02_numerical_pipeline_scaling

May 29, 2021

1 Preprocessing for numerical features

In this notebook, we will still use only numerical features.

We will introduce these new aspects:

- an example of preprocessing, namely **scaling numerical variables**;
- using a scikit-learn **pipeline** to chain preprocessing and model training;
- assessing the statistical performance of our model via **cross-validation** instead of a single train-test split.

1.1 Data preparation

First, let's load the full adult census dataset.

[1]:
```
import pandas as pd

adult_census = pd.read_csv("../datasets/adult-census.csv")
```

[2]:
```
# to display nice model diagram
from sklearn import set_config
set_config(display='diagram')
```

We will now drop the target from the data we will use to train our predictive model.

[3]:
```
target_name = "class"
target = adult_census[target_name]
data = adult_census.drop(columns=target_name)
```

Then, we select only the numerical columns, as seen in the previous notebook.

[4]:
```
numerical_columns = [
    "age", "capital-gain", "capital-loss", "hours-per-week"]

data_numeric = data[numerical_columns]
```

Finally, we can divide our dataset into a train and test sets.

[5]:
```
from sklearn.model_selection import train_test_split
```

```
data_train, data_test, target_train, target_test = train_test_split(
    data_numeric, target, random_state=42)
```

1.2 Model fitting with preprocessing

A range of preprocessing algorithms in scikit-learn allow us to transform the input data before training a model. In our case, we will standardize the data and then train a new logistic regression model on that new version of the dataset.

Let's start by printing some statistics about the training data.

[6]:
```
data_train.describe()
```

[6]:
```
                age  capital-gain  capital-loss  hours-per-week
count  36631.000000  36631.000000  36631.000000    36631.000000
mean      38.642352   1087.077721     89.665311       40.431247
std       13.725748   7522.692939    407.110175       12.423952
min       17.000000      0.000000      0.000000        1.000000
25%       28.000000      0.000000      0.000000       40.000000
50%       37.000000      0.000000      0.000000       40.000000
75%       48.000000      0.000000      0.000000       45.000000
max       90.000000  99999.000000   4356.000000       99.000000
```

We see that the dataset's features span across different ranges. Some algorithms make some assumptions regarding the feature distributions and usually normalizing features will be helpful to address these assumptions.

Tip

Here are some reasons for scaling features:

Models that rely on the distance between a pair of samples, for instance k-nearest neighbors, should be trained on normalized features to make each feature contribute approximately equally to the distance computations.

Many models such as logistic regression use a numerical solver (based on gradient descent) to find their optimal parameters. This solver converges faster when the features are scaled.

Whether or not a machine learning model requires scaling the features depends on the model family. Linear models such as logistic regression generally benefit from scaling the features while other models such as decision trees do not need such preprocessing (but will not suffer from it).

We show how to apply such normalization using a scikit-learn transformer called `StandardScaler`. This transformer shifts and scales each feature individually so that they all have a 0-mean and a unit standard deviation.

We will investigate different steps used in scikit-learn to achieve such a transformation of the data.

First, one needs to call the method `fit` in order to learn the scaling from the data.

[7]:
```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
scaler.fit(data_train)
```

```
[7]: StandardScaler()
```

The `fit` method for transformers is similar to the `fit` method for predictors. The main difference is that the former has a single argument (the data matrix), whereas the latter has two arguments (the data matrix and the target).

In this case, the algorithm needs to compute the mean and standard deviation for each feature and store them into some NumPy arrays. Here, these statistics are the model states.

Note

The fact that the model states of this scaler are arrays of means and standard deviations is specific to the StandardScaler. Other scikit-learn transformers will compute different statistics and store them as model states, in the same fashion.

We can inspect the computed means and standard deviations.

```
[8]: scaler.mean_
```

```
[8]: array([  38.64235211, 1087.07772106,   89.6653108 ,   40.43124676])
```

```
[9]: scaler.scale_
```

```
[9]: array([  13.72556083, 7522.59025606,  407.10461772,   12.42378265])
```

Note

scikit-learn convention: if an attribute is learned from the data, its name ends with an underscore (i.e. *), as in mean* and scale_ for the StandardScaler.

Scaling the data is applied to each feature individually (i.e. each column in the data matrix). For each feature, we subtract its mean and divide by its standard deviation.

Once we have called the `fit` method, we can perform data transformation by calling the method `transform`.

```
[10]: data_train_scaled = scaler.transform(data_train)
      data_train_scaled
```

```
[10]: array([[ 0.17177061, -0.14450843,  5.71188483, -2.28845333],
             [ 0.02605707, -0.14450843, -0.22025127, -0.27618374],
             [-0.33822677, -0.14450843, -0.22025127,  0.77019645],
             …,
             [-0.77536738, -0.14450843, -0.22025127, -0.03471139],
             [ 0.53605445, -0.14450843, -0.22025127, -0.03471139],
             [ 1.48319243, -0.14450843, -0.22025127, -2.69090725]])
```

Let's illustrate the internal mechanism of the `transform` method and put it to perspective with what we already saw with predictors.

The `transform` method for transformers is similar to the `predict` method for predictors. It uses a predefined function, called a **transformation function**, and uses the model states and the input data. However, instead of outputting predictions, the job of the `transform` method is to output a transformed version of the input data.

Finally, the method `fit_transform` is a shorthand method to call successively `fit` and then `transform`.

[11]:
```
data_train_scaled = scaler.fit_transform(data_train)
data_train_scaled
```

[11]:
```
array([[ 0.17177061, -0.14450843,  5.71188483, -2.28845333],
       [ 0.02605707, -0.14450843, -0.22025127, -0.27618374],
       [-0.33822677, -0.14450843, -0.22025127,  0.77019645],
       …,
       [-0.77536738, -0.14450843, -0.22025127, -0.03471139],
       [ 0.53605445, -0.14450843, -0.22025127, -0.03471139],
       [ 1.48319243, -0.14450843, -0.22025127, -2.69090725]])
```

[12]:
```
data_train_scaled = pd.DataFrame(data_train_scaled,
                                 columns=data_train.columns)
data_train_scaled.describe()
```

[12]:
```
                age  capital-gain  capital-loss  hours-per-week
count  3.663100e+04  3.663100e+04  3.663100e+04    3.663100e+04
mean  -1.263553e-16 -1.708425e-15 -1.652358e-15    1.146502e-16
std    1.000014e+00  1.000014e+00  1.000014e+00    1.000014e+00
min   -1.576792e+00 -1.445084e-01 -2.202513e-01   -3.173852e+00
25%   -7.753674e-01 -1.445084e-01 -2.202513e-01   -3.471139e-02
50%   -1.196565e-01 -1.445084e-01 -2.202513e-01   -3.471139e-02
75%    6.817680e-01 -1.445084e-01 -2.202513e-01    3.677425e-01
max    3.741752e+00  1.314865e+01  1.047970e+01    4.714245e+00
```

We can easily combine these sequential operations with a scikit-learn `Pipeline`, which chains together operations and is used as any other classifier or regressor. The helper function `make_pipeline` will create a `Pipeline`: it takes as arguments the successive transformations to perform, followed by the classifier or regressor model.

[13]:
```
import time
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

model = make_pipeline(StandardScaler(), LogisticRegression())
model
```

[13]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('logisticregression', LogisticRegression())])
```

The `make_pipeline` function did not require us to give a name to each step. Indeed, it was automatically assigned based on the name of the classes provided; a StandardScaler will be a step named "standardscaler" in the resulting pipeline. We can check the name of each steps of our model:

[14]:
```
model.named_steps
```

[14]:
```
{'standardscaler': StandardScaler(),
 'logisticregression': LogisticRegression()}
```

This predictive pipeline exposes the same methods as the final predictor: fit and predict (and additionally predict_proba, decision_function, or score).

[15]:
```
start = time.time()
model.fit(data_train, target_train)
elapsed_time = time.time() - start
```

We can represent the internal mechanism of a pipeline when calling fit by the following diagram:

When calling model.fit, the method fit_transform from each underlying transformer (here a single transformer) in the pipeline will be called to:

- learn their internal model states
- transform the training data. Finally, the preprocessed data are provided to train the predictor.

To predict the targets given a test set, one uses the predict method.

[16]:
```
predicted_target = model.predict(data_test)
predicted_target[:5]
```

[16]:
```
array([' <=50K', ' <=50K', ' >50K', ' <=50K', ' <=50K'], dtype=object)
```

Let's show the underlying mechanism:

The method transform of each transformer (here a single transformer) is called to preprocess the data. Note that there is no need to call the fit method for these transformers because we are using the internal model states computed when calling model.fit. The preprocessed data is then provided to the predictor that will output the predicted target by calling its method predict.

As a shorthand, we can check the score of the full predictive pipeline calling the method model.score. Thus, let's check the computational and statistical performance of such a predictive pipeline.

[17]:
```
model_name = model.__class__.__name__
score = model.score(data_test, target_test)
print(f"The accuracy using a {model_name} is {score:.3f} "
      f"with a fitting time of {elapsed_time:.3f} seconds "
      f"in {model[-1].n_iter_[0]} iterations")
```

```
The accuracy using a Pipeline is 0.807 with a fitting time of 0.061 seconds in
12 iterations
```

We could compare this predictive model with the predictive model used in the previous notebook which did not scale features.

[18]:
```
model = LogisticRegression()
start = time.time()
model.fit(data_train, target_train)
elapsed_time = time.time() - start
```

[19]:
```
model_name = model.__class__.__name__
score = model.score(data_test, target_test)
print(f"The accuracy using a {model_name} is {score:.3f} "
      f"with a fitting time of {elapsed_time:.3f} seconds "
      f"in {model.n_iter_[0]} iterations")
```

```
The accuracy using a LogisticRegression is 0.807 with a fitting time of 0.135
seconds in 59 iterations
```

We see that scaling the data before training the logistic regression was beneficial in terms of computational performance. Indeed, the number of iterations decreased as well as the training time. The statistical performance did not change since both models converged.

Warning

Working with non-scaled data will potentially force the algorithm to iterate more as we showed in the example above. There is also the catastrophic scenario where the number of required iterations are more than the maximum number of iterations allowed by the predictor (controlled by the max_iter) parameter. Therefore, before increasing max_iter, make sure that the data are well scaled.

1.3 Model evaluation using cross-validation

In the previous example, we split the original data into a training set and a testing set. This strategy has several issues: in a setting where the amount of data is small, the subset used to train or test will be small. Besides, a single split does not give information regarding the confidence of the results obtained.

Instead, we can use cross-validation. Cross-validation consists of repeating the procedure such that the training and testing sets are different each time. Statistical performance metrics are collected for each repetition and then aggregated. As a result we can get an estimate of the variability of the model's statistical performance.

Note that there exists several cross-validation strategies, each of them defines how to repeat the `fit`/`score` procedure. In this section, we will use the K-fold strategy: the entire dataset is split into `K` partitions. The `fit`/`score` procedure is repeated `K` times where at each iteration `K - 1` partitions are used to fit the model and `1` partition is used to score. The figure below illustrates this K-fold strategy.

Note

This figure shows the particular case of K-fold cross-validation strategy. As mentioned earlier, there are a variety of different cross-validation strategies. Some of these aspects will be covered in more details in future notebooks.

For each cross-validation split, the procedure trains a model on all the red samples and evaluate the score of the model on the blue samples. Cross-validation is therefore computationally intensive because it requires training several models instead of one.

In scikit-learn, the function `cross_validate` allows to do cross-validation and you need to pass it the model, the data, and the target. Since there exists several cross-validation strategies, `cross_validate` takes a parameter `cv` which defines the splitting strategy.

[20]:
```
%%time
from sklearn.model_selection import cross_validate

model = make_pipeline(StandardScaler(), LogisticRegression())
cv_result = cross_validate(model, data_numeric, target, cv=5)
cv_result
```

```
CPU times: user 620 ms, sys: 1.11 s, total: 1.73 s
Wall time: 433 ms
```

[20]:
```
{'fit_time': array([0.06472039, 0.06283402, 0.06517816, 0.06270933,
0.06275654]),
 'score_time': array([0.0131216 , 0.01336575, 0.01297641, 0.01278067,
0.01289797]),
 'test_score': array([0.79557785, 0.80049135, 0.79965192, 0.79873055,
0.80436118])}
```

The output of `cross_validate` is a Python dictionary, which by default contains three entries: (i) the time to train the model on the training data for each fold, (ii) the time to predict with the model on the testing data for each fold, and (iii) the default score on the testing data for each fold.

Setting `cv=5` created 5 distinct splits to get 5 variations for the training and testing sets. Each training set is used to fit one model which is then scored on the matching test set. This strategy is called K-fold cross-validation where `K` corresponds to the number of splits.

Note that by default the `cross_validate` function discards the 5 models that were trained on the different overlapping subset of the dataset. The goal of cross-validation is not to train a model, but rather to estimate approximately the generalization performance of a model that would have been trained to the full training set, along with an estimate of the variability (uncertainty on the generalization accuracy).

You can pass additional parameters to `cross_validate` to get more information, for instance training scores. These features will be covered in a future notebook.

Let's extract the test scores from the `cv_result` dictionary and compute the mean accuracy and the variation of the accuracy across folds.

[21]:
```
scores = cv_result["test_score"]
print("The mean cross-validation accuracy is: "
      f"{scores.mean():.3f} +/- {scores.std():.3f}")
```

```
The mean cross-validation accuracy is: 0.800 +/- 0.003
```

Note that by computing the standard-deviation of the cross-validation scores, we can estimate the uncertainty of our model statistical performance. This is the main advantage of cross-validation and can be crucial in practice, for example when comparing different models to figure out whether one is better than the other or whether the statistical performance differences are within the uncertainty.

In this particular case, only the first 2 decimals seem to be trustworthy. If you go up in this notebook, you can check that the performance we get with cross-validation is compatible with the one from a single train-test split.

In this notebook we have:

- seen the importance of **scaling numerical variables**;
- used a **pipeline** to chain scaling and logistic regression training;
- assessed the statistical performance of our model via **cross-validation**.