k-Nearest Neighbors

Module 1 | Chapter 2 | Notebook 3

In this lesson you will use the k-nearest neighbors algorithm to predict room occupancy. This will be the first time you use a classification algorithm. You will also see what you should consider with the k-nearest neighbors approach. By the end of this lesson you will be able to:

- train a k-nearest neighbors model
- estimate the k-parameter of k-nearest neighbors
- understand the importance of feature standardization

Preparation for classification

Scenario: You work for a *smart-building* provider. In a new pilot project, you have been asked to predict whether a person is in the room. If this is not the case for a longer period of time, the *smart building* could automatically switch off the lights or heating. This can reduce costs. The Training data is stored in *occupancy_training.txt*.

Let's start with importing, cleaning and preparing the data. Follow the steps below:

1. Import the `pandas` module with its conventional alias `pd`.
2. Import the data from the csv file *occupancy_training.txt* and store it in `df_train`.
3. Convert the entries of the `'date'` column to the `datetime` data type.
4. Convert the entries of the `'Occupancy'` column to the `category` data type.
5. Create a new column called `'msm'`, which indicates the minutes since midnight.

In [2]:
```
# module import
import pandas as pd

# data gathering
df_train = pd.read_csv('occupancy_training.txt')

# turn date into DateTime
df_train.loc[:, 'date'] = pd.to_datetime(df_train.loc[:, 'date'])

# turn Occupancy into category
df_train.loc[:, 'Occupancy'] = df_train.loc[:, 'Occupancy'].astype('category')

# define new feature
df_train.loc[:, 'msm'] = (df_train.loc[:, 'date'].dt.hour * 60) + df_train.loc[:, 'dat
```

The data dictionary for this data is as follows:

Column number	Column name	Type	Description
0	`'date'`	continuous (`datetime`)	time of the date measurement
1	`'Temperature'`	continuous (`float`)	temperature in ° celsius
2	`'Humidity'`	continuous (`float`)	relative humidity in %
3	`'Light'`	continuous (`float`)	Brightness in lux
4	`'CO2'`	continuous (`float`)	Carbon dioxide content in the air in parts per million
5	`'HumidityRatio'`	continuous (`float`)	Specific humidity (mass of water in the air) in kilograms of water per kilogram of dry air
6	`'Occupancy'`	categorical	presence (0 = no one in the room, 1 = at least one person in the room)
7	`'msm'`	continuous (`int`)	time of day in minutes since midnight

Now let's train the k-nearest neighbors classification model to predict what the `'Occupancy'` status of space is.

In the lesson *Simple Linear Regression with sklearn (Chapter 1)* we learned that in machine learning with `sklearn`, it's best to follow these five steps:

1. Select model type
2. Instantiate the model with certain hyperparameters
3. Split data into a feature matrix and target vector
4. Model fitting
5. Make predictions with the trained model

In principle, we are now only changing steps 1 and 2. Steps 3 and 4 do not change significantly between regression and classification.

Import `KNeighborsClassifier` directly from the `sklearn.neighbors` module.

```
In [4]: from sklearn.neighbors import KNeighborsClassifier
```

Congratulations: You have taken the first steps towards a classification model. We imported the data, cleaned and prepared it. We also decided on the type of classification model. Then the next step is to decide which settings the model should be instantiated with. But to do this you should know how the model actually works. Let's look at that next.

The k of k-nearest neighbors

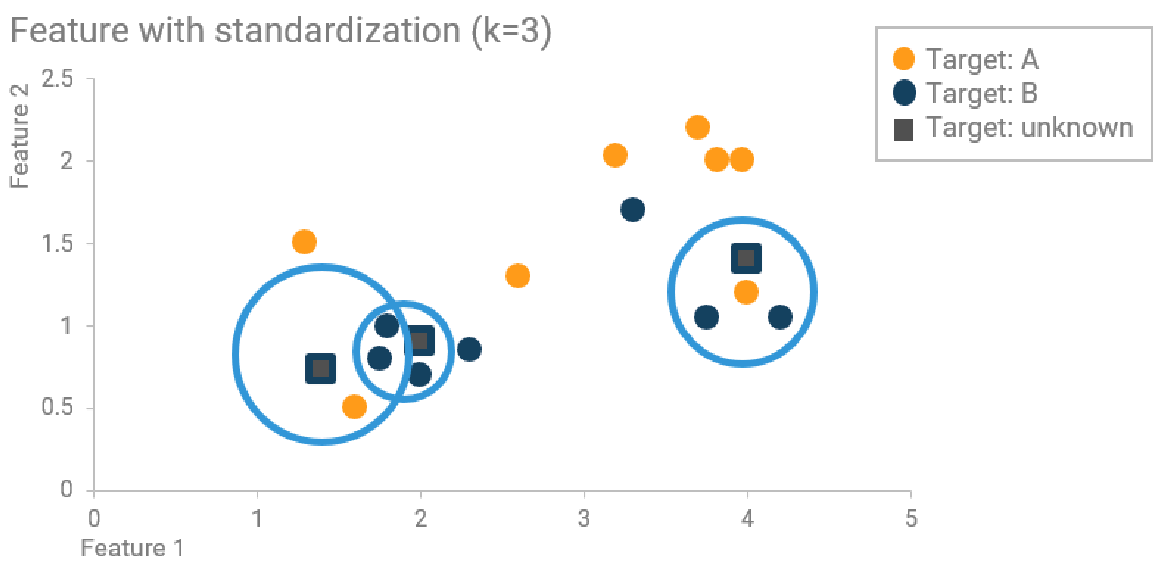
To understand how a k-nearest neighbors model actually works, let's look at sample data in a scatterplot.

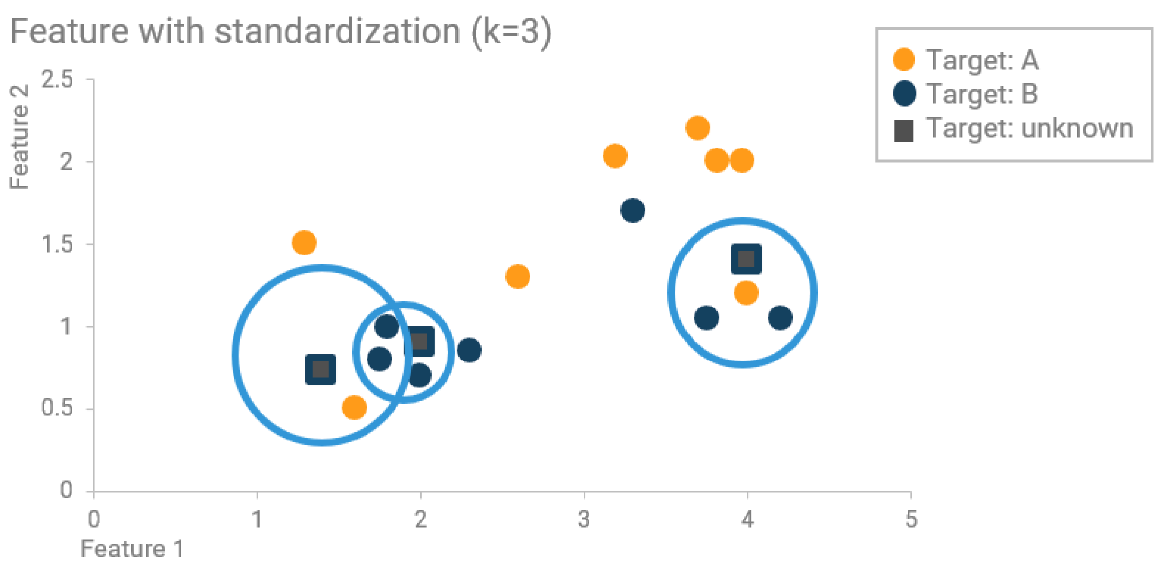
Here we can see two features on the two axes. In our case, they could stand for humidity and brightness. Each data point is a point in time. Orange data points are assigned to class A and dark blue data points to class B. In our case, these would be the categories **person in room** and **no person in room**. The goal is to predict whether the grey data points, which are drawn as squares, belong to A or B.

Unlike most machine learning models, the k-nearest neighbors model does not largely calculate an abstraction of the data points (such as the regression line in the last chapter), but uses the data points directly. If you set k to 1, the system checks which class the nearest data point has. This is then the predicted class of the data point you don't know the class of.

So we can imagine that a circle is drawn around each unknown data point (grey squares). The circle only contains the data point without a category and the nearest neighbor whose category is the predicted category.

If you specify a higher number for the k-parameter, the nearest k-neighbors are used. k is a positive integer (`int`). If you set k to 3, for example, the neighborhood circles are already larger.





Now there are three data points in the neighborhood circle. The category that occurs most frequently in the circle determines which class is predicted for the data point.

Deep dive: The following content is optional, feel free to skip it.

Distance dimensions for k-nearest neighbors

To determine the best value for the k-parameter, the k-nearest neighbors algorithm must calculate the distance from one data point to all the other points and analyze the smallest k-values. We can imagine two data points (x and y) as vectors $\vec{x}$ and $\vec{y}$. The distance between these two vectors is determined by the length (norm) of the distance vector $\vec{d} = \vec{x}-\vec{y}$.

You already learned about norms in the lesson *Measuring Distances with Norms and Metrics (Module 0, Chapter 2)*. The L1 norm, also referred to as the **Manhattan norm**, and the L2 norm, also referred to as the **Euclidean norm**. Both norms follow the general formula:

$L^p(\vec{x})= \left(\sum_i^N |x_i|^{p}\right)^{\frac{1}{p}}$

Applied to our distance vector we get the formula for the **Minkowski distance**:

$d_{minkowski}= \left(\sum_i^N |x_i - y_i|^{p}\right)^{\frac{1}{p}}$

The *p* in the formula is a new hyperparameter that allows us to set the type of distance measure. With $p=1$ we use the Manhanttan distance, with $p=2$ the Euclidean distance. All other values for *p* are simply called Minkowski distances.

The larger the value we choose for *p*, the more important outliers in individual features become.

By default, `sklearn` uses the Euclidean distance with the hyperparameter $p=2$. You can also find more information in the documentation.

All in all, k-nearest neighbors proceeds like this when making predictions:

For each data point to be classified:

1. Determine the distance to every other data point of the training set with $d_{minkowski}= \left(\sum_i^N |x_i - y_i|^{p}\right)^{\frac{1}{p}}$.
2. Select the k-data points with the smallest distance values (the k-nearest neighbors).
3. Determine the mode of classifications of the k-nearest neighbors.
4. Assign the class determined in point 3 to the data point to be classified.

So the name k-nearest neighbors says it all. Remember that k is an integer greater than zero.

Confusingly, `KNeighborsClassifier()` uses the `n_neighbors` parameter to specify k. Instantiate the model with `n_neighbors=3` and store it in the variable `model`. In this chapter we will explore the effect of different k-values.

```
In [6]: model = KNeighborsClassifier(n_neighbors=3)
```

Congratulations: You have instantiated your first classification model. You have also learned the most important hyperparameter: k. Now we can turn to the feature matrix and the target vector.

Standardizing the feature matrix

In the lesson *Lasso Regularization (Module 1 Chapter 1)* you learned that some machine learning models require that all features use the same scale. This can be achieved by standardizing the features using `StandardScaler`. At the end the columns have a mean value of 0 and a standard deviation of 1.

With k-nearest neighbors this is also necessary again. The reason for this is that neighbors should not appear further away on one scale than on the other. Compare the following two scatter plots: The upper one uses two features with similar scale. In the lower graph, the scale of the x-values is twice as long as in the upper graph. The values are otherwise identical.

If you use larger value differences on the x-axis, it is like stretching the data points apart. This may also change the classification of the k-nearest neighbors model. For example, look at the grey square on the far left. In the upper scatter plot, the three nearest neighbors are 2 dark blue (category B) and 1 orange (category A). So category B is predicted for the data point.

In the lower scatter plot, the two blue data points suddenly appear further away because horizontal distances are now twice as large. This means that there are now 2 orange and only 1 dark blue data point among the three nearest neighbors. So the prediction is no longer category B but category A. The scaling of feature 1 on the x-axis has changed the predicted category.

To make sure that the vertical and horizontal proximity is the same, you should use the same scales. You learned how to use a standard scale for all features, in the lesson *Ridge Regularization (Module 1, Chapter 1)*. Import `StandardScaler` directly from the `sklearn.preprocessing` module.

In [7]:
```
from sklearn.preprocessing import StandardScaler
```

Now prepare the feature matrix `features_train` and the target vector `target_train`. At first we'll only use two features: `'CO2'` and `'HumidityRatio'`. `target_train` should store the `Occupancy` column.

In [8]:
```
features_train = df_train.loc[:,['CO2','HumidityRatio']]
target_train= df_train.loc[:,'Occupancy']
```

Now we can standardize `features_train`. First instantiate the variable `scaler` with `StandardScaler()`. Then use its `my_scaler.fit_transform()` method to standardize `features_train` and store it in `features_train_standardized`. `fit_transform` first carries out the *fit* and then the *transform* methods. After that, `scaler` is fitted to the training data and we can use the *scaler* for all further data sets containing the same features.

In [10]:
```
features_train = df_train.loc[:,['CO2','HumidityRatio']]
scaler = StandardScaler()
scaler.fit(features_train)
features_train_standardized = scaler.transform(features_train)
```

Now we can print the eight-value summary of `features_train_standardized` to make sure that the mean values are now really zero (`'mean'`) and the standard deviation is one (`'std'`).

In [11]:
```
pd.options.display.float_format = '{:.2f}'.format  # avoid scientific notation using e
pd.DataFrame(features_train_standardized, columns = ['CO2', 'HumidityRatio']).describe
```

Out[11]:

	CO2	HumidityRatio
count	8143.00	8143.00
mean	-0.00	0.00
std	1.00	1.00
min	-0.62	-1.39
25%	-0.53	-0.92
50%	-0.49	-0.07
75%	0.10	0.57
max	4.52	3.07

Congratulations: We have standardized the values of the feature matrix. This allows us to move on to model training and prediction.

Predicting room occupancy

Equipped with the standardized values in the feature matrix, we can now train the model. We'll use the `my_model.fit()` method, which we already know from the previous chapter. It uses `features_train_standardized` and `target_train`.

In [13]:
```
model.fit(features_train_standardized, target_train)
```

Out[13]:
```
KNeighborsClassifier(n_neighbors=3)
```

The k-Nearest-Neighbors algorithm has a low computational training phase because no abstract model is fitted to the data. Instead, the model remembers all the data points for the prediction. The k-nearest neighbors algorithm is therefore considered to be memory intensive and computationally intensive during the prediction phase.

For the prediction phase, we'll use two sample data points, which you can find in `df_aim` and which we transfer to `features_aim`.

In [14]:
```
df_aim = pd.DataFrame({'CO2':[420, 10000],
                       'HumidityRatio':[0.0038, 0.005]})

features_aim = df_aim.loc[:, ['CO2', 'HumidityRatio']]
```

Standardize the prediction data in `features_aim` and store it in `features_aim_standardized`. Note that `scaler` is already using the standardization settings for the training data - so we'll stick to our principle: **Only ever fit to the training set**. So use `my_scaler.transform()` directly with `features_aim`.

In [20]:
```
features_aim_standardized = scaler.transform(features_aim)
```

Now we can use the `my_model.predict()` method together with `features_aim_standardized` to predict the occupancy of the room. Store the predicted classes in `target_aim_pred` and print them afterwards.

In [22]:
```
target_aim_pred = model.predict(features_aim_standardized)
target_aim_pred
```

Out[22]:
```
array([0, 1])
```

If your output is `array([0, 1], dtype=int64)`, you are correct. The room was predicted to be empty (`0`) for the first point in time with less carbon dioxide content in the air than usual and average humidity. For the second time with a higher CO2 level and high humidity it was predicted that someone is in the room (`1`).

Congratulations: You have learned how to predict categories using a k-nearest neighbors model. You have already learned the most important parameter, k. We'll take a closer look at its effect in the next lesson. You will also learn about the tricky question of how to actually evaluate the model quality for classification models.

Remember:

- The 5 steps and the `sklearn` syntax are almost the same between regression and classification
- The k-nearest neighbors model assigns the class which the k-nearest neighbors have
- The features should be standardized for a prediction with k-nearest-neighbors

---

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: Accurate occupancy detection of an office room from light, temperature, humidity and CO2 measurements using statistical learning models. Luis M. Candanedo, Véronique Feldheim. Energy and Buildings. Volume 112, 15 January 2016, Pages 28-39.