Evaluating Models with Cross Validation

Module 1 | Chapter 2 | Notebook 5

In this lesson you will learn how to apply the model quality metrics for classification algorithms directly to the training data set without risking overfitting. Cross validation helps us to evaluate supervised learning models. By the end of this lesson you will be able to:

- Extract validation data from training data
- Use `KFold`
- Perform cross validation with a `for` loop

Train validation split

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 project managers would like to know whether the F1 score of 61% could possibly be increased. In this lesson we will try to do this by doubling the number of features.

Let's import the training data now 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

So far we have used the training data all for training the model. But this also means that we had no data left that the model hasn't seen yet to determine the model quality metrics.

If you calculate the model quality measures with data that the model trained on, it's possible that the model could just predict data that it has learned off by heart. See *Too many features: Overfitting (Modul 1 Chapter 1)*. Cross validation separates validation data from the training data and uses it to determine the quality of the model. For example, you could use half of the training data actually for training and the other half for evaluation. Schematically it looks like this:

In the visualization, we have now used the first half to train the model. However, there could be some peculiarities with this, which might give a skewed idea of the model quality. This is why it's good to work in two steps. In the first step, the second half of the data is used for training and

the first half for validation, in the second step the first half is used for training and the second half for validation. Schematically, that looks like this:

At each step, the model quality metrics are calculated and then the mean value of these is calculated to get an overall picture. In this context, this is called cross validation. It's up to you how many parts, known as *folds*, the data set is divided into. This number is indicated by k. In our case k=2. In our case, we speak of two-fold cross validation, or more generally of k-fold cross validation.

Three-fold cross validation would look like this schematically:

Because you train the model and validate it three times for a three-fold cross validation, it also takes more time than a two-fold cross-validation. In return, the machine learning model with three-fold cross-validation has more data available for training. To speed things up, we have decided to use two-fold cross validation here.

In Python we use `KFold` from the `sklearn.model_selection` module to divide the data into training data in the strict sense and validation data. Import `KFold` directly.

In [2]:
```
from sklearn.model_selection import KFold
```

First you instantiate a variable that stores the cross-validation settings. The parameter k is expressed with the argument `n_splits`.

In [3]:
```
kf = KFold(n_splits=2)
```

`kf` now has data type `KFold`. You can see for yourself by using the `type()` function with `kf`.

In [4]:
```
type(kf)
```

Out[4]:
```
sklearn.model_selection._split.KFold
```

Variables with the `KFold` data type actually only have one method, which is important to us: `my_kfold.split()`. You can use this method on the training data `df_train`.

In [5]:
```
kf.split(df_train)
```

Out[5]:
```
<generator object _BaseKFold.split at 0x7f412d349c10>
```

This creates what's called a `generator object`. A `generator object` contains instructions for data generation. So far, you have probably only seen this kind of `generator object` with `range()`, which provides instructions to generate a sequence of numbers.

In our case, `kf` does not generate a sequence of numbers, but the **row indices** of the training data in the strict sense and the validation data. A `generator object` is normally used in a loop to generate new data at each loop iteration.

Important: `my_kfold.split()` generates `array` pairs. You can see this very clearly if you use `list()` to force the `generator object` to generate all its data immediately. At each loop iteration you can then assign each element of the pair to a different iterator variable.

In [6]:
```
list(kf.split(df_train))
```

Out[6]:
```
[(array([4072, 4073, 4074, ..., 8140, 8141, 8142]),
  array([   0,    1,    2, ..., 4069, 4070, 4071])),
 (array([   0,    1,    2, ..., 4069, 4070, 4071]),
  array([4072, 4073, 4074, ..., 8140, 8141, 8142]))]
```

Specifically, you would use `my_kfold.split()` in a `for` loop with two iterator variables (`train_index` and `val_index`) like this:

In [7]:
```
step = 0  # set counter to 0
for train_index, val_index in kf.split(df_train):  # for each fold
    step = step + 1  # update counter

    print('Step ', step)
    print("TRAIN:", train_index)
    print("VALIDATE:", val_index, '\n')
```

```
Step  1
TRAIN: [4072 4073 4074 ... 8140 8141 8142]
VALIDATE: [   0    1    2 ... 4069 4070 4071]

Step  2
TRAIN: [   0    1    2 ... 4069 4070 4071]
VALIDATE: [4072 4073 4074 ... 8140 8141 8142]
```

We can see that in the first step, the second half of the rows (4072 up to and including 8142) should be used for training the model and the first half of the rows (0 up to and including 4071) for validation. In the second step, the two halves should be swapped around.

Now we can use the resulting row indices to create a feature matrix and target vector at each step.

Important: `kf` creates **row indices** for the actual training data set and the validation data set. So it doesn't use row names!

In [8]:
```
step = 0  # set counter to 0
for train_index, val_index in kf.split(df_train):  # for each fold
    step = step + 1  # update counter

    print('Step ', step)

    features_fold_train = df_train.iloc[train_index, [4, 5]]  # features matrix of tra
    features_fold_val = df_train.iloc[val_index, [4, 5]]  # features matrix of validat
```

```
    target_fold_train = df_train.iloc[train_index, 6]  # target vector of training dat
    target_fold_val = df_train.iloc[val_index, 6]  # target vector of validation data

    print("TRAIN:", train_index)
    print('Dimensions of features matrix for training: ',features_fold_train.shape)
    print("VALIDATE:", val_index)
    print('Dimensions of features matrix for validation: ', features_fold_val.shape, '
```

```
Step  1
TRAIN: [4072 4073 4074 ... 8140 8141 8142]
Dimensions of features matrix for training:  (4071, 2)
VALIDATE: [   0    1    2 ... 4069 4070 4071]
Dimensions of features matrix for validation:  (4072, 2)

Step  2
TRAIN: [   0    1    2 ... 4069 4070 4071]
Dimensions of features matrix for training:  (4072, 2)
VALIDATE: [4072 4073 4074 ... 8140 8141 8142]
Dimensions of features matrix for validation:  (4071, 2)
```

Congratulations: You have divided the training data into two halves that can be used as training and validation data. Next, we can use our previous knowledge of classification and the relevant model quality metrics to know how good our model actually is, thanks to cross-validation.

Determining model quality metrics with cross validation

Remember the five steps for model predictions:

1. Select model type
2. Instantiate model with k=3
3. Divide (and possibly standardize) data into a feature matrix and target vector
4. Model fitting
5. Make predictions with the trained model

The cross-validation takes effect from step 3. So before that we can do step 1 and 2 as usual. Import `KNeighborsClassifier` directly from `sklearn.neighbors` and instantiate the model with k=3 (`n_neighbors` parameter). Store the instantiated model in the variable `model` .

In [9]:
```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=3)
```

We'll carry out the cross validation with a `for` loop again. For each loop iteration, the data should be divided into training and validation data again and then the model quality metrics should be calculated. So before you program the loop, you should import the functions for the model quality metrics:

- `recall_score()`

- `precision_score()`
- `f1_score()`

We know from the last lesson that *recall* expresses the hit rate, which is the percentage of times with people in the room that are correctly identified. *Precision* describes the relevance, i.e., the proportion of times a person is predicted to be present in a room when there is actually a person in the room. The F1 score is the harmonic mean of *recall* and *precision*.

Import the functions that calculate these three model quality measures directly from `sklearn.metrics`.

In [15]:
```
from sklearn.metrics import recall_score, precision_score, f1_score
```

There's one thing we've forgotten before we can start: standardization. In the lesson *k-Nearest Neighbors*, you learned that all features have to use the same scale, so that no single one is is favored during classification with k-Nearest-Neighbors. So import `StandardScaler` directly from `sklearn.preprocessing` and instantiate a variable `scaler` with `StandardScaler()`.

In [16]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

Now we have everything well prepared to perform steps 3 to 5 in a `for` loop:

1. Divide (and possibly standardize) data into a feature matrix and target vector
2. Model fitting with `my_model.fit()`
3. Make predictions with the trained model with `my_model.predict()`

You can use the `for` loop you recently wrote to create the feature matrix and target vector for the training data and validation data for each cross-validation step.

After that you should use `scaler` to standardize the feature matrices `features_fold_train` and `features_fold_va` in the same way. You can achieve this by first setting the default settings with `scaler.fit(features_fold_train)` and then applying them to both `features_fold_train` and `features_fold_val` with `my_scaler.transform()`. You can then name the standardized feature matrices `features_fold_train_standardized` and `features_fold_val_standardized`.

Next you can adjust the model to the data (`features_fold_train_standardized` and `target_fold_train`) and then predict the occupancy data from the validation set (`model.predict(features_fold_val_standardized)`). You should store the resulting predicted data in `target_fold_val_pred`.

The actual occupancy data (`target_fold_val`) and its predictions (`target_fold_val_pred`) can be then finally be used to determine the model quality metrics. Print the model quality metrics (recall, precision, F1 score) for each loop iteration, so for each step of the cross validation.

Important: The order of the arguments you pass is important for the model quality metrics. In `sklearn` they are defined in such a way that the actual target vector (in this case `target_fold_val`) must be passed first and the predicted target vector (in this case `target_fold_val_pred`).

In [20]:
```
step = 0
for train_index, val_index in kf.split(df_train):  # for each fold
    step = step + 1
    print('Step ', step)

    features_fold_train = df_train.iloc[train_index, [4, 5]]  # features matrix of tra
    features_fold_val = df_train.iloc[val_index, [4, 5]]  # features matrix of validat

    target_fold_train = df_train.iloc[train_index, 6]  # target vector of training dat
    target_fold_val = df_train.iloc[val_index, 6]  # target vector of validation data


    scaler.fit(features_fold_train)
    features_fold_train_standardized = scaler.transform(features_fold_train)
    features_fold_val_standardized = scaler.transform(features_fold_val)

    model.fit(features_fold_train_standardized, target_fold_train)
    target_fold_val_pred = model.predict(features_fold_val_standardized)

    print('recall :',    recall_score(target_fold_val, target_fold_val_pred))
    print('precision :', precision_score(target_fold_val, target_fold_val_pred))
    print('f1 :',        f1_score(target_fold_val, target_fold_val_pred))
    print('/n')
```

```
Step  1
recall : 0.49605609114811566
precision : 0.771117166212534
f1 : 0.6037333333333332
/n
Step  2
recall : 0.8996598639455783
precision : 0.66125
f1 : 0.7622478386167146
/n
```

You should get the following result:

Cross validation step	*recall*	*precision*	F1
1	50%	77%	60%
2	90%	66%	76%
Mean	**70%**	**72%**	**68%**

What do these values express? 70% of the instances with people in the room were identified as such. 72% of positive predictions (a person is supposedly in the room) are correct. To summarize, the prediction quality in the positive cases (person in the room) is 68%.

Congratulations: You've carried out your first cross validation. You first used the first half of the data to determine the model quality of a k-nearest neighbors model after the model was fitted

to the second half of the data. Then you used the second half of the data to determine the quality of the model (trained on the first half). This time we did two-fold cross-validation to keep things simple. Alternatively, you can also do three-fold, or in general k-fold cross validation. The principles are the same. Next, we'll modify the model and see if this also changes the model quality metrics.

Feature Selection - selecting additional features

To improve the model quality, it may be worthwhile increasing the number of features. Use the following columns to predict whether a person is in the room:

- `'Humidity'` (column index 2)
- `'CO2'` (column index 4)
- `'HumidityRatio'` (column index 5)
- `'msm'` (column index 7)

Follow the same five steps again:

1. Select model type
2. Instantiate model with k=3
3. Divide data into a feature matrix and target vector (and standardize if necessary)
4. Model fitting
5. Make predictions with the trained model

We don't need to carry out the first two steps because we can use the model we've already instantiated (`model`). Program a `for` loop to go through the cross validation steps. Use `kf` for that. The feature matrix and target vector of the training and validation data should be created at each loop iteration. Remember that the target vector is the `'Occupancy'` column (column number 6).

The two feature matrices should then be standardized with the same parameters before being used to train and evaluate the model.

Print the *recall*, *precision* model quality metrics and the F1 score at the end. Is the model with twice as many features better?

Tip: You can ignore the warnings that `int` numbers had to be turned into `float` numbers.

In [21]:
```
step = 0
for train_index, val_index in kf.split(df_train):  # for each fold
    step = step + 1
    print('Step ', step)

    features_fold_train = df_train.iloc[train_index, [2,4, 5,7]]  # features matrix of
    features_fold_val = df_train.iloc[val_index, [2,4, 5,7]]  # features matrix of val

    target_fold_train = df_train.iloc[train_index, 6]  # target vector of training dat
    target_fold_val = df_train.iloc[val_index, 6]  # target vector of validation data
```

```
    scaler.fit(features_fold_train)
    features_fold_train_standardized = scaler.transform(features_fold_train)
    features_fold_val_standardized = scaler.transform(features_fold_val)

    model.fit(features_fold_train_standardized, target_fold_train)
    target_fold_val_pred = model.predict(features_fold_val_standardized)

    print('recall :',    recall_score(target_fold_val, target_fold_val_pred))
    print('precision :', precision_score(target_fold_val, target_fold_val_pred))
    print('f1 :',        f1_score(target_fold_val, target_fold_val_pred))
    print('/n')
```

```
Step  1
recall : 0.33917616126205086
precision : 0.8486842105263158
f1 : 0.48465873512836577
/n
Step  2
recall : 0.9081632653061225
precision : 0.7437325905292479
f1 : 0.8177641653905054
/n
```

You should get following result:

Cross validation step	*recall*	*precision*	F1
1	34%	85%	48%
2	91%	74%	82%
Mean	**63%**	**80%**	**65%**

What do these values express? Compared to our two feature model, our four feature model is **worse** at identifying points in time when people are in the room correctly (*recall* 63% < 70%). However, at the same time, predictions that a person is in the room are more likely to be trusted (*precision* 80% > 72%). Our four-feature model is therefore more cautious in its positive predictions (when somebody is supposed to be in the room) and is therefore less likely to detect actual people in the room. At the same time, however, when it did predict a person present, there was more likely to actually be somebody in the room.

Which model you choose depends entirely on what is more important to you: that all actually positive instances are identified as positive or that positive predictions are correct. Since the project managers are focused on the F1 score, both aspects appear to be equally important. According to this metric, the model with two features is slightly better (65% < 68%).

Congratulations: Thanks to cross validation you now know the strengths and weaknesses of two k-Nearest Neigbors models. Once a model has been selected based on this, training and validation data can be combined and the final model can be calculated. Since the two-feature model was the best, we already did that in the previous exercise. So we don't need to repeat that here.

Instead, we'll next learn how to represent the data processing steps we wrote in the `for` loop more elegantly as a *pipeline*. This makes your own code clearer and helps prevent programming errors.

Remember:

- Evaluating models with cross validation prevents overfitting
- Carry out k-fold cross validation with `KFold`
- You can freely choose the number of folds of the training data set between 2 and the number of data points in the dataset

---

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.