linear_models_ex_02

June 24, 2024

1 Exercise M4.02

In the previous notebook, we showed that we can add new features based on the original feature `x` to make the model more expressive, for instance `x ** 2` or `x ** 3`. In that case we only used a single feature in `data`.

The aim of this notebook is to train a linear regression algorithm on a dataset with more than a single feature. In such a "multi-dimensional" feature space we can derive new features of the form `x1 * x2`, `x2 * x3`, etc. Products of features are usually called "non-linear" or "multiplicative" interactions between features.

Feature engineering can be an important step of a model pipeline as long as the new features are expected to be predictive. For instance, think of a classification model to decide if a patient has risk of developing a heart disease. This would depend on the patient's Body Mass Index which is defined as `weight / height ** 2`.

We load the dataset penguins dataset. We first use a set of 3 numerical features to predict the target, i.e. the body mass of the penguin.

Note

If you want a deeper overview regarding this dataset, you can refer to the Appendix - Datasets description section at the end of this MOOC.

[14]:
```python
import pandas as pd

penguins = pd.read_csv("../datasets/penguins.csv")

columns = ["Flipper Length (mm)", "Culmen Length (mm)", "Culmen Depth (mm)"]
target_name = "Body Mass (g)"

# Remove lines with missing values for the columns of interest
penguins_non_missing = penguins[columns + [target_name]].dropna()

data = penguins_non_missing[columns]
target = penguins_non_missing[target_name]
data.head()
```

[14]:

	Flipper Length (mm)	Culmen Length (mm)	Culmen Depth (mm)
0	181.0	39.1	18.7

1	186.0	39.5	17.4
2	195.0	40.3	18.0
4	193.0	36.7	19.3
5	190.0	39.3	20.6

Now it is your turn to train a linear regression model on this dataset. First, create a linear regression model.

[15]:
```python
from sklearn.linear_model import LinearRegression

linear_regression = LinearRegression()
```

Execute a cross-validation with 10 folds and use the mean absolute error (MAE) as metric.

[16]:
```python
from sklearn.model_selection import cross_validate

results = cross_validate(linear_regression, data, target, scoring =␣
 ↪"neg_mean_absolute_error", return_train_score=True, n_jobs=2, verbose=1,␣
 ↪cv=10)
```

```
[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=2)]: Done  10 out of  10 | elapsed:    1.2s finished
```

Compute the mean and std of the MAE in grams (g). Remember you have to revert the sign introduced when metrics start with `neg_`, such as in `"neg_mean_absolute_error"`.

[17]:
```python
print(f"{-results['test_score'].mean()} +- {results['test_score'].std()}")
```

```
337.07133738443895 +- 84.86840942516221
```

Now create a pipeline using `make_pipeline` consisting of a `PolynomialFeatures` and a linear regression. Set `degree=2` and `interaction_only=True` to the feature engineering step. Remember not to include a "bias" feature (that is a constant-valued feature) to avoid introducing a redundancy with the intercept of the subsequent linear regression model.

You may want to use the `.set_output(transform="pandas")` method of the pipeline to answer the next question.

[18]:
```python
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(degree=2, interaction_only=True)
model = make_pipeline(poly_features, linear_regression).
 ↪set_output(transform="pandas")
model
```

[18]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(interaction_only=True)),
                ('linearregression', LinearRegression())])
```

Transform the first 5 rows of the dataset and look at the column names. How many features are generated at the output of the `PolynomialFeatures` step in the previous pipeline?

[20]:
```python
model.fit(data, target)
model[0].transform(data[:5])
```

[20]:
```
     1  Flipper Length (mm)  Culmen Length (mm)  Culmen Depth (mm)  \
0  1.0                181.0                39.1               18.7
1  1.0                186.0                39.5               17.4
2  1.0                195.0                40.3               18.0
4  1.0                193.0                36.7               19.3
5  1.0                190.0                39.3               20.6

   Flipper Length (mm) Culmen Length (mm)  \
0                                  7077.1
1                                  7347.0
2                                  7858.5
4                                  7083.1
5                                  7467.0

   Flipper Length (mm) Culmen Depth (mm)  Culmen Length (mm) Culmen Depth (mm)
0                                3384.7                                 731.17
1                                3236.4                                 687.30
2                                3510.0                                 725.40
4                                3724.9                                 708.31
5                                3914.0                                 809.58
```

Check that the values for the new interaction features are correct for a few of them.

[24]:
```python
data.iloc[0][0]*data.iloc[0][1]
```

[24]:
```
7077.1
```

Use the same cross-validation strategy as done previously to estimate the mean and std of the MAE in grams (g) for such a pipeline. Compare with the results without feature engineering.

[25]:
```python
results_with_p = cross_validate(model, data, target, scoring =
    "neg_mean_absolute_error", return_train_score=True, n_jobs=2, verbose=1,
    cv=10)
print(f"{-results['test_score'].mean()} +- {results['test_score'].std()}")
print(f"{-results_with_p['test_score'].mean()} +- {results_with_p['test_score'].
    std()}")
```

```
[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent workers.

337.07133738443895 +- 84.86840942516221
301.78955228431437 +- 44.34000781934426

[Parallel(n_jobs=2)]: Done  10 out of  10 | elapsed:    1.2s finished
```

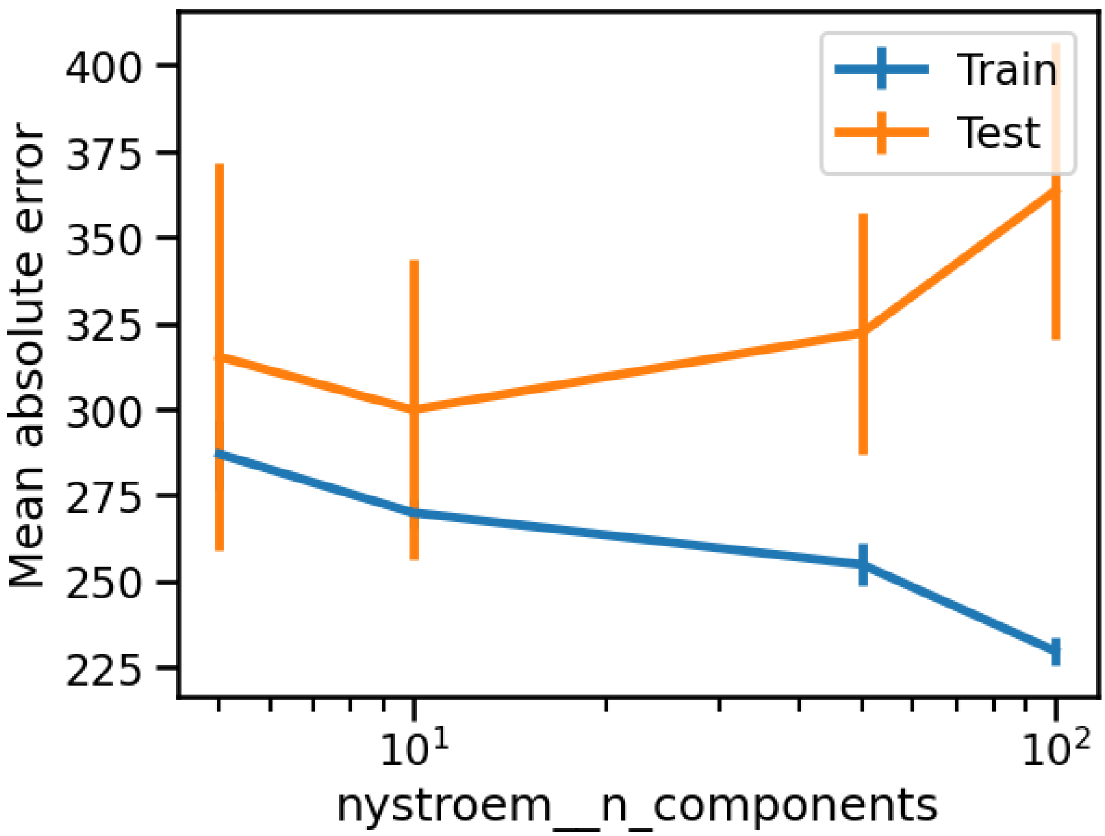
Now let's try to build an alternative pipeline with an adjustable number of intermediate features while keeping a similar predictive power. To do so, try using the `Nystroem` transformer instead of `PolynomialFeatures`. Set the kernel parameter to `"poly"` and `degree` to 2. Adjust the number of components to be as small as possible while keeping a good cross-validation performance.

Hint: Use a `ValidationCurveDisplay` with `param_range = np.array([5, 10, 50, 100])` to find the optimal `n_components`.

[32]:
```
from sklearn.kernel_approximation import Nystroem
from sklearn.model_selection import ValidationCurveDisplay
import numpy as np

ny = Nystroem(kernel="poly", degree=2)
model_with_ny = make_pipeline(ny, linear_regression).
 ↪set_output(transform="pandas")
ValidationCurveDisplay.from_estimator(model_with_ny, data, target,␣
 ↪param_name="nystroem__n_components", cv=10, param_range=np.array([5, 10, 50,␣
 ↪100]), scoring="neg_mean_absolute_error",
    negate_score=True,
    std_display_style = "errorbar",
    n_jobs =2)
```

[32]:
```
<sklearn.model_selection._plot.ValidationCurveDisplay at 0x7f596c1ee910>
```



How do the mean and std of the MAE for the Nystroem pipeline with optimal `n_components` compare to the other previous models?

[34]:
```
model_with_ny.set_params(nystroem__n_components=10)
cv_results = cross_validate(model, data, target, cv=10, scoring =␣
↪"neg_mean_absolute_error", n_jobs=2)

print(f"{-cv_results['test_score'].mean()} +- {cv_results['test_score'].std()}")
```

```
301.78955228431437 +- 44.34000781934426
```

[ ]:
```
```