

# Classification Recap

Module 1 | Chapter 3 | Notebook 1

---

In this notebook we'll recap the most important things from the previous chapter. This is all repetition. If you discover a lot of new things here, we recommend that you take a look back at Chapter 2. The relevant lessons for each section are clearly marked.

---

## The five steps of sklearn

For a classification algorithm like k-nearest neighbors, you go through the same five steps, see *Simple Linear Regression with sklearn (Module 1 Chapter 1)*.

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

The only classification model we have seen so far is k-nearest neighbors, see *k-Nearest Neighbors (Module 1 Chapter 2)*.

```
In [1]: # model import
        from sklearn.neighbors import KNeighborsClassifier
```

The hyperparameter `k` is expressed with the function parameter `n_neighbors`. It expresses how many neighbors are used to determine the class of a data point, see *k-Nearest Neighbors (Module 1 Chapter 2)*.

You can also use the function parameter `weights` to determine whether all neighbors are equally weighted ( `weights='uniform'` ) or whether closer neighbors should be given more weight than distant neighbors ( `weights='distance'` ), see *Grid Search (chapter 2)*.

You can determine the optimal hyperparameter values using a grid search with cross validation, see *Grid Search (Module 1 Chapter 2)*. In this case you first choose settings based on your gut feeling to explain the algorithm.

```
In [18]: model = KNeighborsClassifier(n_neighbors=5, weights='uniform')
```

As with linear regression, you need training data ( `df_train` ), divided into a feature matrix ( `features_train` ) and a target vector ( `target_train` ), to train the model.

```
In [3]: # 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[:, 'date'].dt.minute

# features matrix
features_train = df_train.loc[:, ['CO2', 'HumidityRatio']]

# target vector
target_train = df_train.loc[:, 'Occupancy']
```

To ensure that the neighborhood proximity is the same in every dimension, the feature matrix needs to be standardized, see *Regularization (Module 1 Chapter 1)*. This means that all features have the same mean value and standard deviation.

```
In [4]: # model import
from sklearn.preprocessing import StandardScaler

# model instantiation
scaler = StandardScaler()

# model fitting and data transformation
features_train_standardized = scaler.fit_transform(features_train) # only fit to training data
```

Then we can now move on to *model fitting*.

```
In [5]: model.fit(features_train_standardized, target_train)
```

```
Out[5]: KNeighborsClassifier()
```

When predicting new values, you should make sure that the features are standardized again with the same settings as the training data before, remember **Only ever fit to the training set!**, see *k-Nearest Neighbors (Module 1 Chapter 2)*.

```
In [6]: # features data
df_aim = pd.DataFrame({'CO2':[420, 10000],
                       'HumidityRatio':[0.0038, 0.005]})

# standardize features
features_aim_standardized = scaler.transform(df_aim)

# predict target values
target_aim_pred = model.predict(features_aim_standardized)
target_aim_pred
```

```
Out[6]: array([0, 1])
```

**Congratulations:** You have recapped the general procedure for the k-Nearest Neighbors model. Now we'll recap how to evaluate this kind of model.

## Cross validation

In order to build an optimal model, you need to be clear about what the model should do. There are three established model quality metrics to evaluate classification models:

1. The recall, which is the percentage of cases that is actually positive and that are identified as such
2. The precision, which is the proportion of positively classified cases which are actually positive
3. The F1 score, which is the harmonic mean between recall and precision



It's ok if you're confused at this point. We recommend taking a look at the lesson *Evaluating Classification Performance (Module 1 Chapter 2)*. We'll select the F1 score here. So we want a model that detects positive cases as well as possible and whose predictions of positive cases are trustworthy.

As with linear regression, you shouldn't use training data to determine the model quality, see *Too many features: Overfitting (Module 1 Chapter 1)*. Instead, you divide the training data set into a validation data set and a training data set in the strictest sense. You only use the latter when fitting the model.

By using multiple runs, each part is sometimes used for validation and sometimes it's used for model fitting. This procedure is known as **cross validation**, see *Evaluating Models with Cross Validation (Module 1 Chapter 2)*.

**Attention:** Determining model quality using cross validation is generally a good idea for supervised learning algorithms. This includes k-nearest neighbors as well as linear regression.

In principle, you can also carry out cross validation with `for` loops. However, it is more convenient to use the `cross_val_score()` function provided for this purpose, see *Introduction to Pipelines (Module 1 Chapter 2)*.

```
In [7]: # import function
        from sklearn.model_selection import cross_val_score
```

To ensure that the feature matrix is standardized in the correct way, we need the `Pipeline` data type. This controls the order of the data processing steps, creating a machine learning model as the end result.

```
In [8]: # import
        from sklearn.pipeline import Pipeline
```

You pass `Pipeline()` a list with pairs consisting of a name you define for the processing step yourself and the function. Function parameters that you set at this point can be changed later.

In our case, the `Pipeline` consists of the feature matrix standardization ( `'std'` ) and k-nearest neighbors ( `'knn'` ).

```
In [9]: pipeline_std_knn = Pipeline([('std', StandardScaler()),
                                     ('knn', KNeighborsClassifier(n_neighbors=5,
                                                                weights='uniform'))])
```

You can now use this kind of `Pipeline` as if it were the model itself. For example, it also has a `my_model.fit()` method and then a `my_model.predict()` method. But we will use the `Pipeline` here for cross validation.

Note that the following code uses the unstandardized feature matrix. The standardization is part of `pipeline_std_knn`. We'll choose a two-fold cross validation ( `cv=2` ) to save time.



2 slices of two-fold cross validation

Higher `cv` values often provide more stable and realistic results, but require more patience.

```
In [10]: cv_results = cross_val_score(estimator=pipeline_std_knn,
                                     X=features_train,
                                     y=target_train,
                                     cv=2,
                                     scoring='f1',
                                     n_jobs=-1)

cv_results
```

```
Out[10]: array([0.46083737, 0.80021657])
```

We get an F1 score for each pass of the cross validation. Their mean is the expected model quality according to cross validation.

```
In [11]: cv_results.mean()
```

```
Out[11]: 0.6305269678388766
```


**Congratulations:** You have recapped how to use a pipeline as part of a cross validation with `cross_val_score()`. Now you can realistically assess how good your model actually is. Next you will recap how to use this knowledge to find the optimal hyperparameters.

## Selecting optimal hyperparameters.

You could now go through every promising combination of hyperparameter values with `for` loops and use `cross_val_score()` to determine how good the model is with these settings

each time. You could then use the hyperparameter settings with the best results for your model.

This kind of procedure is called a **grid search**. The grid is the combination of hyperparameter settings. In our case, it looks like this:

 k and weights search space

But instead of working with `for` loops, you should use the `GridSearchCV` data type, see *Grid Search (Chapter 2)*. It carries out a grid search and is clearer and faster than using nested `for` loops.

```
In [12]: # import
from sklearn.model_selection import GridSearchCV
```

**Important:** A grid search is generally suitable for finding the optimal hyperparameters. This doesn't just apply to classification models, but also to regression models.

We store the grid we want to search in the variable `search_space`. `GridSearchCV()` requires a `dict {'key': value}`. 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` parameter of `KNeighborsClassifier()`; a function that has the name `'knn'` in the pipeline.

```
In [13]: # module import
import numpy as np
k = np.geomspace(1, 1000, 15, dtype='int') # quadratic series of whole numbers
k = np.unique(k) # extract unique numbers

search_space = {'knn__n_neighbors': k,
                'knn__weights': ['uniform', 'distance']}
```

Now we can define the grid search:

```
In [14]: model_grid_cv = GridSearchCV(estimator=pipeline_std_knn,
                                     param_grid=search_space,
                                     scoring='f1',
                                     cv=2,
                                     n_jobs=-1)
```

If you now perform the grid search with `my_model.fit()`, it tries out all the hyperparameter settings and calculates the mean F1 score with two-fold cross validation.

The model with the highest F1 value is finally fitted to the data without cross validation. Then `model_grid_cv` automatically uses the settings from the best model at the end and can be used to make predictions.

```
In [15]: model_grid_cv.fit(features_train, target_train)
```

```
Out[15]: 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')
```

Now you can look at the best model using the `my_model.best_estimator_` attribute.

```
In [16]: model_grid_cv.best_estimator_
```

```
Out[16]: Pipeline(steps=[('std', StandardScaler()),
                        ('knn',
                         KNeighborsClassifier(n_neighbors=372, weights='distance'))])
```

In our case it is a model with 372 neighbors with influence weighted by distance. Now it also has the `my_model.predict()` method. This can now be used directly to make a prediction.

```
In [17]: target_aim_pred = model_grid_cv.predict(features_aim_standardized)
target_aim_pred
```

```
Out[17]: array([0, 1])
```

Our model has predicted that at the point in time when the data in the first row of `df_aim` was recorded, the room was empty and that there were people in the room at the second point in time.

**Congratulations:** You have recapped how to find the optimal hyperparameters using a grid search. We have recapped the most important points from the previous chapter and can now turn to new material.

### Remember:

- `from sklearn.neighbors import KNeighborsClassifier`
- Recall: the percentage of cases that is actually positive and that are identified as such
- Precision: the proportion of positively classified cases which are actually positive
- Use a grid search to find optimal hyperparameter settings.

---

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](mailto:support@stackfuel.com).

---