Linear Regression Recap

Module 1 | Chapter 2 | Notebook 1

In this notebook we will recap the most important things from the previous chapter. This is all material you have already covered. If you discover a lot of new things here, we recommend that you take a look back at Chapter 1. The relevant lessons for each section are clearly marked.

The five steps of sklearn

If you want to organize or predict data using a machine learning model, you go through five steps. See *Simple Linear Regression with sklearn (Module 1 Chapter 1)*.

1. Choose model type
2. Instantiate a model with certain settings known as hyperparameters
3. Organize data into a feature matrix and target vector
4. Model fitting
5. Make predictions with the trained model

So far we have got to know three different machine learning models. All three models are linear regression models, which differ only in whether they optimize the size of the trained slopes in addition to the distance between the measured values and predicted values.

We can imagine the distance between the measured values and predicted values as the fine red lines in the following graphic. The dark blue points are the measurements and the light blue regression line describes the predictions:

`LinearRegression` minimizes this line, which is defined by the intercept and the slope. `Ridge` also minimizes the squared slope values and `Lasso` minimizes the absolute slope values, see *Ridge Regression and Lasso Regression (Module 1 Chapter 1)*.

Ridge regressions are popular when you want to avoid overfitting and when features correlate with each other. Lasso regression is more suitable for removing unimportant features completely.

In [1]:
```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
```

`LinearRegression`, `Ridge` and `Lasso` are what we call *estimators*. They can therefore be fitted to data with their `my_model.fit()` method so they learn optimal parameters to predict

data. All parameters that are not learned by `my_model.fit()` are hyperparameters.

We have already learned about two possible hyperparameters. For linear regressions, you can decide whether the intercept is set at zero (`fit_intercept=False`) or is learned from the data (`fit_intercept=True`). The latter is more usual and therefore the default. For `Ridge` and `Lasso` you can also set the strength of the regularization with the `alpha` parameter. The default here is `1.0` .

In [2]:
```
model_linear = LinearRegression()
model_ridge = Ridge()
model_lasso = Lasso()
```

Once trained, the regression models are then also *predictors*. So you can predict data with the `my_model.predict()` method. We'll come back to this later.

Congratulations: You have repeated the first two of the five steps. Now we can turn to the feature matrix and the target vector.

Feature matrix and target vector

In machine learning models of supervised learning, a distinction is made between feature and target. The target values should be predicted using the values of a single feature or a lot of features. In regression models the target values in the target vector are continuous.

In the data, you will usually find the feature and target vector values in different columns. Let's start by importing the training data that the model can use to learn:

In [3]:
```
import pandas as pd
df_train = pd.read_excel('example_data_training.xlsx')
df_train.head()
```

Out[3]:

	Feature_1	Feature_2	Feature_3	Target
0	1.1	23.0	2.0	9.0
1	2.0	1.1	3.0	3.0
2	3.0	5.0	2.0	5.0
3	4.0	32.0	6.0	7.5
4	5.0	4.0	3.0	8.0

Important: The feature matrix must be a matrix, for example a `DataFrame` . This also applies if it has only one column, i.e. only one feature, see *Simple Linear Regression with sklearn (Module 1 Chapter 1)*.

In [4]:
```
features_train = df_train.loc[:, ['Feature_1',
                                  'Feature_2',
                                  'Feature_3']]
```

```
target_train = df_train.loc[:, 'Target']
```

For `Ridge` and `Lasso` regressions, you need to bring the features onto a uniform scale so that minimizing the slope values does not favor features with higher values. You standardize them with `StandardScaler` to do this.

After using their `my_estimator.fit()` method, some *estimators* are not used to make predictions, but for data transformations. They are then called *transformers*. You can recognize them by the fact that they have a `my_transformer.transform()` method. In the lesson *Regularization (Module 1 Chapter 1)* you got to know a *transformer*: `StandardScaler`.

In [5]:
```
from sklearn.preprocessing import StandardScaler
```

Just like with all other *estimators*, you have to instantiate it first.

In [6]:
```
scaler = StandardScaler()
```

`scaler` can now be used to standardize `DataFrame` columns (features), i.e. to convert them according to the following formula:

Word formula	**math. Formula**
\begin{equation*} \operatorname{feature_{standardized}} = \frac{feature - mean_{feature}}{standard deviation_{feature}} \end{equation*}	\begin{equation*} \operatorname{feature_{standard}} = \frac{x - \bar{x}}{\sigma} \end{equation*}

In [19]:
```
scaler.fit(features_train)  # train scaler
features_train_standardized = scaler.transform(features_train)  # use scaler for stand
features_train_standardized
```

Out[19]:
```
array([[-1.54372366,  0.3582492 , -1.18467348],
       [-1.22867802, -0.76416245, -0.87776326],
       [-0.8786273 , -0.56428092, -1.18467348],
       [-0.52857658,  0.81951425,  0.04296743],
       [-0.17852587, -0.6155326 , -0.87776326],
       [ 0.17152485,  2.51081947,  0.04296743],
       [ 0.52157557, -0.76928761,  0.65678789],
       [ 0.87162629,  0.3582492 ,  0.22711357],
       [ 1.221677  , -0.6155326 ,  1.27060835],
       [ 1.57172772, -0.71803594,  1.88442881]])
```

At the end, the columns have a mean value of 0 and a standard deviation of 1. If all features have the same scale, you can use `Ridge` and `Lasso` regressions without any doubt.

In [8]:
```
pd.set_option('display.float_format', '{:.2f}'.format)  # avoid scientific notation us
pd.DataFrame(features_train_standardized).describe()    # eight value summary
```

Out[8]:

	0	1	2
count	10.00	10.00	10.00
mean	0.00	0.00	-0.00
std	1.05	1.05	1.05
min	-1.54	-0.77	-1.18
25%	-0.79	-0.69	-0.88
50%	-0.00	-0.59	0.04
75%	0.78	0.36	0.55
max	1.57	2.51	1.88

Congratulations: You have repeated the most important things about the feature matrix and the target vector. Now we can turn to the model fitting.

Fitting the model to the data

When a model is fitted to the data, the optimal parameter values are calculated to minimize a particular characteristic. In the case of linear regression, this is the mean squared distance between the regression line and the data points, see the image above (*scatter plot with simple linear regression*) and *Simple Linear Regression with sklearn (Module 1 Chapter 1)*.

In [9]:
```
model_linear.fit(features_train, target_train)
```

Out[9]:
```
LinearRegression()
```

The intercept and slope values of the regression model are changed to achieve this goal. The final values for these parameters can be found in the attributes `my_model.coef_` (gradient values) and `my_model.intercept_` (axis intercept). In general, all parameters that the model learns can be viewed as attributes with an underscore at the end.

In [10]:
```
print(model_linear.coef_)
print(model_linear.intercept_)
```

```
[-0.36607633  0.05338684 -0.28424504]
7.688033277296748
```

As you can see, `my_model.coef_` outputs a slope value for each feature. These values can be directly interpreted. If all feature values are zero, a target value of 7.7 is predicted (intercept). If the first feature value increases by 1, while the other features remain at zero, the target value decreases by 0.37. The same principle can be applied to the other gradient values. There is one slope value per feature.

The multiple linear regression, i.e. linear regression with several features, makes five assumptions (see *Mutiple Linear Regression* Module 1 Chapter 1):

1. The data points are independent from each other

2. There is a linear dependency between feature and target
3. The residuals are normally distributed
4. The residuals have a constant variance
5. The Features are independent from each other

If the features influence each other, it's called collinearity. You can recognize this by the fact that the correlation coefficients between two features take on very large values $p <-0.9$ or $p >0.9$. You can check the correlation coefficient with `my_df.corr()`.

In [20]:
```
import pandas as pd
pd.DataFrame(features_train_standardized,columns=features_train.columns).corr()
```

Out[20]:

	Feature_1	Feature_2	Feature_3
Feature_1	1.00	-0.10	0.91
Feature_2	-0.10	1.00	-0.10
Feature_3	0.91	-0.10	1.00

In this case, Feature_1 and Feature_3 are very strongly correlated and collinearity could become a problem. In these cases you should remove one of the features.

Congratulations: You recapped dividing the training data set into feature matrix and target vector and at the same time you saw the assumptions of the regression model once again. We can look at the predictions.

Making predictions

As we mentioned previously, a trained machine learning model goes from being an *estimator* to being a *predictor*. So you can create the feature data, which you want to get the target value for:

In [12]:
```
features_aim = pd.DataFrame({'Feature_1':[1.1],
                             'Feature_2':[2.2],
                             'Feature_3':[3.3]})
features_aim
```

Out[12]:

	Feature_1	Feature_2	Feature_3
0	1.10	2.20	3.30

The `my_model.predict()` method then carries out the actual prediction. See *Simple Linear Regression with sklearn (Module 1 Chapter 1)*.

In [13]:
```
model_linear.fit(features_train, target_train)
model_linear.predict(features_aim)
```

Out[13]:
```
array([6.46479171])
```

For Ridge and Lasso models, you should also note that you should standardize feature values just like you standardize the feature matrix before it - remember: **Only ever fit to the training set!** See *Ridge Regularization* and *Lasso Regularization (Module 1 Chapter 1)*.

In [14]:
```
#fit scaler on training set
scaler.fit(features_train)

#build ridge_model

#instantiate model
model_ridge = Ridge()
#scale data
features_train_standardised = scaler.transform(features_train)
#fit
model_ridge.fit(features_train_standardised,target_train)

#predict new data

#scale new data
features_aim_standardised = scaler.transform(features_aim) #note that we are reusing t
model_ridge.predict(features_aim_standardised)
```

Out[14]:
```
array([6.3978218])
```

Congratulations: You have recapped how to make predictions with machine learning models. We also need this to be able to predict the model quality, as you will see next.

Model quality metrics

You need model quality metrics so that you know how good the predictions are. We got to know two of them, see *Evaluating Regression Models (Module 1 Chapter 1)*.

In [15]:
```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
```

Both compare predicted values with actual target values. The mean squared error is the mean squared distance between the predicted and actual target value. The coefficient of determination indicates how much dispersion in the data can be explained by the model.

You should **always** calculate the model quality metrics with new independent data. This is data that was not provided to the model for training.

You can think of overfitting as the model learning the target values by heart. It occurs when the model had to learn too many parameters with too little data, see *Too many features: Overfitting (Modul 1 Chapter 1)*. You can spot overfitting when the model quality measures look very good when the training data is used for the evaluation, but they look very bad when evaluated with independent test data.

First we'll import the test data:

In [16]:
```
df_test = pd.read_excel('example_data_test.xlsx')
df_test.head()
```

Out[16]:

	Feature_1	Feature_2	Feature_3	Target
0	2.00	22.00	4.00	9.00
1	3.00	8.00	5.00	7.00
2	5.10	1.00	6.00	8.00
3	4.00	45.70	2.00	5.00
4	6.00	3.00	7.00	6.00

We'll organize it in a feature matrix and target vector

In [17]:
```
features_test = df_test.loc[:, ['Feature_1',
                                'Feature_2',
                                'Feature_3']]

target_test = df_test.loc[:, 'Target']
```

Now we can predict the target values and then compare them with `target_test`:

In [18]:
```
# fit model with non-standardized training data
model_linear.fit(features_train, target_train)

# predict target values with test data
target_test_pred = model_linear.predict(features_test)

# calculate model performance values
MSE = mean_squared_error(target_test, target_test_pred)
R2 = r2_score(target_test, target_test_pred)

# print
print('Linear Regression model performance:')
print('Mean squared error: ', MSE)
print('R squared: ', R2)
```

```
Linear Regression model performance:
Mean squared error:  5.48364944535954
R squared:  0.09702951714015706
```

There's a big difference when we compare them. For example, the coefficient of determination (R^2) indicates that only 10% of the dispersion in the data can be explained by the model.

Congratulations: You have recapped the most important things from Chapter 1. Now it's time to get started with chapter 2.

Remember:

There are five steps to making data driven predictions with `sklearn`:

1. Choose model type
2. Instantiate the model with certain hyperparameters

3. Split data into a feature matrix and target vector
4. Model fitting
5. Make predictions with the trained model

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.
