Robust Regression

Module 1 | Chapter 5 | Notebook 4

In this lesson, we'll use the processed data from the last lesson and predict the machine's cooling capacity. The advantages of robust regression will become clear by comparing it with independent test data without outliers. By the end of this lesson you will be able to:

- use a RANSAC regression.

Predicting cooling capacity

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.

In the last lesson (*Robust Feature Engineering*), you calculated the feature and target values. You determined an average sensor value and a dispersion value of the sensor data for each machine cycle. You can find them in the *pickle* files *hydro_data.p* (arithmetic means and standard deviations) and *robust_hydro_data.p* (medians and *median absolute deviations*). If you can't find these files, you should go back to the previous notebook and run the last code cell.

Import the files and store them as a DataFrames named `df_train` for *hydro_data.p* and `df_robust_train` for *robust_hydro_data.p*. Then print the first five rows of `df_robust_train`.

In [1]:
```
import pandas as pd
df_train = pd.read_pickle('hydro_data.p')
df_robust_train = pd.read_pickle('robust_hydro_data.p')
```

Both DataFrames follow the same data dictionary:

Column number	Column name	Type	Description
0	`'cycle_id'`	categorical (`str`)	Cycle ID number
1	`'cool_eff_central'`	continuous (`float`)	mean/typical value of the combined cooling efficiency (%)
2	`'cool_eff_dispersion'`	continuous (`float`)	dispersion of the combined cooling efficiency (%)

Column number	Column name	Type	Description
3	`'cool_power_central'`	continuous (`float`)	mean/typical value of the combined cooling capacity (kW)
4	`'cool_power_dispersion'`	continuous (`float`)	dispersion of the combined cooling capacity (kW)
5	`'mach_eff_central'`	continuous (`float`)	mean/typical value of the combined machine efficiency (%)
6	`'mach_eff_dispersion'`	continuous (`float`)	dispersion of the combined machine efficiency (%)
7	`'temp_1_central'`	continuous (`float`)	mean/typical value of the temperature from sensor 1 (°C)
8	`'temp_1_dispersion'`	continuous (`float`)	dispersion of the temperature from sensor 1 (°C)
9	`'temp_2_central'`	continuous (`float`)	mean/typical value of the temperature from sensor 2 (°C)
10	`'temp_2_dispersion'`	continuous (`float`)	dispersion of the temperature from sensor 2 (°C)
11	`'temp_3_central'`	continuous (`float`)	mean/typical value of the temperature from sensor 3 (°C)
12	`'temp_3_dispersion'`	continuous (`float`)	dispersion of the temperature from sensor 3 (°C)
13	`'temp_4_central'`	continuous (`float`)	mean/typical value of the temperature from sensor 4 (°C)
14	`'temp_4_dispersion'`	continuous (`float`)	dispersion of the temperature from sensor 4 (°C)
15	`'flow_1_central'`	continuous (`float`)	mean/typical value for the flow rate at sensor 1 (l/min)
16	`'flow_1_dispersion'`	continuous (`float`)	dispersion for the flow rate at sensor 1 (l/min)
17	`'flow_2_central'`	continuous (`float`)	mean/typical value for the flow rate at sensor 2 (l/min)
18	`'flow_2_dispersion'`	continuous (`float`)	dispersion for the flow rate at sensor 2 (l/min)

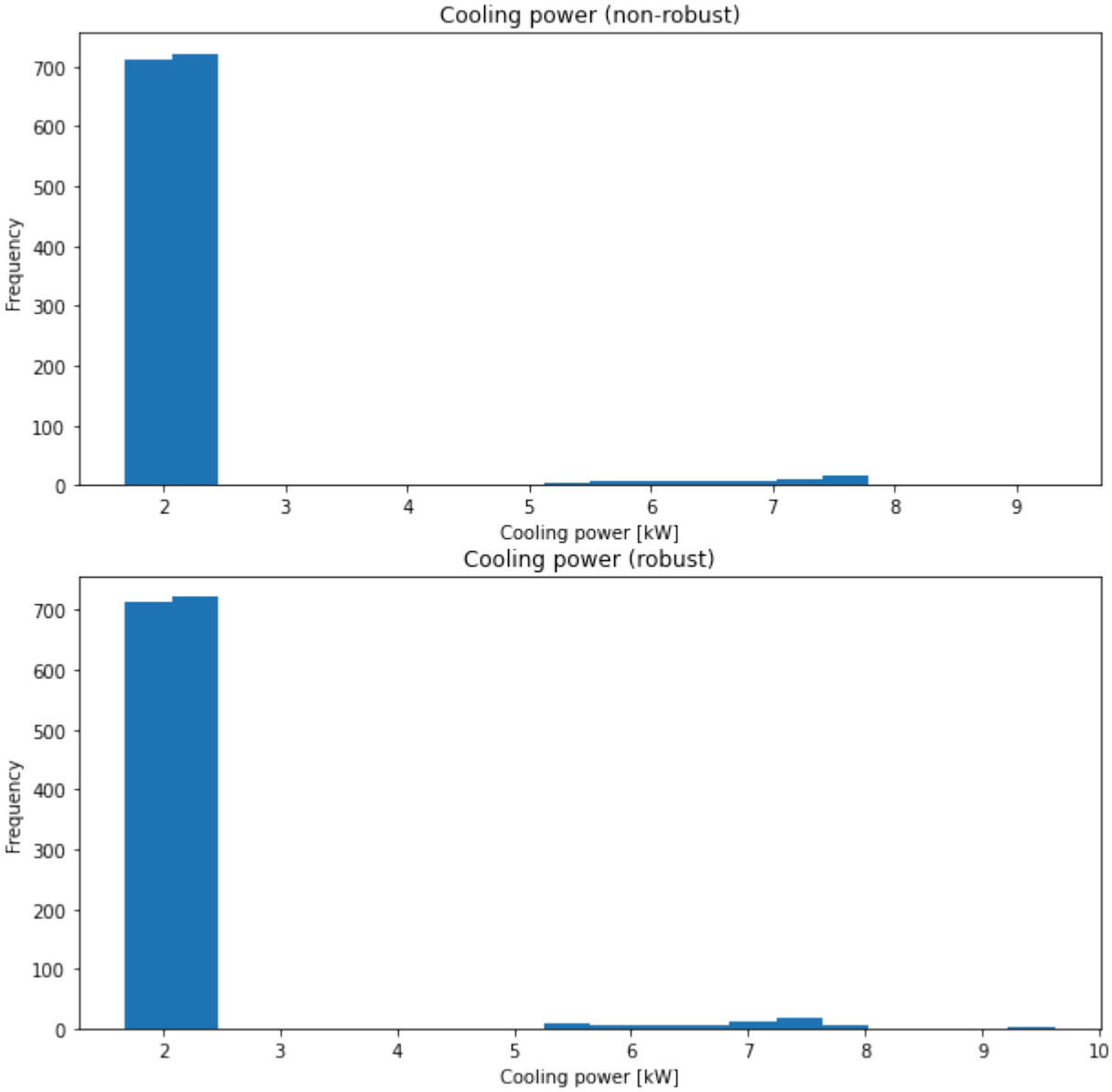
The objective of this lesson is to predict the target variable `'cool_power_central'`. Look at its distribution with a histogram. Use `df_train` as well as `df_robust_train` to draw a histogram of the `'cool_power_central'` column.

In [2]:
```
import matplotlib.pyplot as plt

# initialise figure and axes
fig, axs = plt.subplots(nrows=2, figsize=[10, 10])

# draw histograms
```

```
df_train.loc[:, 'cool_power_central'].plot(kind='hist', bins=20, ax=axs[0])
df_robust_train.loc[:, 'cool_power_central'].plot(kind='hist', bins=20, ax=axs[1])

# optimise histograms
axs[0].set(xlabel='Cooling power [kW]', ylabel='Frequency', title='Cooling power (non-
axs[1].set(xlabel='Cooling power [kW]', ylabel='Frequency', title='Cooling power (robu
```

Out[2]:

```
[Text(0.5, 0, 'Cooling power [kW]'),
 Text(0, 0.5, 'Frequency'),
 Text(0.5, 1.0, 'Cooling power (robust)')]
```



The histograms appear to be almost identical. The vast majority of cooling capacity values are around 2 kilowatts (kW). There are also several outliers around 7 kW. We will now predict the cooling performance step by step using the temperature values (mean value and dispersion value) of all four sensors. In the first step we'll use `df_train`, which contains the arithmetic mean and standard deviation of each cycle. Remember that these values are **not** robust against outliers. Use a linear regression model and 5-fold cross-validation to determine the model quality.

Follow these steps:

1. Import `LinearRegression` and `cross_val_score()` from `sklearn.linear_model`
2. Instantiate the model with the name `model_os`. Use the default settings of `LinearRegression()`. `ols` stands for *ordinary least squares* and describes a standard regression line as you learned in *Simple Linear Regression with sklearn (Chapter 1)*.
3. Divide the data into a feature matrix (every column in `df_train` from `'temp_1_central'` up to and including `'temp_4_dispersion'`) named `features_train` and a target vector (`'cool_power_central'` column from `df_train`) named `target_train`.
4. Use `cross_val_score()` for the 5-fold cross validation, see *Introduction to Pipelines (Chapter 2)*. Use `model_ols`, `features_train`, `target_train` and `'neg_mean_squared_error'` as the `scoring` argument. Store the model quality of each cross validation step in the new variable `cv_results`.
5. Print the averaged model quality value.

In [3]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

model_ols = LinearRegression()

# create features matrix and target vector
features_train = df_train.loc[:, 'temp_1_central':'temp_4_dispersion']
target_train = df_train.loc[:, 'cool_power_central']

# cross validation
cv_results_ols = cross_val_score(estimator=model_ols,
                                 X=features_train,
                                 y=target_train,
                                 cv=5,
                                 scoring='neg_mean_squared_error')

cv_results_ols.mean()
```

Out[3]:
```
-0.7870700613243057
```

You may wonder why we ended up with a **negative** mean squared error (MSE) here now - this is because `sklearn` conventionally always tries to maximize its score, so loss functions like MSE have to be negated here (zero is greater than any negative number). If we get a negative MSE value in `sklearn` like we did here, that's not a bad thing, because we are only interested in the absolute MSE value - which is therefore always positive. However, it's important to understand that the MSE is a quadratic measure, i.e. you always have to take its square root to determine the prediction error.

We get an MSE of about 0.79 kW^2 for the cell. The prediction error is therefore 0.88 kW if you take the root. In the last lesson (*Robust Feature Engineering*) we learned that the arithmetic mean and standard deviation are not robust against outliers. Since the training data had a lot of outliers and these dominated the mean values and standard deviations, these outliers probably also influenced the linear regression's learned slope values a lot.

So using the median and the *median absolute deviation* instead of the arithmetic mean and the standard deviation should solve this problem because these are robust against outliers. Calculate the model quality (*mean squared error*) using a 5-fold cross validation when you are using `df_robust_train`.

In [4]:
```
# create features matrix and target vector
features_robust_train = df_robust_train.loc[:, 'temp_1_central':'temp_4_dispersion']
target_robust_train = df_robust_train.loc[:, 'cool_power_central']

# cross validation
cv_results_robust_ols = cross_val_score(estimator=model_ols,
                                        X=features_robust_train,
                                        y=target_robust_train,
                                        cv=5,
                                        scoring='neg_mean_squared_error')

# summarise scores
cv_results_robust_ols.mean()
```

Out[4]:
```
-0.34117095732490893
```

Thanks to the more robust summary values of each cycle, we were able to halve the mean squared error between the prediction and the measured value. It is now around 0.34 kW^2, so the prediction error is about 0.58 kW.

Congratulations: You have learned how to use the median and *median absolute deviation* to summarize typical values and dispersion without outliers distorting the big picture. This helps when making predictions with a linear regression. However, in a moment you will see that this is not yet sufficient to predict test data that doesn't contain outliers well.

Predicting test data without outliers

Scenario: You company has become aware of all the outliers in the sensor data - and so a small test data set was developed, containing no outliers. The company has asked you to make predictions based on the data in this test data set as best as possible. You can find the data in the *hydro_test.csv* and *hydro_robust_test.csv* CSV files, which you can find in your current working directory.

Import the *.csv* files as `df_test` (*hydro_test.csv*) and `df_robust_test` (*hydro_robust_test.csv*) and print the first 5 rows of each.

In [5]:
```
df_test = pd.read_csv('hydro_test.csv')
df_robust_test = pd.read_csv('hydro_robust_test.csv')
display(df_test.head())
display(df_robust_test.head())
```

	cycle_id	cool_eff_central	cool_eff_dispersion	cool_power_central	cool_power_dispersion	mach_eff_c
0	C_312	35.708767	5.590496	2.236917	0.343522	59.6
1	C_313	44.614367	0.687741	2.771433	0.036325	59.5
2	C_314	46.288917	0.252138	2.840100	0.033436	59.3
3	C_315	46.631317	0.223321	2.787767	0.031884	59.2
4	C_316	47.013633	0.182173	2.735817	0.026775	59.3

	cycle_id	cool_eff_central	cool_eff_dispersion	cool_power_central	cool_power_dispersion	mach_eff_c
0	C_312	36.9290	5.006249	2.3185	0.307225	
1	C_313	44.7280	0.593260	2.7825	0.031370	
2	C_314	46.2810	0.189147	2.8430	0.026457	
3	C_315	46.5715	0.189815	2.7970	0.027064	
4	C_316	47.0295	0.143449	2.7340	0.023444	

They contain summary indicators for each test cycle: *hydro_test.csv* contains mean values and standard deviations, while *hydro_robust_test.csv* contains median values and *median absolute deviations*. Otherwise, the data seems to follow the data dictionary above:

How well can `model_ols` predict the test data with the mean and standard deviation? Calculate the mean squared error.

First train `model_ols` with `features_train` and `target_train`. Then create a feature matrix `features_test` (all columns from `'temp_1_central'` up to and including `'temp_4_dispersion'`) and a target vector `target_test` (`'cool_power_central'` column) from the test data `df_test`. Then use the model to predict the target values of the test data set (`target_test_pred_ols`). Use `mean_squared_error()` from `sklearn.metrics` to compare the actual target values (`target_test`) with the predicted target values (`target_test_pred_ols`).

In [6]:
```
from sklearn.metrics import mean_squared_error

# model fitting
model_ols.fit(features_train, target_train)

# create features matrix
features_test = df_test.loc[:, 'temp_1_central':'temp_4_dispersion']

# predict new target values
target_test_pred_ols = model_ols.predict(features_test)

# save true target values
target_test = df_test.loc[:,'cool_power_central']

# compare actual and predicted target values
mean_squared_error(target_test, target_test_pred_ols)
```

Out[6]:
```
4.504564052960768
```

The mean squared error is now much higher than we would have expected from the cross validation: 4.50 kW^2, i.e. with a prediction error of 2.1 kW The model quality metrics from the cross validation and those from the test data are different when the training data is not representative of the test data. This is the case here because the training data contains a great deal of outliers and the test data does not.

Are the robust values that summarized the cycle data more successful? Calculate the *mean squared error* with the model (`model_ols`) trained on `features_robust_train` and `target_robust_train` , which predicts the target values in `df_robust_test` .

In [21]:
```
# model fitting
model_ols.fit(features_robust_train , target_robust_train)

# create features matrix
features_robust_test = df_robust_test.loc[:, 'temp_1_central':'temp_4_dispersion']

# predict new target values
target_test_pred_ols = model_ols.predict(features_robust_test)

# save true target values
target_robust_test = df_robust_test.loc[:,'cool_power_central']

# compare actual and predicted target values
mean_squared_error(target_robust_test, target_test_pred_ols)
```

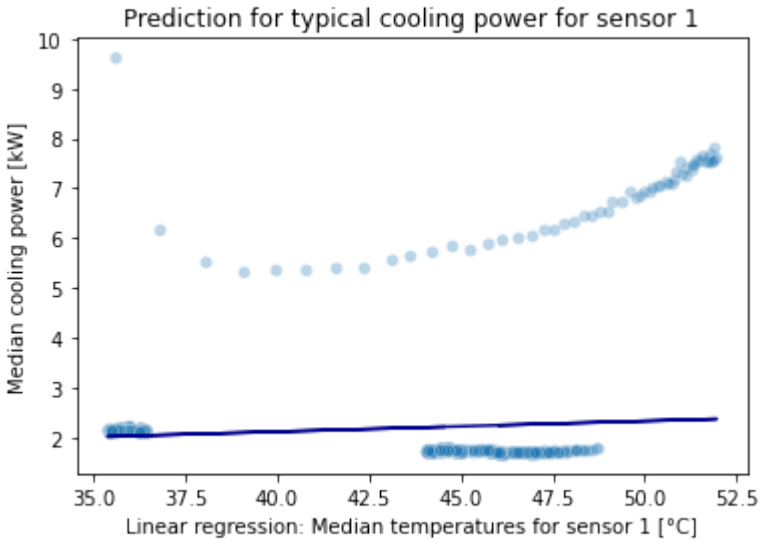
Out[21]:
```
13.789232288315805
```

The mean squared error is now even higher: 13.79 kW^2, the prediction error is therefore around 3.7 kW. Why is that?

The best way to explore the problem is with a scatter plot. If we limit the regression problem to making predictions with only one feature, we can visualize it well. The following scatter diagram shows the median cooling capacity values (y-axis) as a function of the median temperature values for sensor 1 (x-axis).

In [24]:
```
# module import
import seaborn as sns

# initialise figure and axes
fig, ax = plt.subplots()

# draw scatter plot
sns.scatterplot(data=df_robust_train,
                x='temp_1_central',
                y='cool_power_central',
                alpha=0.3,
                ax=ax)

# fit model in preparation for visualising regression
model_ols.fit(features_robust_train.iloc[:, [0]], target_robust_train)

# add regression line
```

```
ax.plot(features_robust_train.iloc[:, 0],
        model_ols.predict(features_robust_train.iloc[:, [0]]),
        color='darkblue')

# optimise line chart
ax.set(xlabel='Linear regression: Median temperatures for sensor 1 [°C]',
       ylabel='Median cooling power [kW]',
       title='Prediction for typical cooling power for sensor 1');
```



The dark blue line represents the predicted values. Although the typical cooling capacity values decrease with higher temperatures, the line increases as it tends to follow the trend of the outliers and so it neglects the non-outliers to some extent. You can see the outliers at the top of the scatterplot, especially gathered at the top right.

So it seems that, in addition to certain points in time within many cycles, which are outliers, there were also whole outlier cycles in which all measured cooling performance values are extremely unusual. You can see this in the scatter plot between a cooling capacity of 5 and 10 kW. The typical cooling capacities are around 2 kW. If the entire cycle has extremely unusual values, even the summary with median and *median absolute deviation* will not help.

Important: This problem was not apparent during cross validation. The reason is obvious: For cross validation, the model quality measures are calculated with a validation set, which itself can also contain outliers. So a high model quality according to cross validation does not protect against outliers influencing the predictions too much. You should therefore take a good look at your data set in advance and have a test data set without outliers up your sleeve.

Congratulations: The test data has shown how much outliers - i.e. extremely unusual and rare data points - influence linear regression predictions. In our case a few outlier cycles confused the machine learning model. Therefore, the next thing we you'll learn is a robust regression model that does not get so easily confused.

Predictions with RANSAC regression

Because outliers are unfortunately a common problem in companies, there are several robust machine learning methods. `sklearn` offers three robust alternatives to linear regression, see the official documentation page on the topic. In this course we'll looks at the RANSAC algorithm.

RANSAC stands for *RANdom SAmple Consensus*. As the name suggests, the point is that *random samples* only lead to similar model values if they don't contain outliers. RANSAC regression therefore assumes that the non-outliers, or *inliers* of the data set lead to the model learning approximately the same parameters, while the outliers produce all kinds of extreme deviations without consensus.

RANSAC regression proceeds as follows:

1. Select a random sample of data points. The default sample size is one larger than the number of features. All these data points are considered to be hypothetical *inliers*.
2. Fit a linear regression model to the data.
3. All the data points from the entire dataset, whose distance to their prediction is too large, are labeled as outliers. All other data points of the dataset are considered to be *inliers*. From what point is the distance too large and when does the data point become an outlier? The *median absolute deviation* of the residuals (distances from the predicted value to the actual observed value) is used by default for this purpose.
4. The *inliers* are included in the sample. Then points 3 and 4 are repeated.
5. If no more *inliers* are expected, the model is only trained only with the *inliers* in the sample. The outliers are completely ignored.

The algorithm tries this process a few times (100 times by default, stored in the `max_trials` parameter) and then selects the best model.

To get some experience with the RANSAC regression algorithm, you can now import `RANSACRegressor` from `sklearn.linear_model`.

In [9]:
```
from sklearn.linear_model import RANSACRegressor
```

When you instantiate a RANSAC regression, you can set all kinds of hyperparameters, such as the size of the initial sample (`min_samples`) or the distance to the prediction for a data point to be considered an outlier (`residual_threshold`). You can also set the number of attempts (`max_trials`). In our case, however, the default values are a pretty good choice.

Instantiate `RANSACRegressor` and store the model in `model_ransac`.

```
RANSACRegressor(min_samples=int, #Minimum number of samples chosen at
random from original data
                residual_threshold=float, #Maximum residual for a data
sample to be classified as an inlier (MAD by default)
                max_trials=int #Maximum number of iterations for random
sample selection
                )
```

In [10]:
```
model_ransac = RANSACRegressor()
```

Now train the model with the feature and target values that represent the arithmetic mean and standard deviation of each machine cycle, i.e. `features_train` and `target_train`.

In [11]:
```
model_ransac.fit(features_train, target_train)
```

Out[11]:
```
RANSACRegressor()
```

Predict the target values of the test dataset using `features_test` for the RANSAC model from previously and compare them to the actual target values. Is the *mean squared error* now smaller than for the linear regression above?

In [12]:
```
target_test_pred_ransac = model_ransac.predict(features_test)
mean_squared_error(target_test, target_test_pred_ransac)
```

Out[12]:
```
0.11577041343874139
```

Now the predictions are much better: the prediction error has decreased from a MSE of 4.50 kW^2 (i.e. a prediction error of 2.1 kW, see `model_ols`) to a MSE of 0.11 kW^2 (i.e. a prediction error of 0.34 kW, see `model_ransac`). The algorithm was apparently able to minimize the influence of the outlier cycles. If we now minimize the influence of the outlier times in certain cycles by combining the cycles with the median and the *median absolute deviation*, we might achieve even better prediction quality.

Try it out. Train `model_ransac` with `features_robust_train` and `target_robust_train`. Then predict target values using `features_robust_test` and compare them with `target_robust_test`. How big is the *mean squared error* now?

In [22]:
```
model_ransac.fit(features_robust_train, target_robust_train)

target_robust_test_pred_ransac = model_ransac.predict(features_robust_test)
mean_squared_error(target_robust_test, target_robust_test_pred_ransac)
```

Out[22]:
```
0.00043099519286035224
```

The prediction error has now dropped to just 0.02 kW (MSE = 0.0004 kW^2). This shows that it was worth paying attention to minimizing the influence of outliers even when summarizing the sensor values in each cycle. The RANSAC regression can then ignore outliers as best it can, but this is not a panacea.

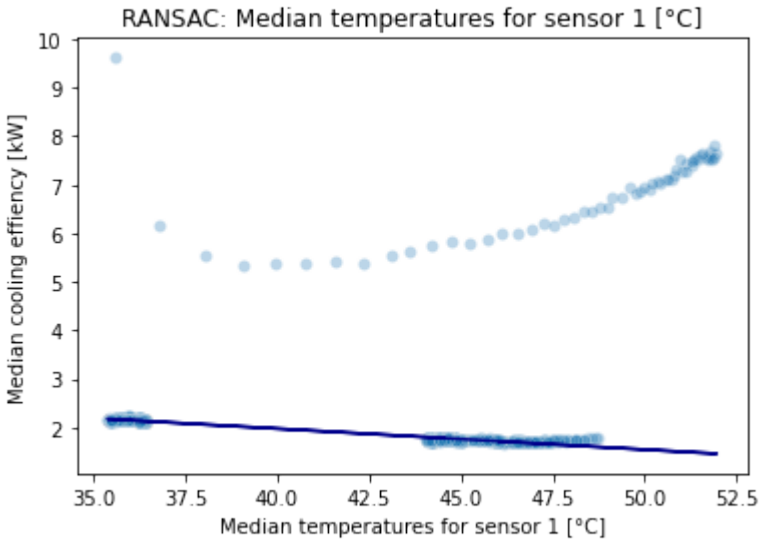
Let's take a visual look at how the RANSAC algorithm fits a regression model to the data. To do this, we'll break the problem down to one feature again:

In [25]:
```
# initialise figure and axes
fig, ax = plt.subplots()

# draw scatter plot
sns.scatterplot(data=df_robust_train,
                x='temp_1_central',
```

```
                y='cool_power_central',
                alpha=0.3,
                ax=ax)

# model fitting (RANSAC regression)
model_ransac.fit(features_robust_train.iloc[:, [0]],
                 target_robust_train)

# add regression line (RANSAC regression)
ax.plot(features_robust_train.iloc[:, 0],
        model_ransac.predict(features_robust_train.iloc[:, [0]]),
        color='darkblue')

# optimise line chart
ax.set(xlabel='Median temperatures for sensor 1 [°C]',
       ylabel='Median cooling effiency [kW]',
       title='RANSAC: Median temperatures for sensor 1 [°C]'
      )
```

Out[25]:

```
[Text(0.5, 0, 'Median temperatures for sensor 1 [°C]'),
 Text(0, 0.5, 'Median cooling effiency [kW]'),
 Text(0.5, 1.0, 'RANSAC: Median temperatures for sensor 1 [°C]')]
```



If you compare the same graph with the linear regression above, it becomes clear that the RANSAC regression did not get confused. The outlier cycles at the top of the scatter plot are simply ignored. Instead, the regression line only follows the inliers at the bottom of the image.

In general, however, the "*garbage in - garbage out*" principle still applies: No matter how good or robust the algorithm is, too many outliers inevitably affect prediction performance. Therefore, at each step of a data pipeline, you should take care to minimize their impact. Robust processes are a promising approach for this.

Congratulations: You've got to know RANSAC regression, which is a robust equivalent to linear regression. RANSAC regression tries to assess what is an outlier and what is not. This will benefit us later, because your company wants to know what data you would now consider to be outliers. So our next step is not to automatically minimize the influence of outliers, but to detect outliers instead.

Remember:

- `RANSACRegressor` is a robust equivalent to `LinearRegression`
- Robust methods can only handle a small number of outliers. If the proportion of outliers in the data set becomes too large, robust procedures can be stretched to their limits.

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.