[[149, 150], :].mean(axis=1))
```

```
149    41.360111
150    41.152167
dtype: float64
```

Although `'C_461'` and `'C_462'` are virtually identical except for one outlier, their mean values are clearly different. We can see that `'C_461'` has a slightly higher mean value due to a single outlier.

The discrepancy is even clearer in the standard deviations. Print the standard deviations of the temperature values of the fourth sensor from `'C_461'` and `'C_462'`.

In [9]:
```
df_temp4.iloc[[149, 150], :].std(axis=1)
```

Out[9]:
```
149    1.915462
150    0.053144
dtype: float64
```

The standard deviation here is much higher for `'C_461'` than for `'C462'`. This clearly shows that outliers can have a great influence on the arithmetic mean and the standard deviation. These characteristic values therefore suggest a strong discrepancy between the cycles, although at first glance there was practically no difference.

Now if you wanted to train a machine learning model based on the mean and the standard deviation, the algorithm would be strongly influenced by the outliers, since it relies exclusively on numerical differences and would learn the outlier's influence. So what can you do to protect a machine learning model from this kind of misleading information? Generally speaking, there are three approaches:

1. Delete outliers (either delete the data point or just mark the value as `NaN`).
2. Replace outlier values (with a consistent extreme value that is still "acceptable", or with *data imputation* - using a probable value).
3. Minimize the influence of outliers (transform features or use robust methods).

Congratulations: You have learned how outliers influence the mean values and standard deviations. Next, we'll use an approach to minimize their impact: robust methods.

Summarizing cycles with robust methods

Now let's take a closer look at the robust characteristic values for a data set with outliers. You got to know two of them in *Continuous Distributions: Measuring Values (Module 0, Chapter 3)*.

The **median** is the typical value of a distribution. It marks the middle value if you put the data in order from the smallest to the largest value. Note that for the calculation of the median it does not matter whether extreme values occur at the upper or lower end of the value distribution. They don't change the order of the values.

The `my_df.median()` method calculates the median values. Let's try this out with the last two example cycles from the fourth temperature sensor. Remember that `'C_461'` (row index `149`) had almost identical values to `'C_462'` (row index `150`). The only exception was one outlier value.

Use the `my_df.median()` method to calculate the median values of the fourth temperature sensor of these two cycles. Make sure to specify the `axis` parameter correctly here as well.

In [10]:
```
df_temp4.iloc[[149, 150], :].median(axis=1)
```

Out[10]:
```
149    41.105
150    41.172
dtype: float64
```

You can see that `'C_461'` and `'C_462'`, in contrast to the arithmetic mean, have almost identical median values. If you have outliers in your data, it is therefore advisable to take the median rather than the arithmetic mean as the central value of a data series.

The **median absolute deviation** (MAD) (reference link) is a robust value for the data's dispersion. This describes the median of the distances from all data points to the median value of the data series. Since we are also working with the order things are in here, we do not need to worry about extreme values. They do not change the order.

To calculate the MAD, you can use the `mad()` function from `statsmodels.robust`.

Important: The `my_df.mad()` method from `pandas` **does not** calculate the *median absolute deviation*, but the ***mean absolute deviation***, i.e. the **mean value** of the deviations from the **median value** of the data. This also reduces the influence of outliers, but not that much. We recommend using the *median absolute deviation* instead if you want to keep the influence of outliers as small as possible. So you have to use the `mad()` function from the `statsmodels.robust` module!

Remember that the variance in cycles `'C_461'` (row index `149`) and `'C_462'` (row index `150`) looked almost the same. Does this also show up in the dispersion values calculated with the *median absolute deviation*? You can find this with the `mad()` function in the `statsmodels.robust` module.

In [11]:
```
from statsmodels.robust import mad
mad(df_temp4.iloc[[149, 150], :-1],axis=1)
```

Out[11]:
```
array([0.01186082, 0.07561271])
```

Great! The values are more similar, while the standard deviations are very different (see above). This shows how robust methods can minimize the influence of outliers.

We now want to define a second `DataFrame` named `df_robust` to store the robust typical temperature values (median) and robust dispersion values (*median absolute deviation*). First initialize this `DataFrame` with the `'cycle_id'` column from `df_temp1`.

In [12]:
```
df_robust = df_temp1.loc[:,['cycle_id']]
```

Now fill `df_robust` with the median values and the *median absolute deviations* of all the temperature sensors using a `for` loop in the same way you did with `df` previously. For example, name the column for median values from the first temperature sensor `'temp1_central'` and the column for *median absolute deviation* values from the first temperature sensor `'temp1_dispersion'`. Adjust the names for the other sensors accordingly. Then print the first 5 rows of `df_robust`.

In [13]:
```
for sensor in range(1, 5):  # for each thermometer
    sql_query = '''
    SELECT *
    FROM temperature_sensor_{}
    '''.format(sensor)
    df_tmp = pd.read_sql(sql_query, con=connection)

    df_robust.loc[:, 'temp{}_central'.format(sensor)] = df_tmp.median(axis=1)
    df_robust.loc[:, 'temp{}_dispersion'.format(sensor)] = mad(df_tmp.iloc[:, :-1], ax
df_robust.head()
```

Out[13]:

	cycle_id	temp1_central	temp1_dispersion	temp2_central	temp2_dispersion	temp3_central	temp3_
0	C_312	46.043	0.139365	50.873	0.087474	48.199	
1	C_313	46.453	0.191256	51.387	0.113419	48.535	
2	C_314	44.109	0.139365	49.031	0.121573	46.264	
3	C_315	45.830	0.176430	50.645	0.121573	48.016	
4	C_316	46.006	0.176430	50.818	0.108230	48.105	

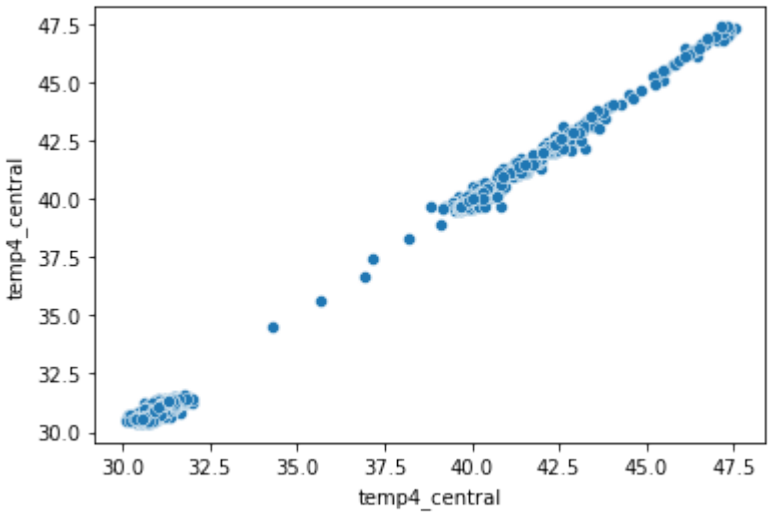
Here's what you should see:

Congratulations: You've got to know robust alternatives to the arithmetic mean and standard deviation. The median and *median absolute deviation* are barely influenced by outliers. This is the kind of behavior you normally want. Next we'll look at how similar the robust and non-robust key figures in this data set are.

df_temp4## Comparing robust and non-robust values

Draw a scatter plot of the mean values (x-axis) and median values(y-axis) of the fourth temperature sensor of all the cycles. If these two ratios do not differ for the typical temperature value of a cycle, they should lie perfectly on one diagonal.

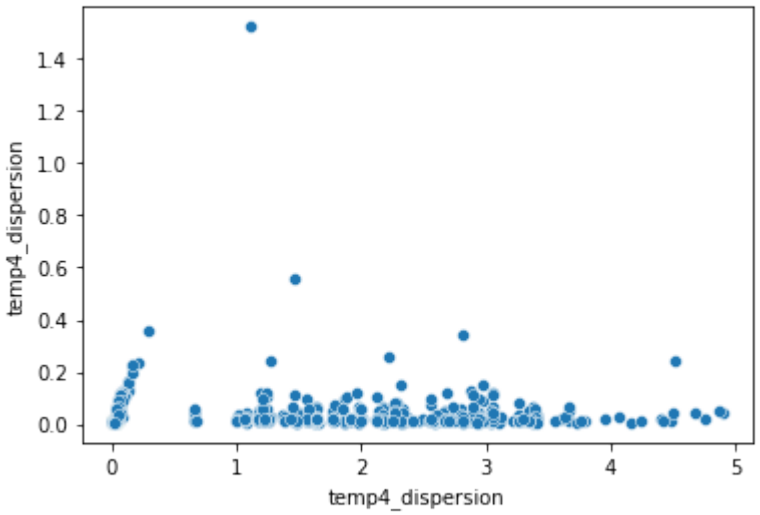
In [14]:
```
import seaborn  as sns
sns.scatterplot(x=df.loc[:, 'temp4_central'], y=df_robust.loc[:, 'temp4_central']);
```



Most of the data points actually do lie on a diagonal. Outliers seem to be quite rare. However, a few data points deviate slightly from it. This shows that the mean and median in our data set are generally almost the same, but in case of doubt only the latter is robust against outliers.

The situation is much more dramatic when it comes to key figures for the data's dispersion. Visualize the standard deviation (x-axis) and the *median absolute deviation* (y-axis) of the fourth temperature sensor of all the cycles in a scatter plot.

In [15]:
```
sns.scatterplot(x=df.loc[:, 'temp4_dispersion'], y=df_robust.loc[:, 'temp4_dispersion'
```



As you can see just by looking at it, the values here do not lie on a variable like the mean and median. From the plot it can be concluded that standard deviation and *median absolute deviation* are generally very dissimilar and only the latter is robust against outliers.

You can now check over this purely visual impression statistically using a correlation matrix. However, we'll skip this point to so we can finish feature engineering for the time being. All you need to do is execute the following code cell. It stores non-robust key figures from all sensors in `df` and the robust key figures from the same sensors in `df_robust`. Then these DataFrames are saved so that we can use them again in the following lesson.

In [16]:
```
# table names in database
tables = ['cooling_efficiency',
          'cooling_power',
          'machine_efficiency',
          'temperature_sensor_1',
          'temperature_sensor_2',
          'temperature_sensor_3',
          'temperature_sensor_4',
          'volume_flow_sensor_1',
          'volume_flow_sensor_2']

# columns names in DataFrames
col_names = ['cool_eff',
             'cool_power',
             'mach_eff',
             'temp_1',
             'temp_2',
             'temp_3',
             'temp_4',
             'flow_1',
             'flow_2']

# initialise DataFrames
df = df_temp1.loc[:, ['cycle_id']]
df_robust = df_temp1.loc[:, ['cycle_id']]

for s in range(len(tables)):  # for each sensor

    # read in data from database
    sql_query = '''
    SELECT *
    FROM {}
    '''.format(tables[s])
    df_tmp = pd.read_sql(sql_query, con=connection)

    # non-robust summary values
    df.loc[:, '{}_central'.format(col_names[s])] = df_tmp.mean(axis=1)
    df.loc[:, '{}_dispersion'.format(col_names[s])] = df_tmp.std(axis=1)

    # robust summary values
    df_robust.loc[:, '{}_central'.format(col_names[s])] = df_tmp.median(axis=1)
    df_robust.loc[:, '{}_dispersion'.format(col_names[s])] = mad(df_tmp.iloc[:, :-1],

# pickle data
df.to_pickle('hydro_data.p')
df_robust.to_pickle('robust_hydro_data.p')

# close connection to data base
connection.close()
```

Congratulations: You have finished feature engineering and created features that shouldn't include outliers and features that should include outliers. In the next lesson we can now use these features to work with a regression to solve the task we have been given.

Remember:

- Median and *median absolute deviation* are robust equivalents of the arithmetic mean and standard deviation
- You can find the *median absolute deviation* with the `mad()` function in `statsmodels.robust`

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.

The data was published here first: Nikolai Helwig, Eliseo Pignanelli, Andreas Schütze, 'Condition Monitoring of a Complex Hydraulic System Using Multivariate Statistics', in Proc. I2MTC-2015 - 2015 IEEE International Instrumentation and Measurement Technology Conference, paper PPS1-39, Pisa, Italy, May 11-14, 2015, doi: 10.1109/I2MTC.2015.7151267.