Optional: Combining Grid Search and Feature Selection

Module 1 | Chapter 2 | Notebook 8

In this lesson we will combine the knowledge from previous lessons to select both the most important features and the best hyperparameter values. This will increase the model quality by 10 percentage points so the project can be successfully completed. By the end of this lesson you will be able to:

- Create a list of all possible feature combinations
- Automatically find the best features and hyper parameter values

---

Specifying a grid search

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. Management is already very satisfied that you achieved an F1 score of 87% just by using a clock and a CO2 sensor. Finally, the project managers would like to know which sensors would be necessary to increase this value more. How much can you increase the F1 score by combining the best features?

Let's import the training data now in order to get started quickly.

In [1]:
```
# 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

# take a look
df_train.head()
```

Out[1]:

	date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy	msm
1	2015-02-04 17:51:00	23.18	27.2720	426.0	721.25	0.004793	1	1071
2	2015-02-04 17:51:59	23.15	27.2675	429.5	714.00	0.004783	1	1071
3	2015-02-04 17:53:00	23.15	27.2450	426.0	713.50	0.004779	1	1073
4	2015-02-04 17:54:00	23.15	27.2000	426.0	708.25	0.004772	1	1074
5	2015-02-04 17:55:00	23.10	27.2000	426.0	704.50	0.004757	1	1075

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

The algorithms in this lesson often have to convert `int` numbers to `float` numbers, and will give you a warning each time that you can safely ignore. The following code prevents this kind of warning from being printed:

In [2]:
```
# module import
from sklearn.exceptions import DataConversionWarning
import warnings

warnings.filterwarnings('ignore', category=DataConversionWarning)  # suppress data cor
```

The project managers would like to know which sensors are necessary to increase the model quality (F1 score) to over 87%. For us, this means that we'll try out different features in the feature matrix, determine the best hyperparameters with a grid search and then record the model quality with cross validation. First we have to prepare the grid search.

For the model quality we'll choose the F1 score, which - remember the lesson *Evaluating Classification Performance* - is the harmonic mean of the *recall* and *precision*.

$$\begin{equation*} recall = \frac{correct\ positive}{actually\ positive} \end{equation*}\begin{equation*} precision = \frac{correct\ positive}{positive\ predicted} \end{equation*}$$

And this is how you get the F1 score:

$$\begin{equation*} F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \end{equation*}$$

Let's start by importing the most important tools directly:

- `Pipeline` from `sklearn.pipeline`
- `StandardScaler` from `sklearn.preprocessing`
- `KNeighborsClassifier` from `sklearn.neighbors`
- `GridSearchCV` from `sklearn.model_selection`

In [6]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
```

Next, store the target vector (`'Occupancy'` column in `df_train`) in the variable `target_train` . We'll deal with the feature matrix later.

In [7]:
```
target_train = df_train.loc[:, 'Occupancy']
```

Since we need to standardize the feature matrix at each step of the cross validation during the grid search, we need a pipeline. You learned exactly how this works in *Introduction to Pipelines*.

Name the pipeline here `pipeline_std_knn` . Your first step is standardization with `StandardScaler()` . Call it `'std'` . The next step is a k-nearest neighbors model. Call this step `'knn'` . Use `Pipeline` to create `pipeline_std_knn` .

In [10]:
```
pipeline_std_knn = Pipeline([('std', StandardScaler()),
                             ('knn', KNeighborsClassifier())])
```

Use `GridSearchCV` to define a grid search that goes through all combinations of neighborhood sizes (`n_neighbors` parameter in `KNeighborsClassifier()`) and neighbor weightings (`weights` parameter in `KNeighborsClassifier()`). Store them in the variable `model` . Use this graph to find out the specific settings of the hyperparameters you want to try out:

In [11]:
```
ks = [1, 2, 4, 7, 11, 19, 31, 51, 84, 138, 227, 372, 610, 1000]
```

You can first store the possible hyperparameter values in a variable named `search_space` and then assign `search_space` to the `param_grid` parameter in `GridSearchCV()` . Also, don't forget to set the F1 score as the `scoring` value and specify a two-fold cross validation.

Important: `search_space` must be a `dict`. Each *key* is a parameter (`str`), each *value* is a `list` of the parameter values to be tried out.

In [17]:
```
search_space = {'knn__n_neighbors': ks,
                'knn__weights': ['uniform', 'distance']}

model = GridSearchCV(estimator = pipeline_std_knn,
                     param_grid = search_space,
                     scoring='f1',
                     cv=2,
                     n_jobs=-1)

model
```

Out[17]:
```
GridSearchCV(cv=2,
             estimator=Pipeline(steps=[('std', StandardScaler()),
                                       ('knn', KNeighborsClassifier())]),
             n_jobs=-1,
             param_grid={'knn__n_neighbors': [1, 2, 4, 7, 11, 19, 31, 51, 84,
                                              138, 227, 372, 610, 1000],
                         'knn__weights': ['uniform', 'distance']},
             scoring='f1')
```

Congratulations: You set up a grid search on your own. This involved combining knowledge about k-nearest neighbors, classification quality metrics, pipelines and grid search. These are all things that you learned about in this lesson. So the first part of this lesson was not that easy. Next, you'll use all of this preparation to measure the model quality of various feature matrices.

Testing feature combinations

For the feature matrices, we have so far only tried the combinations `['CO2', 'HumidityRatio']`, `['Humidity', 'CO2', 'HumidityRatio', 'msm']` and `['CO2', 'msm']`. But with six possible features there are 63 possible combinations. Here's a reminder of the features that are available to us.

In [18]:
```
col_of_interest = ['Temperature',
                   'Humidity',
                   'Light',
                   'CO2',
                   'HumidityRatio',
                   'msm']
```

The Python module `itertools` contains a useful function `itertools.combinations()` which returns all combinations of entries in a list. Import `itertools`.

In [20]:
```
import itertools
```

The `itertools.combinations()` function takes two arguments: a list of entries and an `int` number that specifies how many entries should appear in the combinations. You can do this as follows: All the possible feature pairs are:

In [21]:
```
list(itertools.combinations(col_of_interest, 2))
```

Out[21]:
```
[('Temperature', 'Humidity'),
 ('Temperature', 'Light'),
 ('Temperature', 'CO2'),
 ('Temperature', 'HumidityRatio'),
 ('Temperature', 'msm'),
 ('Humidity', 'Light'),
 ('Humidity', 'CO2'),
 ('Humidity', 'HumidityRatio'),
 ('Humidity', 'msm'),
 ('Light', 'CO2'),
 ('Light', 'HumidityRatio'),
 ('Light', 'msm'),
 ('CO2', 'HumidityRatio'),
 ('CO2', 'msm'),
 ('HumidityRatio', 'msm')]
```

You have to pass the output of the `itertools.combinations()` function to the `list()` function, because otherwise it would just output an *iterator*. This is a variable that only contains the instructions on how to output values. It does not include the values themselves. This saves memory. The `range()` function also works this way.

Now write a `for` loop to go through the possible combination lengths 1, 2, 3, 4, 5 and 6. For each combination length, the combinations should be calculated and added to the new `feature_combinations` list. Print `feature_combinations` at the end.

Tip:
Proceed as follows:

1. Create an empty `list` (called `feature_combinations`) where you collect your data
2. In the loop:
   - Iterate through [1,2,3,4,5,6] (for example, call the iterator variable `possible_size_of_combinations`)
   - Use `list(itertools.combinations())` to create a `list` (called `new_combinations`) which contains all new feature combinations of the current length.
   - Add `new_combinations` to `feature_combinations`

In [22]:
```
feature_combinations = []
for possible_size_of_combinations in range(1,  len(col_of_interest) + 1):  # for each

    new_combinations = list(itertools.combinations(col_of_interest,
                                                   possible_size_of_combinations))
    feature_combinations += new_combinations

feature_combinations
```

Out[22]:
```
[('Temperature',),
 ('Humidity',),
 ('Light',),
 ('CO2',),
 ('HumidityRatio',),
 ('msm',),
 ('Temperature', 'Humidity'),
 ('Temperature', 'Light'),
 ('Temperature', 'CO2'),
 ('Temperature', 'HumidityRatio'),
 ('Temperature', 'msm'),
 ('Humidity', 'Light'),
 ('Humidity', 'CO2'),
 ('Humidity', 'HumidityRatio'),
 ('Humidity', 'msm'),
 ('Light', 'CO2'),
 ('Light', 'HumidityRatio'),
 ('Light', 'msm'),
 ('CO2', 'HumidityRatio'),
 ('CO2', 'msm'),
 ('HumidityRatio', 'msm'),
 ('Temperature', 'Humidity', 'Light'),
 ('Temperature', 'Humidity', 'CO2'),
 ('Temperature', 'Humidity', 'HumidityRatio'),
 ('Temperature', 'Humidity', 'msm'),
 ('Temperature', 'Light', 'CO2'),
 ('Temperature', 'Light', 'HumidityRatio'),
 ('Temperature', 'Light', 'msm'),
 ('Temperature', 'CO2', 'HumidityRatio'),
 ('Temperature', 'CO2', 'msm'),
 ('Temperature', 'HumidityRatio', 'msm'),
 ('Humidity', 'Light', 'CO2'),
 ('Humidity', 'Light', 'HumidityRatio'),
 ('Humidity', 'Light', 'msm'),
 ('Humidity', 'CO2', 'HumidityRatio'),
 ('Humidity', 'CO2', 'msm'),
 ('Humidity', 'HumidityRatio', 'msm'),
 ('Light', 'CO2', 'HumidityRatio'),
 ('Light', 'CO2', 'msm'),
 ('Light', 'HumidityRatio', 'msm'),
 ('CO2', 'HumidityRatio', 'msm'),
 ('Temperature', 'Humidity', 'Light', 'CO2'),
 ('Temperature', 'Humidity', 'Light', 'HumidityRatio'),
 ('Temperature', 'Humidity', 'Light', 'msm'),
 ('Temperature', 'Humidity', 'CO2', 'HumidityRatio'),
 ('Temperature', 'Humidity', 'CO2', 'msm'),
 ('Temperature', 'Humidity', 'HumidityRatio', 'msm'),
 ('Temperature', 'Light', 'CO2', 'HumidityRatio'),
 ('Temperature', 'Light', 'CO2', 'msm'),
 ('Temperature', 'Light', 'HumidityRatio', 'msm'),
 ('Temperature', 'CO2', 'HumidityRatio', 'msm'),
 ('Humidity', 'Light', 'CO2', 'HumidityRatio'),
 ('Humidity', 'Light', 'CO2', 'msm'),
 ('Humidity', 'Light', 'HumidityRatio', 'msm'),
 ('Humidity', 'CO2', 'HumidityRatio', 'msm'),
 ('Light', 'CO2', 'HumidityRatio', 'msm'),
 ('Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio'),
 ('Temperature', 'Humidity', 'Light', 'CO2', 'msm'),
 ('Temperature', 'Humidity', 'Light', 'HumidityRatio', 'msm'),
 ('Temperature', 'Humidity', 'CO2', 'HumidityRatio', 'msm'),
```

```
('Temperature', 'Light', 'CO2', 'HumidityRatio', 'msm'),
('Humidity', 'Light', 'CO2', 'HumidityRatio', 'msm'),
('Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio', 'msm')]
```

`feature_combinations` should now look like this:

```
[('Temperature',),
 ('Humidity',),
 ('Light',),
 ('CO2',),
 ('HumidityRatio',),
 ('msm',),
 ('Temperature', 'Humidity'),
 ('Temperature', 'Light'),
 ('Temperature', 'CO2'),
 ('Temperature', 'HumidityRatio'),
 ('Temperature', 'msm'),
 ('Humidity', 'Light'),
 ('Humidity', 'CO2'),
 ('Humidity', 'HumidityRatio'),
 ('Humidity', 'msm'),
 ('Light', 'CO2'),
 ('Light', 'HumidityRatio'),
 ('Light', 'msm'),
 ('CO2', 'HumidityRatio'),
 ('CO2', 'msm'),
 ('HumidityRatio', 'msm'),
 ('Temperature', 'Humidity', 'Light'),
 ('Temperature', 'Humidity', 'CO2'),
 ('Temperature', 'Humidity', 'HumidityRatio'),
 ('Temperature', 'Humidity', 'msm'),
 ('Temperature', 'Light', 'CO2'),
 ('Temperature', 'Light', 'HumidityRatio'),
 ('Temperature', 'Light', 'msm'),
 ('Temperature', 'CO2', 'HumidityRatio'),
 ('Temperature', 'CO2', 'msm'),
 ('Temperature', 'HumidityRatio', 'msm'),
 ('Humidity', 'Light', 'CO2'),
 ('Humidity', 'Light', 'HumidityRatio'),
 ('Humidity', 'Light', 'msm'),
 ('Humidity', 'CO2', 'HumidityRatio'),
 ('Humidity', 'CO2', 'msm'),
 ('Humidity', 'HumidityRatio', 'msm'),
 ('Light', 'CO2', 'HumidityRatio'),
 ('Light', 'CO2', 'msm'),
 ('Light', 'HumidityRatio', 'msm'),
 ('CO2', 'HumidityRatio', 'msm'),
 ('Temperature', 'Humidity', 'Light', 'CO2'),
 ('Temperature', 'Humidity', 'Light', 'HumidityRatio'),
 ('Temperature', 'Humidity', 'Light', 'msm'),
 ('Temperature', 'Humidity', 'CO2', 'HumidityRatio'),
 ('Temperature', 'Humidity', 'CO2', 'msm'),
 ('Temperature', 'Humidity', 'HumidityRatio', 'msm'),
 ('Temperature', 'Light', 'CO2', 'HumidityRatio'),
```

```
('Temperature', 'Light', 'CO2', 'msm'),
('Temperature', 'Light', 'HumidityRatio', 'msm'),
('Temperature', 'CO2', 'HumidityRatio', 'msm'),
('Humidity', 'Light', 'CO2', 'HumidityRatio'),
('Humidity', 'Light', 'CO2', 'msm'),
('Humidity', 'Light', 'HumidityRatio', 'msm'),
('Humidity', 'CO2', 'HumidityRatio', 'msm'),
('Light', 'CO2', 'HumidityRatio', 'msm'),
('Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio'),
('Temperature', 'Humidity', 'Light', 'CO2', 'msm'),
('Temperature', 'Humidity', 'Light', 'HumidityRatio', 'msm'),
('Temperature', 'Humidity', 'CO2', 'HumidityRatio', 'msm'),
('Temperature', 'Light', 'CO2', 'HumidityRatio', 'msm'),
('Humidity', 'Light', 'CO2', 'HumidityRatio', 'msm'),
('Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio', 'msm')]
```

Next we can go through every entry in `feature_combinations` with a `for` loop and create a new feature matrix for each one. You can then use this to find an optimal k-nearest neighbors model with `model` and its `my_model.fit()` method with a grid search. For each loop iteration, print the features used and the best F1 score achieved, as well as the optimal model specification according to the grid search.

Tip:
Proceed as follows:
For each entry in feature_combinations:

1. Print the features used
2. Define the feature matrix
3. Fit the model to the feature matrix and `target_train`.
4. Print the best F1 score.
5. Then print the model specifications.

Attention: In the Datalab the runtime is limited to 10 minutes per cell. So you will get a `TookTooLong` error if you try to run all 63 grid searches in one cell. Adjust your code so that you divide it into three batches of 21 grid searches.

In [23]:
```
#batch 1
for cols in feature_combinations[:21]:  # for each feature combination
    print('Features: ', cols)
    features_train = df_train.loc[:, cols]
    model.fit(features_train, target_train)
    print('Best F1-score: ', round(model.best_score_, 3))
    print('Model spec: ', model.best_estimator_, '\n\n')
```

```
Features:  ('Temperature',)
Best F1-score:  0.597
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=227))])


Features:  ('Humidity',)
Best F1-score:  0.26
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn',
                 KNeighborsClassifier(n_neighbors=2, weights='distance'))])


Features:  ('Light',)
Best F1-score:  0.955
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=11))])


Features:  ('CO2',)
Best F1-score:  0.821
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=372))])


Features:  ('HumidityRatio',)
Best F1-score:  0.336
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=1))])


Features:  ('msm',)
Best F1-score:  0.565
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=11))])


Features:  ('Temperature', 'Humidity')
Best F1-score:  0.709
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=11))])


Features:  ('Temperature', 'Light')
Best F1-score:  0.925
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=31))])


Features:  ('Temperature', 'CO2')
Best F1-score:  0.793
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=1000))])


Features:  ('Temperature', 'HumidityRatio')
Best F1-score:  0.7
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=7))])
```

```
Features:  ('Temperature', 'msm')
Best F1-score:  0.729
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=227))])


Features:  ('Humidity', 'Light')
Best F1-score:  0.963
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Humidity', 'CO2')
Best F1-score:  0.639
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Humidity', 'HumidityRatio')
Best F1-score:  0.374
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=227, weights='distance'))])


Features:  ('Humidity', 'msm')
Best F1-score:  0.308
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=610, weights='distance'))])


Features:  ('Light', 'CO2')
Best F1-score:  0.957
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=19))])


Features:  ('Light', 'HumidityRatio')
Best F1-score:  0.961
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Light', 'msm')
Best F1-score:  0.95
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=84))])


Features:  ('CO2', 'HumidityRatio')
Best F1-score:  0.829
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=372, weights='distance'))])
```

```
Features:  ('CO2', 'msm')
Best F1-score:  0.869
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=84))])


Features:  ('HumidityRatio', 'msm')
Best F1-score:  0.39
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=4))])
```

In [24]:

```python
#batch 2
for cols in feature_combinations[21:42]:  # for each features combination

    print('Features: ', cols)
    features_train = df_train.loc[:, cols]
    model.fit(features_train, target_train)
    print('Best F1-score: ', round(model.best_score_, 3))
    print('Model spec: ', model.best_estimator_, '\n\n')
```

```
Features:  ('Temperature', 'Humidity', 'Light')
Best F1-score:  0.965
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'Humidity', 'CO2')
Best F1-score:  0.744
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Temperature', 'Humidity', 'HumidityRatio')
Best F1-score:  0.69
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=84))])


Features:  ('Temperature', 'Humidity', 'msm')
Best F1-score:  0.696
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=138))])


Features:  ('Temperature', 'Light', 'CO2')
Best F1-score:  0.907
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=84))])


Features:  ('Temperature', 'Light', 'HumidityRatio')
Best F1-score:  0.928
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'Light', 'msm')
Best F1-score:  0.925
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=19))])


Features:  ('Temperature', 'CO2', 'HumidityRatio')
Best F1-score:  0.751
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Temperature', 'CO2', 'msm')
Best F1-score:  0.776
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=1000))])


Features:  ('Temperature', 'HumidityRatio', 'msm')
Best F1-score:  0.701
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=227))])
```

```
Features:  ('Humidity', 'Light', 'CO2')
Best F1-score:  0.922
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn',
                 KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Humidity', 'Light', 'HumidityRatio')
Best F1-score:  0.966
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn',
                 KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Humidity', 'Light', 'msm')
Best F1-score:  0.959
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn',
                 KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Humidity', 'CO2', 'HumidityRatio')
Best F1-score:  0.631
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn',
                 KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Humidity', 'CO2', 'msm')
Best F1-score:  0.698
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn',
                 KNeighborsClassifier(n_neighbors=610, weights='distance'))])


Features:  ('Humidity', 'HumidityRatio', 'msm')
Best F1-score:  0.308
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=227))])


Features:  ('Light', 'CO2', 'HumidityRatio')
Best F1-score:  0.928
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn',
                 KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Light', 'CO2', 'msm')
Best F1-score:  0.95
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=19))])


Features:  ('Light', 'HumidityRatio', 'msm')
Best F1-score:  0.956
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn',
```

```
                    KNeighborsClassifier(n_neighbors=1000, weights='distance'))])
```

```
Features:  ('CO2', 'HumidityRatio', 'msm')
Best F1-score:  0.807
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=227))])


Features:  ('Temperature', 'Humidity', 'Light', 'CO2')
Best F1-score:  0.92
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=610))])
```

In [25]:
```
#batch 3
for cols in feature_combinations[42:]:  # for each features combination

    print('Features: ', cols)
    features_train = df_train.loc[:, cols]
    model.fit(features_train, target_train)
    print('Best F1-score: ', round(model.best_score_, 3))
    print('Model spec: ', model.best_estimator_, '\n\n')
```

```
Features:  ('Temperature', 'Humidity', 'Light', 'HumidityRatio')
Best F1-score:  0.946
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'Humidity', 'Light', 'msm')
Best F1-score:  0.937
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'Humidity', 'CO2', 'HumidityRatio')
Best F1-score:  0.788
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Temperature', 'Humidity', 'CO2', 'msm')
Best F1-score:  0.752
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'Humidity', 'HumidityRatio', 'msm')
Best F1-score:  0.696
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=227))])


Features:  ('Temperature', 'Light', 'CO2', 'HumidityRatio')
Best F1-score:  0.892
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=19))])


Features:  ('Temperature', 'Light', 'CO2', 'msm')
Best F1-score:  0.908
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=84))])


Features:  ('Temperature', 'Light', 'HumidityRatio', 'msm')
Best F1-score:  0.92
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'CO2', 'HumidityRatio', 'msm')
Best F1-score:  0.837
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Humidity', 'Light', 'CO2', 'HumidityRatio')
Best F1-score:  0.916
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=1000, weights='distance'))])
```

```
Features:  ('Humidity', 'Light', 'CO2', 'msm')
Best F1-score:  0.927
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Humidity', 'Light', 'HumidityRatio', 'msm')
Best F1-score:  0.934
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Humidity', 'CO2', 'HumidityRatio', 'msm')
Best F1-score:  0.638
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=227, weights='distance'))])


Features:  ('Light', 'CO2', 'HumidityRatio', 'msm')
Best F1-score:  0.918
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio')
Best F1-score:  0.896
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'Humidity', 'Light', 'CO2', 'msm')
Best F1-score:  0.921
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'Humidity', 'Light', 'HumidityRatio', 'msm')
Best F1-score:  0.959
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'Humidity', 'CO2', 'HumidityRatio', 'msm')
Best F1-score:  0.754
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn',
                KNeighborsClassifier(n_neighbors=1000, weights='distance'))])


Features:  ('Temperature', 'Light', 'CO2', 'HumidityRatio', 'msm')
Best F1-score:  0.886
Model spec:  Pipeline(steps=[('std', StandardScaler()),
               ('knn', KNeighborsClassifier(n_neighbors=31))])


Features:  ('Humidity', 'Light', 'CO2', 'HumidityRatio', 'msm')
```

```
Best F1-score:  0.913
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=610))])


Features:  ('Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio', 'msm')
Best F1-score:  0.919
Model spec:  Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=610))])
```

If you scroll through the printed results, you will notice that the highest model quality according to the F1 score was 96.6% with the feature matrix `['Humidity', 'Light', 'HumidityRatio']`. The only feature that always appears in the feature matrix with a model quality of over 96% is `'Light'`. The best model without `'Light'` only achieves an F1 score of 86.9% with the feature matrix `['CO2', 'msm']`. If you exclude `'Light'`, `'CO2'` and the time of day (`'msm'`), the highest F1 score is 70.9% (`['Temperature', 'Humidity']`).

It seems that the room occupancy can best be determined by a sensor for light and humidity. However, if you want to switch the light on and off automatically, you cannot measure it at the same time. In this case you should rely on a CO2 sensor and a clock.

Congratulations: You got a little impression of how to judge the importance of a feature. You remove the feature and see how this affects the model quality. Similar approaches are used to interpret the predictions, even for complicated models. We now know what the best model looks like and can use it to predict the test data.

Predicting test data

The best k-nearest neighbors model uses the features `['Humidity', 'Light', 'HumidityRatio']` and the maximum neighborhood size we tried out (`n_neighbors=1000`). In return, however, the voices of the neighbors are weighted according to their distance from the data point (`weights='distance'`). Next, we can now use the test data to see if this model is really as good as predicted by the cross validation during the grid search.

Let's start by copying the best model. Instantiate a k-nearest neighbors model called `model` with the best hyperparameter settings (`n_neighbors=1000` and `weights='distance'`). Then define the feature matrix with the three features `['Humidity', 'Light', 'HumidityRatio']` and standardize it. Then train `model` with the standardized feature matrix and the target vector you already created, `target_train`.

In [26]:
```
# instantiate model
model = KNeighborsClassifier(n_neighbors=1000, weights='distance')

# define features matrix
features_train = df_train.loc[:, ['Humidity', 'Light', 'HumidityRatio']]

# standardize features matrix
standardizer = StandardScaler()
```

```
standardizer.fit(df_train.loc[:, ['Humidity', 'Light', 'HumidityRatio']])
features_train_standardized = standardizer.transform(features_train)

# train model
model.fit(features_train_standardized, target_train)
```

Out[26]:
```
KNeighborsClassifier(n_neighbors=1000, weights='distance')
```

Now import the test data from *occupancy_test.txt* and save it in `df_test`. Use it to create a feature matrix, which you should standardize just like you did in the last code cell, and a target vector. Run the following cell:

In [30]:
```
# data gathering
df_test = pd.read_csv('occupancy_test.txt')

# turn date into DateTime (not really necessary)
df_test.loc[:, 'date'] = pd.to_datetime(df_test.loc[:, 'date'])

# turn Occupancy into category (not really necessary)
df_test.loc[:, 'Occupancy'] = df_test.loc[:, 'Occupancy'].astype('category')

# define new feature (not really necessary)
df_test.loc[:, 'msm'] = (df_test.loc[:, 'date'].dt.hour * 60) + df_test.loc[:, 'date']

# features matrix and target vector
features_test = df_test.loc[:, ['Humidity', 'Light', 'HumidityRatio']]
target_test = df_test.loc[:, 'Occupancy']

# standardize features matrix
features_test_standardized = standardizer.transform(features_test)
```

Now use `model` to predict the target data of the test data set and then use `f1_score()` (imported from `sklearn.metrics`) to determine the model quality.

In [31]:
```
from sklearn.metrics import f1_score

target_test_pred = model.predict(features_test_standardized)

f1_score(target_test, target_test_pred)
```

Out[31]:
```
0.9714857428714357
```

According to the test data, an F1 score of 97.1% was achieved, very close to the 96.6% obtained with cross validation.

Congratulations: With an F1 score of 97.1%, you have proven with independent test data that you can predict the presence or absence of people in the room very well. The project managers are very satisfied with your work and wish you every success with this course.

Remember:

- Combine list entries with `itertools.combinations()`
- Feature selection together with grid searches takes a lot of time
- You should always evaluate the best model with independent test data at the end

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.