Grid Search

Module 1 | Chapter 2 | Notebook 7

In this exercise, you will learn about the grid search to optimize hyperparameters. This allows you to find the best middle ground between *bias* and *variance* and to avoid both *underfitting* and *overfitting*. By the end of this lesson you will be able to:

- Create a validation curve with `validation_curve()`
- Find the best hyper parameters with `GridSearchCV`
- Change the weighting of the neighbors of a data point for k-nearest-neighbors classification

---

Bias and variance in k-nearest neighbors

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. The training data is stored in *occupancy_training.txt*. The best F1 score you have achieved so far is 83%. The project management would like you to further increase this model quality so that the pilot project can be a success.

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

In [1]:
```python
# 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]:
```python
# module import
from sklearn.exceptions import DataConversionWarning
import warnings

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

To improve the model quality, we have so far only changed the feature matrix. Alternatively you could also estimate the k-parameter of k-nearest neighbors. Remember that with k=1 only the nearest neighbor is used for the prediction.

With k=3, the setting we have used so far, the three nearest neighbors are used. The majority vote then decides the prediction.

So far we have worked with k=3. How does the model quality change when the neighborhood size is reduced to k=1? Try it yourself by carrying out a cross validating with a `Pipeline` and looking at the prediction quality of the features `'CO2'` and `'msm'` with k=1.

First import everything you need directly:

- `Pipeline` from `sklearn.pipeline`
- `StandardScaler` from `sklearn.preprocessing`
- `KNeighborsClassifier` from `sklearn.neighbors`
- `cross_val_score()` from `sklearn.model_selection`

Then create the feature matrix called `features_train` and the target vector called `target_train`. Then define a `Pipeline` named `pipeline_std_knn` which first uses the `StandardScaler` and then the `KNeighborsClassifier` with `n_neighbors=1`. You can then use this pipeline together with `features_train` and `target_train` to perform a two-fold cross validation. Save the resulting F1 values from each step of the cross validation in `cv_results` and then print `cv_results` afterwards.

In [5]:
```
from sklearn.pipeline import  Pipeline
from sklearn.preprocessing import  StandardScaler
from sklearn.neighbors import  KNeighborsClassifier
from sklearn.model_selection import  cross_val_score

features_train = df_train.loc[:, ['CO2', 'msm']]
target_train = df_train.loc[:, 'Occupancy']

pipeline_std_knn = Pipeline([('std', StandardScaler()),
                             ('knn', KNeighborsClassifier(n_neighbors=1))])

cv_results = cross_val_score(estimator=pipeline_std_knn,#pipeline or Model
                            X=features_train,           #feature matrix
                            y=target_train,             #target values
                            cv=2,                       #number of folds for cross-val
                            scoring='f1',               #scoring function e.g ['accura
                            n_jobs=-1)                  #number of cpu cores to use fo
cv_results.mean()
```

Out[5]:
```
0.8315592684240305
```

The F1 value of 83% (`cv_results.mean()`) does not differ between the k-Nearest-Neighbors model with k=1 and k=3 (83%). Perhaps you will get much better predictions if you use more neighbors for each prediction. Increase the neighborhood (k) to 1,000 (`n_neighbors=1000`). Then calculate the F1 scores of a two-fold cross validation again.

In [22]:
```
pipeline_std_knn = Pipeline([('std', StandardScaler()),
                             ('knn', KNeighborsClassifier(n_neighbors=25))])
```

```
cv_results = cross_val_score(estimator=pipeline_std_knn,#pipeline or Model
                             X=features_train,          #feature matrix
                             y=target_train,            #target values
                             cv=2,                      #number of folds for cross-val
                             scoring='f1',              #scoring function e.g ['accura
                             n_jobs=-1)                 #number of cpu cores to use fo
cv_results.mean()
```

Out[22]:
```
0.8376779860774295
```

The model quality has dropped to 72%, although **more** neighbors were now used to predict each data point. How is that possible?

The number of neighbors (k) affects how influential each individual data point in the training data is. If k=1, each individual data point has the potential to influence the prediction by itself. If k is very large (k=1,000), the influence of a single data point is negligible. Then such a large neighborhood is used for the prediction, that every single point in it gets a bit lost.

So the neighborhood size (k) influences the balance between *bias* and *variance*, see *Too Many Features: Overfitting (Chapter 1)*. With k=1 the predictions change very much if individual data points in the training data set change. So this corresponds to a low *bias* and a high *variance*. If k is very large, the situation is exactly the opposite (high *bias*, low *variance*).

We can also visualize this. In the following figure you can see the actual data points and the predictions of the k-nearest neighbors model at k=1 (upper scatterplot), k=10 (middle scatterplot) and k=100 (lower scatterplot). Each point represents actual measured instances of a person's presence/absence in the room in the training data set. The areas represent the predicted categories. Blue stands for absence and gray for the presence of people in the room.

Note that we only used 6% of the data here to illustrate the influence of the k hyperparameter. The scatter plots showing all the data look a little denser.

Using the x-coordinates showing the CO2 content, we can see that more CO2 is associated with a higher probability of presence (a person supposedly in the room). So on the right in the scatterplot, you can see more gray than on the left. That makes sense, because people produce CO2.

Using the y-coordinates, which indicate the time of day, we can see that people are more likely to be predicted as present in the middle of the day than early or late. This makes sense, because the values were measured in an office where people tend to be present during the day.

In the top scatterplot (k=1) you can see how each data point leads to a prediction of its class in its surroundings. Roughly speaking, a black data point is always black (presence predicted) and a blue data point is always blue (absence predicted). Especially in the middle of the scatterplot you can see how the data points seem to compete for the prediction in their area.

In the middle scatterplot (k=10) the situation is already different. In the middle you can see isolated blue dots, which no longer lead to a predicted absence in their surroundings, because they are overruled by adjacent black dots. The reduced influence of individual data points reduces the *variance*.

In the bottom scatter plot (k=100) this situation is even more extreme. There are no more blue islands, only very rough prediction regions. A data point's exact location is not significant for its prediction. This indicates a very low *variance*, since the predictions would barely be shifted by slight changes in the data.

This dynamic can be summarized very nicely by what's called a validation curve. We want to test the following neighborhood sizes:

In [29]:
```python
import numpy as np
k = np.geomspace(1, 1000, 15, dtype='int')
k = np.unique(k)
k
```

Out[29]:
```
array([   1,    2,    4,    7,   11,   19,   31,   51,   84,  138,  227,
        372,  610, 1000])
```

Note that we will create values with increasingly large gaps between them using `numpy.geomspace()`. This is because we can assume that the difference between 1 and 2 data points in the neighborhood is greater than the difference between 998 and 999 neighbors. So that each value in `k` tests something new enough, the differences between the k values become larger and larger.

The question now is how the F1 score changes with increasing neighborhood size, which determines the classifications. To create the validation curve, we'll use `validation_curve()`. You don't need to understand the exact syntax for this function exactly. we can redefine `pipeline_std_knn` here. However, this variable should have the same functionality as it did so far.

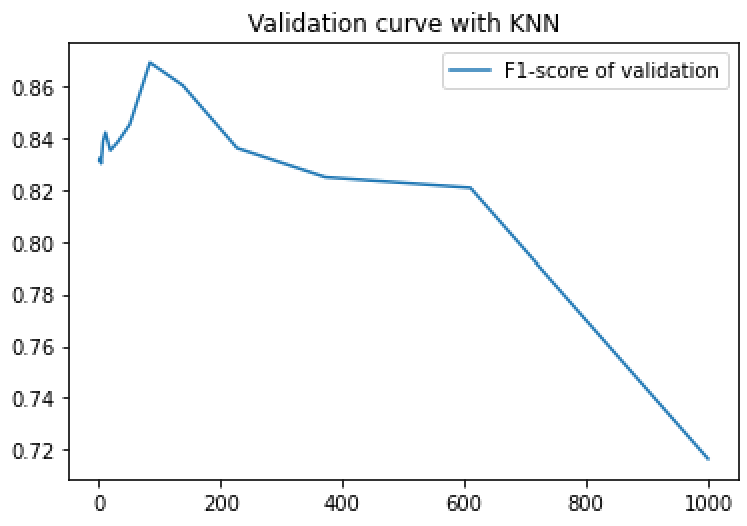
Tip: Be patient with the following code cell. It may well take a minute to complete.

In [26]:
```python
# define pipeline to ensure standardization of features matrix
pipeline_std_knn = Pipeline([('std', StandardScaler()),
                             ('knn', KNeighborsClassifier())])

# calculate training and validation scores
from sklearn.model_selection import validation_curve
train_scores, valid_scores = validation_curve(estimator=pipeline_std_knn,  # estimator
                                              X=features_train,  # feature matrix
                                              y=target_train,  # target vector
                                              param_name='knn__n_neighbors',  # define
                                              param_range=k,  # test these k-values
                                              cv=2,  # two-fold cross-validation
                                              scoring='f1', # use F1-score for validat
                                              n_jobs=-1) # use all available Cores to

# visualise validation curve
F1 = pd.DataFrame({'F1-score of validation' : np.mean(valid_scores, axis=1)}, index=k)
```

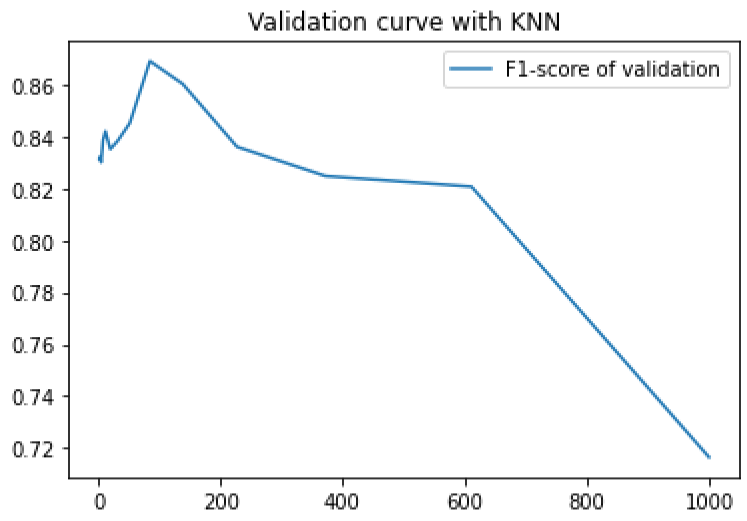
```
F1.plot(title='Validation curve with KNN')
F1
```

Out[26]:

	F1-score of validation
1	0.831559
2	0.832266
4	0.830085
7	0.838477
11	0.842026
19	0.835128
31	0.838297
51	0.845156
84	0.869043
138	0.860221
227	0.836042
372	0.824809
610	0.820728
1000	0.716510



Now we can see in the validation curve that the F1 score is highest at a k value of 84 (approx. 87%). At lower values for k (further left in the line chart), the model suffers from overfitting (*variance* too high), while at higher values for k, (further right in the line chart) it suffers from underfitting (*bias* too high).



Congratulations: You have learned how the k parameter of k nearest neighbors affects the model quality. values that are too small or too large are bad for predictions. The best value for k is typically in the middle, but you don't know where in advance. That depends on the data.

Now you could find this value by looking at the validation curves each time. Alternatively, you can let Python do that and just take the best value. We'll look at how to do this now.

Finding optimal hyperparameters with a grid search

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

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

You can determine the details of the first two steps (finding model type and hyperparameter values) in a data-driven way thanks to cross validation. In this lesson we will start with the hyperparameter k of k-nearest neighbors. How large should the optimal neighborhood be? We can use what's known as a grid search with `GridSearchCV` from the `sklearn.model_selection` to help us determine this. Import `GridSearchCV` directly.

In [ ]:
```
from sklearn.model_selection import GridSearchCV
```

A grid search goes through all the specified hyperparameters that you want to try. To gain a better understanding, let's recreate the search for the optimal k parameter for k-nearest neighbors, which we performed above with the validation curve.

In this case, the grid of the grid search is a one-dimensional row whose values we have already defined in the variable `k`:

Now we can create a `dict` with (`{'key': value}`) in the variable `search_space` . The `dict` contains the hyperparameters to be changed (*key*), together with the values to be tried out (*value*).

The *key* specifies both the pipeline step and, after two underscores, the function's argument: `<pipeline_step>__<argument` >. We have given the steps of the pipeline names such as `'knn'` . `knn__n_neighbors` is therefore the `n_neighbors` argument of `KNeighborsClassifier()` ; a function which is called `knn` in the pipeline.

In [31]:
```
search_space = {'knn__n_neighbors': k}
```

Now we can use `GridSearchCV()` together with `pipeline_std_knn` and `search_space` . We want to use two-fold cross-validations again during the grid search. First instantiate the grid search in a variable called `grid_search_for_k` . Use the following outline:

```
GridSearchCV(estimator=Pipeline or Model,  # estimator
             param_grid=dict,              # the grid of the grid search
             scoring=str,                  # which measure to optimise on
             cv=int,                       # number of folds during cross-
validation
             n_jobs=int)                   # number of CPU cores to use
(use -1 for all cores)
```

In [39]:
```
grid_search_for_k = GridSearchCV(estimator=pipeline_std_knn,  # estimator
             param_grid=search_space,                        # the grid of the grid s
             scoring='f1',                                   # which measure to optim
             cv=2,                                           # number of folds during
             n_jobs=-1)                                      # number of CPU cores to
grid_search_for_k
```

```
{'knn__n_neighbors': array([   1,    2,    4,    7,   11,   19,   31,   51,   84,  13
8,  227,
        372,  610, 1000])}
```

Then we can fit this variable to the data. In our grid search, this corresponds to finding the best k value for k-nearest neighbors. Use the `my_GridSearchCV.fit()` method together with `features_train` and `target_train` . The method therefore follows the convention we already know from machine learning models.

Tip: Be patient with the following code cell.

In [40]:
```
grid_search_for_k.fit(features_train, target_train)
```

Out[40]:
```
GridSearchCV(cv=2,
             estimator=Pipeline(steps=[('std', StandardScaler()),
                                       ('knn', KNeighborsClassifier())]),
             n_jobs=-1,
             param_grid={'knn__n_neighbors': array([   1,    2,    4,    7,   11,   1
9,   31,   51,   84,  138,  227,
        372,  610, 1000])},
             scoring='f1')
```

The `grid_search_for_k` variable now has some interesting attributes. As usual with `sklearn`, all results specified by training a model are written with an underscore at the end of an attribute.

For example, there is the `my_GridSearchCV.best_score_` attribute, which determines the best model quality score achieved during the grid search.

In [41]:
```
grid_search_for_k.best_score_
```

Out[41]:
```
0.8690425507372819
```

87% is the highest F1 score achieved by the two-fold cross validation. What exactly did the model that achieved this value look like? Print the `my_GridSearchCV.best_estimator_` attribute of `grid_search_for_k` to find out.

In [42]:
```
grid_search_for_k.best_estimator_
```

Out[42]:
```
Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=84))])
```

Each step of the pipeline is now broken down in detail. We didn't look at many of the parameters because the defaults are already very good and don't need to be changed. But you also see `n_neighbors=84`. The optimal value for k is therefore 84. This is the same value that we took from the validation curve.

At the very end, the algorithm then fitted the best model to all the data. So training and validation data were merged again and then a k-nearest neighbors model with k=84 was fitted to the data. `grid_search_for_k` now behaves like this best model. So it has a `my_model.predict()` method, which we can use to make predictions.

Congratulations: You have found the best value for k with a one-dimensional grid search. But there are other hyperparameters that we can change to potentially improve the model quality even more. We will look at another hyperparameter of k-Nearest-Neighbors next.

Finding optimal hyperparameters with a grid search

So far, all k-neighbours of a data point have been assigned the same weight for the model quality. This is specified with the `weights` parameter of `KnighborsClassifier()`. The default is `'uniform'`.

The only alternative is `'distance'`. With this setting, neighbors that are closer to the data point are given more weight than data points that are further away. It could be that this setting improves the model quality. Let's try it out.

Define a grid search that goes through all combinations of neighborhood sizes as defined in the `k` variable, and `weights` settings. Schematically that would look like this:

Create the variable `search_space_grid` as a `dict` with two *keys*. Just like before, the first *key* should refer to the `n_neighbors` parameter of the step named `'knn'` in `pipeline_std_knn`. The *value*, i.e. the values to be tried out, is the list `k`.

The second *key* is the `weights` argument of the `knn` step of `pipeline_std_knn`. This *key* is also assigned a list. It should contain the possible *strings* that `weights` can take: `['uniform', 'distance']`.

Create `search_space_grid` in this way to prepare a grid search through `n_neighbors` and `weights`.

In [43]:
```
search_space_grid = {'knn__n_neighbors' :  k ,
                     'knn__weights': ['uniform', 'distance']}
```

Now use `search_space_grid` and `pipeline_std_knn` to perform a grid search with two-fold cross validation together with `GridSearchCV()`, which finds the best model according to the F1 score. This model should be stored in the variable `model`.

Tip: You first have to instantiate this model named `model` with `GridSearchCV()` and then match it to the data with `my_model.fit()`.

In [ ]:
```
model = GridSearchCV(estimator=pipeline_std_knn,                  # estimator
             param_grid=search_space_grid,                       # the grid of the grid s
             scoring='f1',                                       # which measure to optim
             cv=2,                                               # number of folds during
             n_jobs=-1)                                          # number of CPU cores to

model.fit(features_train, target_train)
```

Out[ ]:
```
GridSearchCV(cv=2,
             estimator=Pipeline(steps=[('std', StandardScaler()),
                                       ('knn', KNeighborsClassifier())]),
             n_jobs=-1,
             param_grid={'knn__n_neighbors': array([   1,    2,    4,    7,   11,   1
9,   31,   51,   84,  138,  227,
        372,  610, 1000]),
                         'knn__weights': ['uniform', 'distance']},
             scoring='f1')
```

What is the best model now? Print the model specification of the best model according to the grid search.

In [48]:
```
model.best_estimator_
```

Out[48]:
```
Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=84))])
```

In [49]:
```
model.best_score_
```

Out[49]:
```
0.8690425507372819
```

The model with 84 equally-weighted neighbors wins. It still has the best F1 score. At the moment it seems that the best k-nearest neighbors model has an F1 value of 87%.

Congratulations: You have learned how to find the optimal hyperparameters in a data-driven way using a grid search. You were able to increase the F1 score to 87%. The project managers are already quite satisfied with this value.

They now just want a summary of which sensors are really needed in a room to best determine whether a person is in it. This means that we have to determine which features we want to use. For each feature selection, we have to use a grid search to find the optimal hyperparameters, which then lead to an F1 score determined by cross validation. Now we'll look at how to achieve this.

Remember:

- A grid search tries out hyperparameter settings and evaluates them with cross validation
- `GridSearchCV` carries out grid search and behaves like a model once instantiated

---

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

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.