Introduction to Pipelines

Module 1 | Chapter 2 | Notebook 6

In this lesson you will learn how to carry out cross validation more quickly and elegantly by using pipelines. We will also use the pipeline concept for the following lesson. So it's useful to look at the basics here. By the end of this lesson you will be able to:

- Summarize data processing steps with `Pipeline`
- Perform cross validation quickly with `cross_val_score`

Pipelines

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 65% from the last lesson could possibly be improved upon. In that lesson you used the features `'Humidity'`, `'CO2'`, `'HumidityRatio'`, and `'msm'`. In this lesson we will try to achieve better prediction performance by using different combinations of 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

Before we turn to the problems facing the *smart-building* provider and model evaluation with cross validation in more detail, we should first take a look at pipelines. As the name suggests, pipelines are a sequence of data processing steps. A pipeline summarizes steps that would otherwise take many lines of code.

They also have the huge advantage that you can pass them to functions such as `cross_val_score()` to perform cross-validation quickly. But we'll get to that later in the lesson.

The `Pipeline` data type from the `sklearn.pipeline` module implements pipelines. Import `Pipeline` directly.

In [2]:
```
from sklearn.pipeline import Pipeline
```

It is best to create a simple pipeline first, so that the principle and syntax are clear. Our pipeline should standardize the feature matrix `features_train`. Import `StandardScaler` directly

from `sklearn.preprocessing`.

In [3]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

Split the data into a feature matrix `features_train` (`'CO2'` and `'msm'` columns) and a target vector `target_train` (`'Occupancy'` column).

In [4]:
```
features_train = df_train.loc[:, ['CO2', 'msm']]
target_train  = df_train.loc[:, 'Occupancy']
```

If we now wanted to standardize the feature matrix, we would have to proceed as follows.

In [9]:
```
#ignore warnings
from sklearn.exceptions import DataConversionWarning
import warnings
warnings.filterwarnings(action='ignore', category=DataConversionWarning)

scaler = StandardScaler()  # instantiate scaler
scaler.fit_transform(features_train)  # train scaler and transform features matrix
```

Out[9]:
```
array([[ 0.36494808,  0.84069022],
       [ 0.34188106,  0.84069022],
       [ 0.34029023,  0.84538594],
       ...,
       [ 0.61073113, -0.33558572],
       [ 0.68019732, -0.33089   ],
       [ 0.68231843, -0.32854215]])
```

You can do the same in just one step by using a pipeline.

In [10]:
```
pipeline_std = Pipeline([('std', StandardScaler())])  # instantiate pipeline with one
pipeline_std.fit_transform(features_train)  # train pipeline (scaler) and transform fe
```

Out[10]:
```
array([[ 0.36494808,  0.84069022],
       [ 0.34188106,  0.84069022],
       [ 0.34029023,  0.84538594],
       ...,
       [ 0.61073113, -0.33558572],
       [ 0.68019732, -0.33089   ],
       [ 0.68231843, -0.32854215]])
```

Let's take a closer look at what `Pipeline()` does. It expects a `list`. Don't forget to use square brackets `[]`. The `list` contains the data processing steps one after the other. Each step is specified as a pair - name and function. Pairs are enclosed in round brackets `()` and entries are separated by a comma, e.g. `('std', StandardScaler())`.

Every step of the pipeline apart from the last one must be a *transformer*, so it must have a `.fit()` and a `.transform()` method. The last step is the only exception. So far we have only encountered `StandardScaler` as a *transformer* data type. But you will learn and use more of them throughout this course. When the pipeline is executed, the `.fit()` and `.transform()` methods are executed one after the other, processing the data step by step.

The pipeline as a whole has the methods of the final step, which does not necessarily have to be a *transformer*, as mentioned above. It could also be a classification model, for example. This would then look something like this:

In [11]:
```
from sklearn.neighbors import KNeighborsClassifier
pipeline_std_knn = Pipeline([('std', StandardScaler()),
                             ('knn', KNeighborsClassifier(n_neighbors=3))])
pipeline_std_knn.fit(features_train, target_train)
```

Out[11]:
```
Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=3))])
```

`pipeline_std_knn` now contains two steps called `'std'` and `'knn'`. The first step is a scaler and the second is a k-nearest neighbors model. You can now use `pipeline_std_knn` like a normal k-nearest neighbors model, which standardizes its feature matrix during training, and uses it to make predictions, for example. The following code recreates the pipeline without the `pipeline` data type, the same processing steps are performed.

In [8]:
```
model_knn = KNeighborsClassifier(n_neighbors=3)  # instantiate k-Nearest-Neighbors mod
model_knn.fit(scaler.fit_transform(features_train),  # model fitting with standardised
              target_train)
```

Out[8]:
```
KNeighborsClassifier(n_neighbors=3)
```

You can now use `pipeline_std_knn` and `model_knn` exactly the same way. They would make the same predictions.

If you can recreate a pipeline this easily, why do you need to know the `Pipeline` data type? Because you can pass it to other functions, such as `cross_val_score()`, as we will see in a moment.

Congratulations: You have now got to know a new data type called `Pipeline`. This may not seem very useful to you now. But now we will use it right away to get faster results with cross validation.

Cross validation with cross_val_score()

In the last lesson we did cross validation with `KFold` and `for` loops. This was very useful to understand the principle. However, in your work as a data scientist, it's likely that you will prefer to use specialized functions such as `cross_val_score()` from the `sklearn.model_selection` module, because it is more handy.

Import `cross_val_score()` directly.

In [12]:
```
from sklearn.model_selection import cross_val_score
```

`cross_val_score()` requires several inputs:

- an *estimator*, something with a `.fit()` method, like `pipeline_std_knn` or even `model_knn`
- a feature matrix (`features_train`)
- a target vector (`target_train`)
- a model quality metric
- a specification for the cross validation

For model quality metrics, you can use one of these *strings*:

- `'accuracy'`
- `'recall'`
- `'precision'`
- `'f1'`

You can specify the cross validation simply by using an `int` number, which indicates the number of folds in the training data.

Now we are ready and can perform the cross validation with `cross_val_score()` . Specify a two-fold cross validation here. Use `pipeline_std_knn` so that the feature matrix is standardized. Store the F1 scores in the variable `cv_results` . Then print `cv_results` .

Tip 1: Use the following outline:

```
cv_results = cross_val_score(estimator=var_pipeline,#pipeline or Model
                             X=DataFrame or Matrix,  #feature matrix
                             y=Series or Array,      #target values
                             cv=int,                 #number of folds for
cross-validation
                             scoring=str             #scoring function e.g
['accuracy','f1','recall']
                             n_jobs=-1)              #number of cpu cores to
use for computation (use -1 to use all available cores)
```

Tip 2: You can ignore or mute the warnings that `int` numbers are converted into `float` numbers.

In [17]:
```
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
```

Out[17]:
```
array([0.84221526, 0.81694731])
```

A great result. An average F1 score of 83% is the highest value so far (`cv_results`). Only the model with the features `'CO2'` and `'msm'` seems to provide better predictions than the previous models with the features `'CO2'` and `'HumidityRatio'` or `'Humidity'` , `'CO2'` , `'HumidityRatio'` and `'msm'` .

You may now be wondering whether `cross_val_score()` has performed the standardization in `pipeline_std_knn` in such a way that at each step of the cross validation, the standardization settings were only taken from the actual training data. At each step, these settings should then be applied to both the actual training data and the validation data (see *Evaluating Models with Cross Validation*).

As you can read here in the `sklearn` documentation, `cross_val_score` is smart enough to do this correctly. At each step of the cross validation, only the actual training data is used to find the settings for all the *transformers* in the pipeline (e.g. `StandardScaler`). Then they are applied to this data as well as the validation data.

Congratulations: You have learned how to use a pipeline as part of a cross validation with `cross_val_score()`. This means you need much less code, which reduces the number of errors that can occur. It also gives you the possibility to find the optimal hyper parameter settings of your classifier with a grid search. As we will see in the following lesson, we can increase the model quality even more by doing this.

Remember:

- You can chain together *transformers* like `Standard Scaler`, potentially with an *estimator* like `NeighborsClassifier`
- `Pipeline` expects a list of `('name', transformer or estimator)` pairs
- `cross_val_score()` performs cross validation in one line of code

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.