Optional: Defining Your Own Evaluation Metrics

Module 1 | Chapter 4 | Notebook 6

This notebook is optional. You don't have to work through it to complete this training successfully. It will give you the chance to practice your previous knowledge of simple linear regression, model quality metrics and residual plots. If you are still unsure about these things, this notebook can help you.

Building a custom scorer to predict integer values

Scenario: 1001Wines is an online retailer that sells wines through its website. Recently, 1001Wines took over a start-up company that specializes in producing wines synthetically. They want to use this expertise to create a separate product range. The customers of 1001Wines leave ratings for the products. This data will be used to predict how well new wines will be accepted by customers in the future.

Since you already created a good product recommendation algorithm for 1001Wines (see *Module 0, Chapter 2*), they came back to you for more help. They want you to analyze the composition of the wines in view of their ratings. They provided you with the respective wine characteristics in the file *wine_composition.csv*. The corresponding ratings are in the file *clustering_data-2.csv*. Both files are already in the current working directory.

In the last lesson, you used principal component analysis as part of a `Pipeline`. We'll do this again here. We'll also look at a little trick that can help to improve the error metrics for ratings. Run the following cell to import the data so that you can work with it.

In [1]:
```
import pandas as pd
df_composition = pd.read_csv('wine_composition.csv')  # read the composition of the wi
df_rating = pd.read_csv('wine_rating.csv')  # read the ratings for the wines

target = df_rating.loc[:, 'rating']  # do not use id for target variable
features = df_composition.iloc[:, :-2]  # only use numeric values as features

df_composition.head()
```

Out[1]:

	fixed.acidity	volatile.acidity	citric.acid	residual.sugar	chlorides	free.sulfur.dioxide	total.sulfur.dio
0	7.0	0.27	0.36	20.7	0.045	45.0	1
1	6.3	0.30	0.34	1.6	0.049	14.0	1
2	8.1	0.28	0.40	6.9	0.050	30.0	
3	7.2	0.23	0.32	8.5	0.058	47.0	1
4	7.2	0.23	0.32	8.5	0.058	47.0	1

The following data dictionary explains what the data means.

Column number	Column name	Type	Description
0	`'fixed.acidity'`	continuous (`float`)	liquid acid content.
1	`'volatile.acidity'`	continuous (`float`)	gaseous acid content, particularly relevant for the odor.
2	`'citiric.acid'`	continuous (`float`)	citric acid content. Part of the liquid acids. Affects freshness in taste.
3	`'residual.sugar'`	continuous (`float`)	Natural sugar of the grapes, which is still present at the end of fermentation stage.
4	`'chlorides'`	continuous (`float`)	chloride content. Minerals, which can be dependent on the wine growing region.
5	`'free.sulfur.dioxide'`	continuous (`float`)	unbound sulfur dioxide. Perceptible by smell.
6	`'total.sulfur.dioxide'`	continuous (`float`)	total sulfur dioxide content
7	`'density'`	continuous (`float`)	density of the wine
8	`'PH'`	continuous (`float`)	pH value. Determined by the acid content.
9	`'sulphates'`	continuous (`float`)	sulphate content.
10	`'alcohol'`	continuous (`float`)	alcohol content
11	`'color'`	categorical	color of the wine. Contains only `'red'` and `'white'`
12	`'id'`	categorical (`int`)	Unique identification number of the wine.

The ratings are currently on a 10-level scale. However, our data only contains the values 3 to 9. We have always predicted continuous values. But our target variable isn't continuous. For example, if we predict the value `5.4` but the real value is `5`, we get an unnecessary error of

`0.4`. This error does not really help us to evaluate how good our model is. It may therefore make sense to round the predictions to integer values. Let's see how that affects our error.

To do this, first define the `Pipeline` so that `PCA` reduces the dimensions to 26 components. The `Pipeline` should therefore contain the following steps: `PolynomialFeatures(degree=2, include_bias=False)` -> `StandardScaler>()` - `PCA>(n_components=26)` - `LinearRegression>()`.

In [2]:
```
import numpy as np

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression

poly_transformer = PolynomialFeatures(degree=2, include_bias=False)
scaler = StandardScaler()
pca = PCA(n_components=26)
model = LinearRegression()
pipeline = Pipeline([('poly',poly_transformer ),
                     ('scale', scaler ),
                     ('pca', pca),
                     ('reg', model)])
```

To make the rounding off part of the pipeline, we have to build our own `sklean` transformer and integrate this into the pipeline. If you want to learn more about this, feel free to suggest the topic for the next webinar. For now, we'll simply define a new *scorer* that rounds off the pipeline predictions and uses the rounded predictions to evaluate our model.

We can then use this scorer within `cross_val_score` or `GridSearchCV`.

First, we should define a function that takes `target_pred` and `target` as parameters, rounds `target_pred` and then determines the *mean absolute error* compared to `target`. The head of the function should look like this:

```
def rounded_scoring(target , target_pred):
    """ Round Predictions and Calculate mean_absolute_error

    Args:
        target (ndarray): True target values
        target_pred (ndarray): Predicted target values

    Returns:
        float: Mean absolute error

    """
```

In [3]:
```
from sklearn.metrics import mean_absolute_error

def rounded_scoring(target , target_pred):
    """ Round Predictions and Calculate mean_absolute_error

    Args:
```

```
        target (ndarray): True target values
        target_pred (ndarray): Predicted target values

    Returns:
        float: Mean absolute error

    """
    target_pred = np.round(target_pred,0)
    mae = mean_absolute_error(target,target_pred)
    return mae
```

Run the following cell to test your function. You should get a value of approx. `5.05`:

In [4]:
```
pipeline.fit(features,target)
target_pred = pipeline.predict(features)
rounded_scoring(target, target_pred)
```

Out[4]:
```
0.5051562259504386
```

Scorers in `sklearn` are classes that have a `.score()` method. When evaluating an estimator (e.g. our `Pipeline`) `sklearn` calls this `.score()` function of the *scorer* and returns the result. But how do you define classes and methods? Fortunately, you don't need to worry about this, but can use a support function from `sklearn` that builds a *scorer* object from our `rounded_scoring` function. You can find the function `make_scorer` in `sklearn.metrics`.

Here is a brief overview of the most important parameters:

```
make_scorer(score_func=function, # Function that returns our metric with
the following structure: score_func(y, y_pred, **kwargs)
            greater_is_better=bool # Metric should be maximized (True), or
minimized (False)
           )
```

Now use `make_scorer` to create the scorer class `rounded_mae`.

Tip: Our metric, the *mean absolute error* is better the smaller its value is, so you should specify `greater_is_better=False`.

In [5]:
```
from sklearn.metrics import make_scorer
rounded_mae = make_scorer(rounded_scoring,
                          greater_is_better=False
                         )
```

Now we can use our scorer to evaluate our model quality. Use the `cross_val_score()` function, which you encountered in *Pipelines Introduction (Module 1 Chapter 2)*. The function is located in the sklearn module `sklearn.model_selection`. Use them to perform a five-fold cross validation. You have to assign the `rounded_mae` class to the `scoring` parameter.

Then print the mean of all the results of `cross_val_score()`

In [12]:
```
from sklearn.model_selection import cross_val_score
cross_val_score(estimator=pipeline,
                X = features,
```

```
                y = target,
                scoring=rounded_mae,
                cv = 5).mean()
```

Out[12]:
```
-0.5256157991354297
```

We get a value around of approximately 0.526. We have already reduced the error by about 10% with this trick! However, this does not represent a real improvement in prediction. We have simply adapted the model quality measure to our problem.

Are there values for which the prediction is particularly difficult? To find this out, let's look at the prediction errors in relation to the target value. First fit the `Pipeline` to all the data points. Then generate a prediction for all the data points and save these as `target_pred` and round them off to produce integer values. Then create a new column called `'residuals'` in `df_rating`, which consists of the prediction errors, i.e. `target_pred - target`.

In [13]:
```
pipeline.fit(features, target)
target_pred = np.round(pipeline.predict(features), 0)
df_rating.loc[:,'residuals'] = target_pred - target
```

Now group `df_rating` by `'rating'` and aggregate the data with its median to see the typical error for each rating.

In [14]:
```
df_rating.groupby('rating').median()
```

Out[14]:

rating	id	residuals
3	3176.0	2.0
4	2502.5	1.0
5	3440.5	1.0
6	3348.5	0.0
7	3096.0	-1.0
8	2795.0	-2.0
9	827.0	-2.0

We see that the low ratings (3, 4, 5) are mostly predicted a little too high (positive residuals) and the high ratings are mostly predicted a little too low (negative residuals). All in all, we have now created features and filtered the most important information with the principal component analysis. But our prediction has not improved much except for the little trick with the rounding. Why could that be?

Perhaps polynomial regression is not best suited for this prediction problem. We can only find this out by trying out other prediction algorithms. However, it can also simply be the case that the data does not provide much more. For example, the chemical properties measured may not cover all the properties required for the flavor. Or there is too much noise in the data. The

ratings are relatively subjective and we do not know whether some wines received a lot of ratings or very few. If one wine only received only a handful of ratings, these don't yet provide a reliable indication of whether a large number of people find the wine good or bad.

In these cases, even feature engineering will not help much. But you don't always need your predictions to be perfect. Most of the time, a first model is already much better than having to make decisions based on gut feeling.

Congratulations: 1001Wines says thanks you for developing a model for predicting the wine ratings according to their chemical composition so quickly. They are very satisfied with it and will use it to assess the quality of synthetically produced wines. Next, we'll look at how principal component analysis can help us when it comes to visualizing high-dimensional data.

Remember:

- Every data science project needs a suitable scorer.
- Thanks to `sklearn`, writing a scorer class is easier.
- Before thinking about potential improvements to a prediction model for too long, it is important to understand exactly what the data you have can actually provide.

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: P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.
