

Lecture 10: Introduction to scikit-learn

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn

NHH Norwegian School of Economics

March 24, 2026

See GitHub repository for notebooks and data:

<https://github.com/richardfoltyn/FIE463-V26>

Contents

1	Introduction to scikit-learn	1
1.1	Univariate linear regression	2
1.2	Multivariate linear regression	9
1.3	Optimizing hyperparameters with cross-validation	17
1.4	Advanced pipelines and cross-validation: Principal component regression (PCR)	23
1.5	Dealing with categorical data	27
1.6	Optional exercises	29
1.7	Solutions	31

1 Introduction to scikit-learn

Broadly speaking, we can categorize machine learning algorithms into three groups: *supervised learning*, *unsupervised learning*, and *reinforcement learning*. In the remainder of this course, we'll focus on the first category, and we'll study how we can use the supervised learning algorithms implemented in **scikit-learn**, a popular machine learning library for Python.

Supervised learning uses input data (often called independent variables, features, covariates, predictors, or X variables) and the corresponding output data (dependent variable, outcome, target, or y variable) to establish some relationship $y = f(X)$ within a training data set. We can then use this relationship to make predictions about outputs in new data.

Within supervised learning, we distinguish two broad types of approaches:

1. *Regression*, where output data is allowed to take on continuous values; and
2. *Classification*, where output data is restricted to a few values, often called categories or labels.

Unlike in “standard” econometrics, regression and classification usually include a penalty term which helps improve the predictive performance of these models on new data. We will study this setting in the next lecture.

For a more detailed treatment of the topics covered in this and the next lecture, you should check out the book [Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python](#) and the companion GitHub repository.

1.1 Univariate linear regression

Imagine the simplest linear model where the dependent variable y is assumed to be an affine function of the explanatory variable x and an error term ϵ , given by

$$y_i = \alpha + \beta x_i + \epsilon_i$$

for each observation i . In econometrics, the parameters α and β are called the intercept and slope parameters, respectively. In machine learning, the terminology often differs and you might see the same model written as

$$y_i = b + w x_i + \epsilon_i$$

where b is called the *bias* and w is called a *weight*.

Our goal is to estimate the parameters α and β which is most commonly done by ordinary least squares (OLS). OLS is defined as the estimator that finds the estimates $(\hat{\alpha}, \hat{\beta})$ such that the sum of squared errors is minimized. We denote this sum of squared errors as the loss function L ,

$$L(\alpha, \beta) = \sum_i^N (y_i - \alpha - \beta x_i)^2$$

which depends on the choice of parameters (α, β) . Note that we use the “hat” notation $\hat{\alpha}$ to distinguish the OLS estimate from the (usually unknown) true parameter α . The exact values of $\hat{\alpha}$ and $\hat{\beta}$ will vary depending on the sample size and estimator used as we will see later in this unit.

For this simple model, the estimates are given by the expressions

$$\begin{aligned}\hat{\beta} &= \frac{\widehat{\text{Cov}}(y, x)}{\widehat{\text{Var}}(x)} \\ \hat{\alpha} &= \bar{y} - \hat{\beta} \bar{x}\end{aligned}$$

where $\widehat{\text{Cov}}(\bullet, \bullet)$ and $\widehat{\text{Var}}(\bullet)$ are the *sample* covariance and variance, respectively, and \bar{y} and \bar{x} are the sample means of y and x .

There is a straightforward generalization to the multivariate setting where we have a vector \mathbf{x}_i of explanatory variables (which usually includes the intercept) and a parameter vector β so that the model is given by

$$y_i = \mathbf{x}_i' \beta + \epsilon_i$$

If we stack all \mathbf{x}_i in the matrix \mathbf{X} and all y_i in the vector \mathbf{y} , the OLS estimate of β is given by the well-known formula

$$\hat{\beta} = (\mathbf{X}' \mathbf{X})^{-1} \mathbf{X}' \mathbf{y}$$

We won’t be estimating linear regressions based on this formula and you should never attempt this, as naively implementing such matrix operations can lead to numerical problems. Instead, always use pre-packaged least-squares solvers such as those implemented in NumPy’s `lstsq()` or SciPy’s `lstsq()` functions. For econometrics and machine learning, it usually makes sense to use linear regression models such as those implemented in `statsmodels` or `scikit-learn`, which is what we turn to next.

1.1.1 Univariate linear regressions with scikit-learn

We start by fitting the simple linear model

$$y_i = \alpha + \beta x_i + \epsilon_i$$

using `scikit-learn`. For now, we proceed using synthetically generated data where we know the true relationship, assuming that

$$\begin{aligned}y_i &= 1 + \frac{1}{2} x_i + \epsilon_i \\ \epsilon_i &\stackrel{\text{iid}}{\sim} N(0, \sigma^2)\end{aligned}$$

so that the true parameters are $\alpha = 1$ and $\beta = \frac{1}{2}$. The error term ϵ_i is assumed to be normally distributed with mean 0 and variance σ^2 which we set to $\sigma^2 = 0.7^2$ for this example.

We generate a sample of $N = 50$ observations as follows:

```
[1]: import numpy as np
from numpy.random import default_rng

def create_sample(N, alpha, beta, sigma, rng=None):
    """
    Create sample for univariate linear regression
    """

    # Generate RNG instance with seed 1234
    rng = default_rng(seed=1234)

    # Use x that are uniformly distributed on [0, 10]
    x = rng.uniform(0, 10, size=N)

    # Normally distributed errors
    epsilon = rng.normal(scale=sigma, size=N)

    # Create outcome variable
    y = alpha + beta * x + epsilon

    return x, y
```

```
[2]: # Sample size
N = 50

# True model parameters
alpha = 1.0
beta = 0.5
sigma = 0.7

# Create sample
x, y = create_sample(N=50, alpha=1.0, beta=0.5, sigma=0.7)
```

To fit a linear model, we use the `LinearRegression` class provided by scikit-learn. Before doing so, we need to import it from `sklearn.linear_model`. Note that most fitting routines in scikit-learn expect a *matrix* as opposed to a vector even if the model has only one explanatory variable, so we need to insert an artificial axis to create the matrix $X = x[:, \text{None}]$.

```
[3]: from sklearn.linear_model import LinearRegression

# Create LinearRegression object
lr = LinearRegression(fit_intercept=True)

# fit() expects a two-dimensional object, convert x to N-by-1 matrix
X = x[:, None]

# Fit model to data
lr.fit(X, y)

# Predict fitted values
yhat = lr.predict(X)
```

Note that scikit-learn models usually store the fitted model parameters in the `coef_` attribute (which is an array even if there is only a single explanatory variable). If the model includes an intercept, its fitted value is stored in the attribute `intercept_`:

```
[4]: # Extract parameter estimates from LinearRegression object
alpha_hat = lr.intercept_
beta_hat = lr.coef_[0]

print(f'Estimated alpha: {alpha_hat:.3f}')
print(f'Estimated beta: {beta_hat:.3f}')
```

Estimated alpha: 1.376
 Estimated beta: 0.482

The following code visualizes the sample as a scatter plot of (x_i, y_i) and adds the fitted line (in solid orange). Intuitively, for $x = 0$ the fitted line has the value $\hat{\alpha}$ (the intercept) and its slope is equal to $\hat{\beta}$.

```
[5]: import matplotlib.pyplot as plt

plt.figure(figsize=(5, 3.5))

# Plot true relationship
plt.axline(
    (0.0, alpha), slope=beta, lw=1.0, ls='--', c='black',
    label=r'$y = \alpha + \beta x$'
)

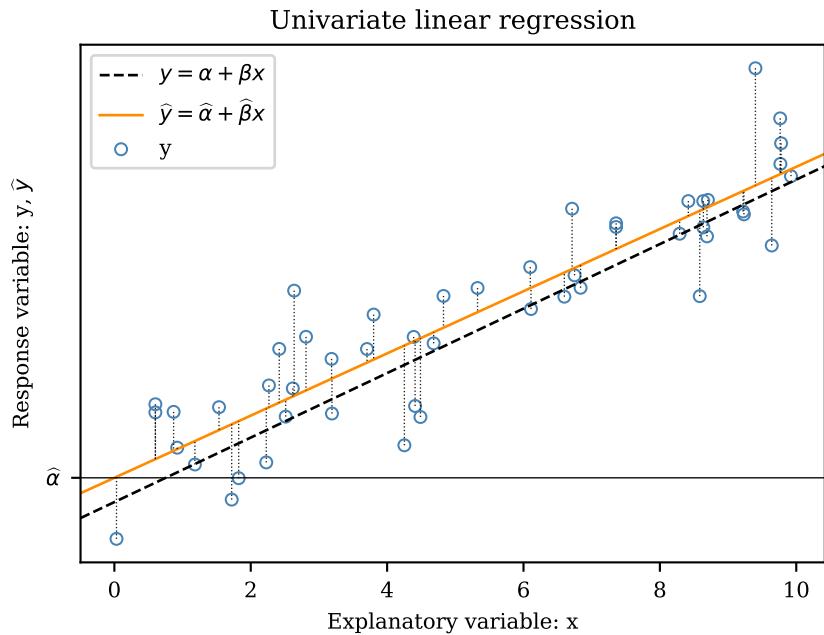
# Plot regression line
plt.axline(
    (0.0, alpha_hat), slope=beta_hat, lw=1.0, c='darkorange',
    label=r'$\widehat{y} = \widehat{\alpha} + \widehat{\beta} x$'
)

# Plot raw data
plt.scatter(
    x, y, s=20, color='none', edgecolor='steelblue', alpha=1.0, lw=0.75, label='y'
)

# Plot lines connecting true and predicted values for
# each observation
for i in range(len(x)):
    plt.plot([x[i], x[i]], [y[i], yhat[i]], lw=0.5, ls=':', c='black', alpha=0.9)

# Add annotations
plt.xlabel('Explanatory variable: x')
plt.ylabel(r'Response variable: y, $\widehat{y}$')
plt.axhline(alpha_hat, lw=0.5, c='black')
plt.legend(loc='best')
plt.yticks((alpha_hat,), (r'$\widehat{\alpha}$',))

_ = plt.title('Univariate linear regression')
```



The dashed black line shows the true relationship which is known in this synthetic sample, but in general is unknown for real data. For small sample sizes, the true and estimated parameters need not be close as the graph illustrates. The graph also shows the prediction errors as the dotted lines between the sample points y_i and the fitted model. OLS minimizes the sum of the squares of these distances.

Your turn.

1. Recreate and re-estimate the model for samples of size 10, 50, 100, 1000, and 10000.
2. Print the estimated values of α and β . Can you say something about the deviation from the true values as the sample size increases?

1.1.2 Training and test samples

In econometrics, we usually emphasize *inference*, i.e., we are interested in testing a hypothesis about a model parameter, for example whether it is statistically different from zero. Conversely, in machine learning the emphasis is often on prediction, i.e., our goal is to estimate a relationship from a *training* sample and make predictions for new data. Usually, we use a *test* sample that is different from the training data to assess how well our model is able to predict outcomes for new data which has not been used for estimation.

To demonstrate the use of training and test data sets, we use a simplified variant of the Ames house price data set. The original data set has 2,930 observations on around 80 features (explanatory variables) which are characteristics of houses sold between 2006 and 2010 in Ames, a city of about 60 thousand inhabitants in the middle of Iowa, USA. This is one of the most common data sets used in introductory machine learning classes. See [here](#) for an overview of the data, [here](#) for a detailed documentation of the original variables, and [here](#) for the original data in tab-separated format.

The goal is to use these features to predict the house price (or “target” in ML terminology). The above websites provide details on all 80 features.

We will use a local copy of the simplified data set which contains only a subset of 15 features that are slightly adapted for our purposes.

```
[6]: # Use the local data/ directory
DATA_PATH = '../.../data'
```

```
# Load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/FIE463-V26/main/data'
```

[7]: import pandas as pd

```
file = f'{DATA_PATH}/ames_houses.csv'
df = pd.read_csv(file)
```

```
# List columns present in DataFrame
df.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2930 entries, 0 to 2929
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype  
---  --  
 0   LotArea          2930 non-null   float64 
 1   Neighborhood     2930 non-null   object  
 2   OverallQuality   2930 non-null   int64  
 3   OverallCondition 2930 non-null   int64  
 4   YearBuilt        2930 non-null   int64  
 5   YearRemodeled   2930 non-null   int64  
 6   BuildingType     2767 non-null   object  
 7   CentralAir       2930 non-null   object  
 8   LivingArea       2930 non-null   float64 
 9   Bathrooms        2930 non-null   int64  
 10  Bedrooms         2930 non-null   int64  
 11  Fireplaces       2930 non-null   int64  
 12  SalePrice        2930 non-null   float64 
 13  YearSold         2930 non-null   int64  
 14  MonthSold        2930 non-null   int64  
 15  HasGarage        2930 non-null   int64  
dtypes: float64(3), int64(10), object(3)
memory usage: 366.4+ KB
```

In this section, we focus on the column SalePrice which contains the house price in thousands of US dollars. Our goal is to predict this sale price using some other variables from the data. We can use corr() to compute the pairwise correlations of the sale price and all other variables in this data set:

[8]: df.corr(numeric_only=True).loc["SalePrice"]

```
[8]: LotArea          0.266550
OverallQuality    0.799261
OverallCondition   -0.101694
YearBuilt         0.558425
YearRemodeled    0.532974
LivingArea        0.706780
Bathrooms         0.545605
Bedrooms          0.143914
Fireplaces        0.474555
SalePrice          1.000000
YearSold          -0.030568
MonthSold         0.035261
HasGarage         0.225950
Name: SalePrice, dtype: float64
```

As you can see, the correlation is highest with the living area (LivingArea) in m^2 (for this exercise, we ignore OverallQuality which is a categorical variable).

To get some intuition for the data, we look at some descriptive statistics:

[9]: df[['SalePrice', 'LivingArea']].describe()

```
[9]:      SalePrice  LivingArea
count    2930.000000  2930.000000
mean     180.796382  139.311980
std      79.887027  46.958135
min      12.800000  31.000000
25%     129.500000  104.600000
50%     160.000000  134.000000
75%     213.500000  161.875000
max     755.000000  524.100000
```

For our analysis, we first remove a handful of outliers and then extract the target (`SalePrice`) and the features (`LivingArea`) from the data set.

```
[10]: # Remove outliers in terms of living area
df = df.query('LivingArea <= 350')

features = ['LivingArea']
target = 'SalePrice'

# Create separate Series / DataFrames for target and features
y = df[target]
X = df[features]
```

Automatically creating training and test samples

Instead of splitting the sample manually into training and test sub-samples, we use scikit-learn's `train_test_split()` to automate this task. We need to either specify the `test_size` or `train_size` arguments to determine how the sample should be (randomly) split. The `random_state` argument is used to seed the RNG and get reproducible results:

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

# fraction of data used for test sample
test_size = 0.3

# Split into training / test samples
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=test_size,           # Fraction used for test sample
    random_state=1234,            # Seed for random number generator
)
```

Once we have split the sample, we estimate the model on the training sample.

```
[12]: from sklearn.linear_model import LinearRegression

# Fit model on training sample
lr = LinearRegression()
lr.fit(X_train, y_train)

# Extract estimated parameters
intercept = lr.intercept_
slope = lr.coef_[0]

print('Regression coefficients')
print(f'  Intercept: {intercept:.2f}')
print(f'  Slope: {slope:.2f}')
```

Regression coefficients

Intercept: 8.59

Slope: 1.23

The fitted coefficients show that for each additional square meter of living area, the sale price on average increases by approximately \$1,200.

The following code creates a scatter plot showing the training and test samples and adds the fitted line.

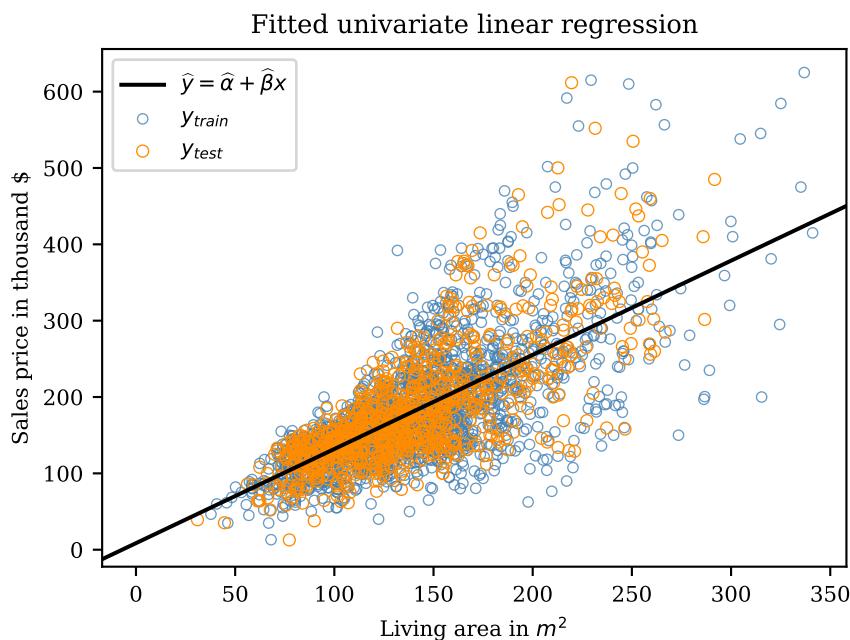
```
[13]: plt.subplots(figsize=(5, 3.5))

# Plot regression line
plt.axline((0.0, intercept), slope=slope, lw=1.5, c='black',
           label=r'$\widehat{y} = \widehat{\alpha} + \widehat{\beta} x$')

# Plot training data
plt.scatter(X_train, y_train, s=15, color='none', edgecolor='steelblue',
            alpha=0.8, lw=0.5, label=r'$y_{train}$')

# Plot test data
plt.scatter(X_test, y_test, s=20, color='none', edgecolor='darkorange',
            alpha=1.0, lw=0.5, label=r'$y_{test}$')

# Add annotations
plt.xlabel('Living area in $m^2$')
plt.ylabel(r'Sales price in thousand $')
plt.title('Fitted univariate linear regression')
_ = plt.legend(loc='best')
```



Since this is a univariate model, we can also plot the prediction error against the explanatory variable x . For this, we first need to compute the predicted values in the test sample and then the prediction error for each observations,

$$\epsilon_i = y_i - \hat{y}_i = y_i - \hat{\alpha} - \hat{\beta}x_i$$

for all i that are part of the test sample.

```
[14]: # Predict model for test sample
y_test_hat = lr.predict(X_test)

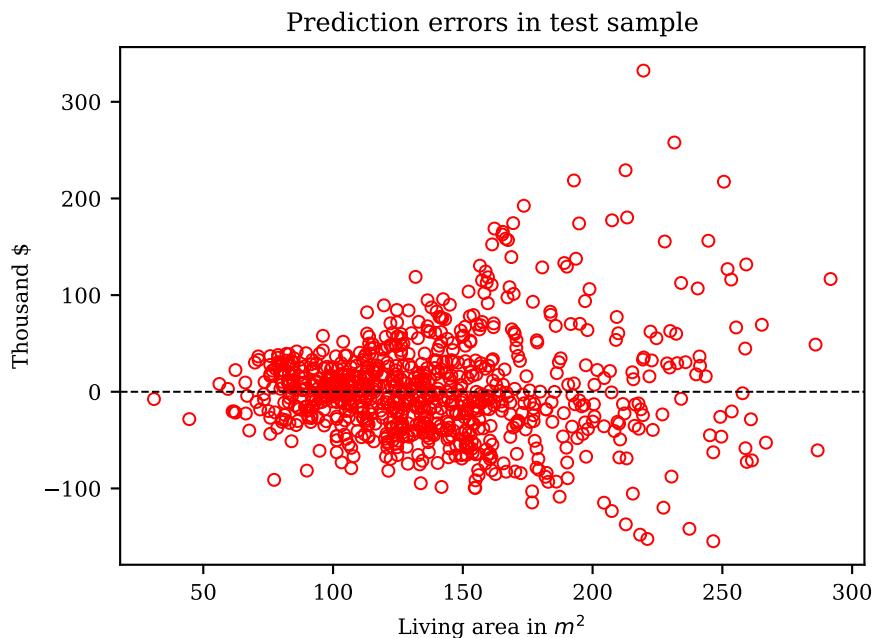
# Prediction error for test sample
error = y_test - y_test_hat
```

We can now plot the predition errors against the living area:

```
[15]: plt.figure(figsize=(5, 3.5))

# Scatter plot of prediction errors
plt.scatter(X_test, error, s=20, color='none', edgecolor='red', alpha=1.0, lw=0.75)

# Add annotations
plt.xlabel('Living area in $m^2$')
plt.ylabel(r'Thousand $')
plt.title('Prediction errors in test sample')
_ = plt.axhline(0.0, lw=0.75, ls='--', c='black')
```



As the graph shows, the errors are reasonably centered around 0 as we would expect from a model that contains an intercept. However, the error variance seems to be increasing in x which indicates that our model might be missing some explanatory variables.

1.2 Multivariate linear regression

1.2.1 Data with several different explanatory variables

Multivariate (or multiple) linear regression extends the simple model to multiple explanatory variables or regressors. To illustrate, we'll load the Ames housing data again but use several (continuous) explanatory variables.

```
[16]: # Use local data/ directory
DATA_PATH = '../data'

# Load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/FIE463-V26/main/data'
```

```
[17]: import pandas as pd

# Load data from CSV file
file = f'{DATA_PATH}/ames_houses.csv'
df = pd.read_csv(file)

# Print info about columns contained in DataFrame
df.info(show_counts=True)
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2930 entries, 0 to 2929
Data columns (total 16 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   LotArea            2930 non-null    float64
 1   Neighborhood       2930 non-null    object  
 2   OverallQuality     2930 non-null    int64  
 3   OverallCondition   2930 non-null    int64  
 4   YearBuilt          2930 non-null    int64  
 5   YearRemodeled     2930 non-null    int64  
 6   BuildingType       2767 non-null    object  
 7   CentralAir         2930 non-null    object  
 8   LivingArea         2930 non-null    float64
 9   Bathrooms          2930 non-null    int64  
 10  Bedrooms           2930 non-null    int64  
 11  Fireplaces         2930 non-null    int64  
 12  SalePrice          2930 non-null    float64
 13  YearSold          2930 non-null    int64  
 14  MonthSold         2930 non-null    int64  
 15  HasGarage          2930 non-null    int64  
dtypes: float64(3), int64(10), object(3)
memory usage: 366.4+ KB

```

Compared to our earlier analysis, we'll now add the lot area (in m^2) as a feature to the model, which is thus given by

$$SalePrice_i = \alpha + \beta_0 LivingArea_i + \beta_1 LotArea_i + \epsilon_i$$

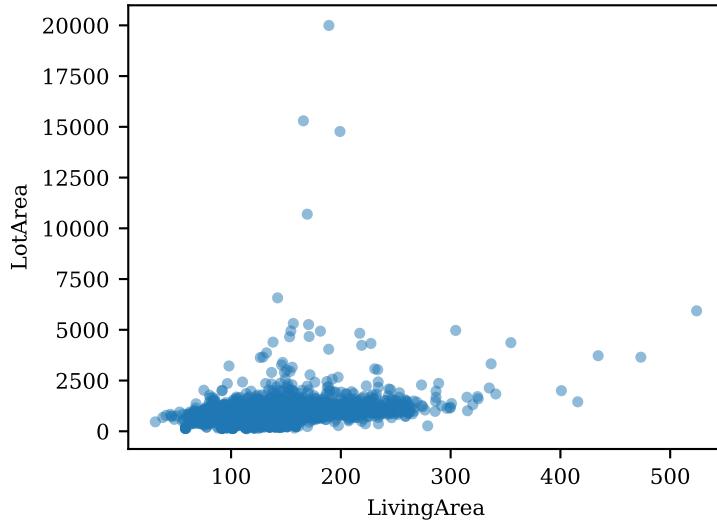
We first inspect the descriptive statistics of the newly added explanatory variable to get some idea about its distribution.

```
[18]: df[['SalePrice', 'LivingArea', 'LotArea']].describe()
```

```
[18]:      SalePrice  LivingArea  LotArea
count  2930.000000  2930.000000  2930.000000
mean   180.796382  139.311980  942.681877
std    79.887027  46.958135  732.005230
min    12.800000  31.000000  120.800000
25%   129.500000  104.600000  691.125000
50%   160.000000  134.000000  876.600000
75%   213.500000  161.875000  1073.425000
max    755.000000  524.100000  19995.000000
```

We see that as with the living area, there are some quite large values for the lot area. We can visualize these in the scatter plot:

```
[19]: # Scatter plot of living area and lot area
_ = df.plot.scatter(x='LivingArea', y='LotArea', s=10, alpha=0.5)
```



For our analysis, we drop observations with a lot area above 5,000m² as well as properties with a living area of more than 350m², as we did earlier.

We define the function below to load a fresh copy of the Ames data, apply the sample restrictions and return the feature matrix and target variables. We'll use this function throughout the remained of the section.

```
[20]: import pandas as pd

def load_ames_data(features, target):
    """
    Load Ames housing data from CSV file and select features and target variable
    """

    # Load data from CSV file
    file = f'{DATA_PATH}/ames_houses.csv'
    df = pd.read_csv(file)

    # Remove extreme outliers in terms of living area or lot area
    df = df.query('LivingArea <= 350 & LotArea <= 5000')

    # Drop any missing obs in selected columns
    columns = features + [target]
    df = df.dropna(subset=columns)

    # Select features and target variable
    X = df[features]
    y = df[target]

    return X, y
```

```
[21]: features = ['LivingArea', 'LotArea']
target = 'SalePrice'

X, y = load_ames_data(features, target)
```

We can now split the data into training and test samples:

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

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1234
)
```

Fitting the model on the training data is the same as in the univariate case:

```
[23]: from sklearn.linear_model import LinearRegression  
  
lr = LinearRegression()  
lr.fit(X_train, y_train)  
  
print(f'Intercept: {lr.intercept_}')  
print(f'Coefficients: {lr.coef_}')
```

```
Intercept: -4.88926070624683  
Coefficients: [1.15441095 0.0275464 ]
```

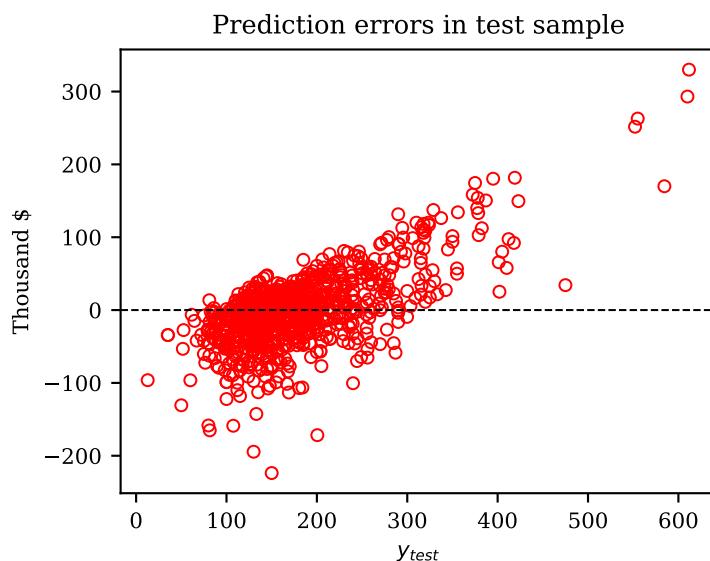
The coefficient array `coef_` now stores two values, the first one for `LivingArea` and the second one for `LotArea`. These coefficients are in the same order as the features in the feature matrix `X` passed when fitting the model.

Since we now have multiple explanatory variables, we can no longer easily plot prediction errors against one feature unless we fix the remaining features at some value or use 3D scatter plots. The latter of course does not help if we have more than two explanatory variables. Instead, we can plot the prediction errors against `y` which we do below.

```
[24]: # Predict values in test sample  
y_test_hat = lr.predict(X_test)  
  
# Compute prediction errors  
errors = y_test - y_test_hat
```

We can now plot these prediction errors against the target `y`:

```
[25]: import matplotlib.pyplot as plt  
  
plt.scatter(y_test, errors, s=20, lw=0.75, color='none', edgecolor='red')  
plt.axhline(0.0, lw=0.75, ls='--', c='black')  
plt.title('Prediction errors in test sample')  
plt.xlabel(r'$y_{test}$')  
plt.ylabel('Thousands $')
```



The scatter plot indicates that the prediction errors differ systematically for different levels of y . The model on average overpredicts the sale price for low y (hence the error is negative) and underpredicts the price for large y (hence the error is positive).

1.2.2 Creating polynomial features

Instead of including additional explanatory variables, we can also create additional terms that are functions of variables. The most common way to do this is to include a polynomial in x (or polynomials in multiple explanatory variables). Thus a simple model could be turned into a model with several terms such as

$$y_i = \alpha + \beta_0 x_i + \beta_1 x_i^2 + \beta_2 x_i^3 + \epsilon_i$$

where y is modelled as a cubic polynomial in x . Note that this model is still called *linear* despite the fact that the mapping between x and y is obviously *non-linear*. However, what matters for estimation is that the model is linear in the model parameters ($\alpha, \beta_0, \beta_1, \beta_2$). Linear models are thus quite flexible since they can include almost arbitrary non-linear transformations of explanatory and response variables.

To illustrate, we extend the previous model which included `LivingArea` and `LotArea` to now include a polynomial of degree 2 in both variables (often the terms “degree” and “order” are used interchangeably, so this might be called a 2nd-order polynomial). Specifically, if we have two variables x and z , such a polynomial would include all terms with exponents summing up to 2 or less:

$$p(x, z) = \beta_0 + \beta_1 x + \beta_2 z + \beta_3 x^2 + \beta_4 x \cdot z + \beta_5 z^2$$

We can use these six terms as explanatory variables in our linear models and estimate the parameters β_0, \dots, β_5 .

It would be quite error-prone to create such polynomials ourselves, so we are going to use scikit-learn’s `PolynomialFeatures` class to accomplish this tasks. As the name implies, this transformation creates new features that are polynomials of a given input matrix. We can request that the resulting data should include an intercept (“bias”) by specifying `include_bias=True`. Note that if an intercept is included in the feature matrix, we should fit the linear model *without* an intercept (by specifying `fit_intercept=False`) as otherwise the model would contain two constant terms.

```
[26]: from sklearn.preprocessing import PolynomialFeatures

# Create polynomials of degree 2 or less, including an intercept (bias)
poly = PolynomialFeatures(degree=2, include_bias=True)
Xpoly_train = poly.fit_transform(X_train)
```

The `powers_` attribute stores the list of exponents for each generated feature:

```
[27]: # print polynomial exponents
poly.powers_
```

```
[27]: array([[0, 0],
       [1, 0],
       [0, 1],
       [2, 0],
       [1, 1],
       [0, 2]])
```

The above output tells us that the first feature was constructed as $X_{(1)}^0 + X_{(2)}^0$ since the first row of exponents is $[0, 0]$ and is thus the intercept, while the second feature is given by $X_{(1)}^1 + X_{(2)}^0 = X_{(1)}$, where the notation $X_{(i)}$ refers to the i -th column of the input matrix X .

Now that we have transformed the input matrix X , we can estimate the linear model on the expanded feature matrix as before (adding `fit_intercept=False`):

```
[28]: # Fit linear regression to polynomial features
lr = LinearRegression(fit_intercept=False)
lr.fit(Xpoly_train, y_train)

print(f'Intercept: {lr.intercept_}')
print(f'Coefficients: {lr.coef_}')
```

```

Intercept: 0.0
Coefficients: [ 1.58390691e+01  1.08210136e+00 -5.69279846e-03 -9.87407071e-04
 3.65461705e-04 -8.12934055e-06]

```

The fitted model has 6 coefficients and the intercept is 0 as the `LinearRegression` model did not explicitly include one.

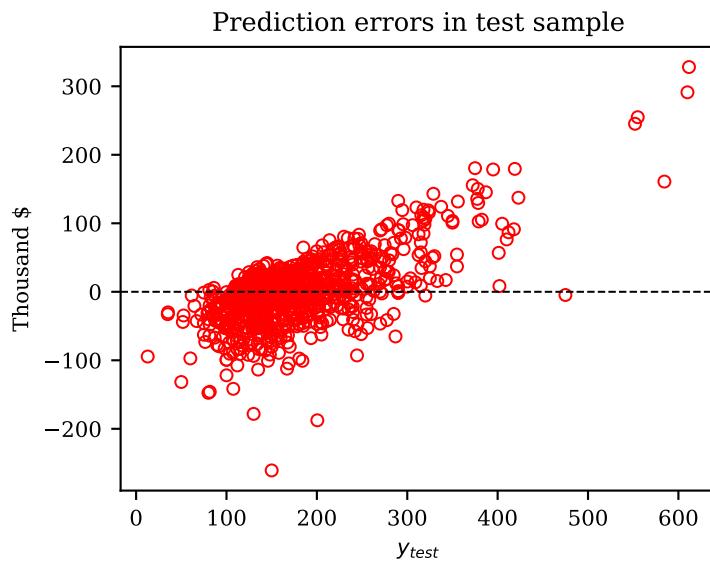
As before, we plot the prediction errors as a function of the response variable. Before doing this, it is crucial to also transform the original explanatory variables in the test data set using the same polynomial transformation. We can achieve this by using the `transform()` method of the `poly` object we stored from earlier:

```
[29]: # Predict values in test sample, applying the same polynomial to
# explanatory variables in test sample.
Xpoly_test = poly.transform(X_test)
y_test_hat = lr.predict(Xpoly_test)

# Compute prediction errors
errors = y_test - y_test_hat
```

As before, we plot the prediction errors against the target y in the test sample:

```
[30]: plt.scatter(y_test, errors, s=20, lw=0.75, color='none', edgecolor='red')
plt.axhline(0.0, lw=0.75, ls='--', c='black')
plt.xlabel(r'$y_{test}$')
plt.ylabel(r'Thousand $')
_ = plt.title('Prediction errors in test sample')
```



1.2.3 Using scikit-learn pipelines

As you just saw, additional transformation steps before fitting the model can be tedious and error-prone (we might, for example, forget to transform the test data before computing predictions for the test sample). For this reason, scikit-learn implements a feature called “pipelines” which allows us to combine multiple transformations and a final estimation step. For this to work, all steps in the pipeline except for the last must support `fit()` and `transform()` methods, and the final step in the pipeline should be an estimator such as `LinearRegression` (for details, see the [section on pipelines](#) in the scikit-learn user guide).

There are two ways to construct a pipeline:

1. Create an instance of the `Pipeline` class and specify the steps as name-value pairs.

2. Use the `make_pipeline()` convenience function, which sets a default name for each step.

The first approach requires a list of tuples, where each tuple contains an (arbitrary) name and an object that implements the actions taken at this step. To compose a pipeline that creates polynomial features and fits a linear model to them, we therefore proceed as follows:

```
[31]: from sklearn.pipeline import Pipeline

# Create pipeline with two steps
pipe = Pipeline(steps=[
    ('poly', PolynomialFeatures(degree=2)),
    ('lr', LinearRegression(fit_intercept=False))
])

# visualize pipeline (interactive notebook only)
pipe
```

```
[31]: Pipeline(steps=[('poly', PolynomialFeatures()),
                      ('lr', LinearRegression(fit_intercept=False))])
```

In an interactive notebook, printing the `pipe` object will generate a visualization which contains details for each step (some editors such as Visual Studio Code even make this visualization interactive). To transform and fit the model in a single call, we simply need to invoke the `fit()` method. For example, using the training data from above, we run

```
[32]: # transform and fit in a single step
pipe.fit(X_train, y_train)

print(f'Coefficients: {pipe.named_steps.lr.coef_}')
```

```
Coefficients: [ 1.58390691e+01  1.08210136e+00 -5.69279846e-03 -9.87407071e-04
 3.65461705e-04 -8.12934055e-06]
```

The alternative approach is to construct the pipeline using `make_pipeline()`. For this to work, we only need to pass the objects that constitute the individual steps of a pipeline as follows:

```
[33]: from sklearn.pipeline import make_pipeline

# Simplified pipeline creation
pipe = make_pipeline(
    PolynomialFeatures(degree=2),
    LinearRegression(fit_intercept=False)
)

# transform and fit model in single step
pipe.fit(X_train, y_train)

# print default names assigned to each step
pipe.named_steps
```

```
[33]: {'polynomialfeatures': PolynomialFeatures(),
        'linearregression': LinearRegression(fit_intercept=False)}
```

The function assigns default names to each step which are printed above (basically, these are just lowercase versions of the class defining a step). These names are occasionally required to retrieve information for a specific step. To demonstrate that the pipeline generates results that are equivalent to our manual implementation, we conclude this section by recreating the error plot from above.

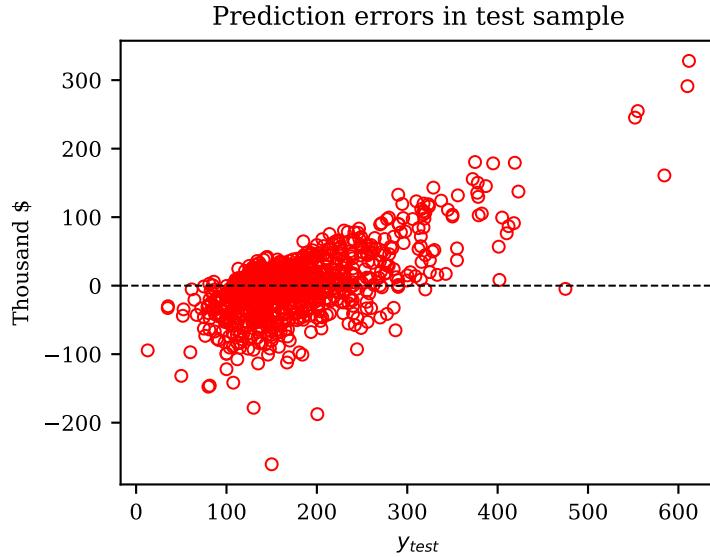
```
[34]: # Predict test responses in a single step. No manual transformation required!
y_test_hat = pipe.predict(X_test)

# Compute prediction errors
errors = y_test - y_test_hat
```

```

plt.scatter(y_test, errors, s=20, lw=0.75, color='none', edgecolor='red')
plt.axhline(0.0, lw=0.75, ls='--', c='black')
plt.xlabel(r'$y_{\text{test}}$')
plt.ylabel(r'Thousand $')
_ = plt.title('Prediction errors in test sample')

```



1.2.4 Evaluating the model fit

So far, we evaluated models only based on the visual inspection of the prediction errors. A more systematic approach uses one of several common metrics listed below. These become particularly relevant for models with hyperparameters, i.e., additional parameters that are particular to the estimation technique but are not included in the model as such. We can use these metrics to select optimal hyperparameters, as we demonstrate later in the lecture.

One of the commonly used metrics to evaluate models is the *mean squared error (MSE)*, defined as

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

which computes the average squared prediction error $y - \hat{y}$. The magnitude of the MSE is usually hard to interpret, so we often compute the *root mean squared error (RMSE)*,

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2}$$

which can be interpreted in units of the response variable y . Lastly, another measure of a model's fit is the *coefficients of determination* or R^2 , which is a normalized version of the MSE and usually takes on values between $[0, 1]$. The R^2 is defined as

$$R^2 = 1 - \frac{MSE}{\widehat{Var}(y)}$$

where $\widehat{Var}(y)$ is the sample variance of the response y . Intuitively, an R^2 of 1 means that the model predicts the response for each observation perfectly (which is unlikely), whereas an R^2 of 0 implies that the model possesses no explanatory power relative to a model that includes only the sample mean. Note that in a test sample, the R^2 could even be negative.

While these measures are easy to implement ourselves, we can just as well use the functions provided in `scikit-learn.metrics` to do the work for us:

- `mean_squared_error()` for the MSE;
- `root_mean_squared_error()` for the RMSE; and
- `r2_score()` for the R^2 .

To illustrate, we continue with the example from above and compute these statistics using the original and predicted target values on the test sample:

```
[35]: from sklearn.metrics import mean_squared_error, root_mean_squared_error, r2_score

# Mean squared error (MSE)
mse = mean_squared_error(y_test, y_test_hat)

# Mean squared error (MSE)
rmse = root_mean_squared_error(y_test, y_test_hat)

# Coefficient of determination (R²)
r2 = r2_score(y_test, y_test_hat)

print(f'Mean squared error: {mse:.1f}')
print(f'Root mean squared error {rmse:.1f}')
print(f'Coefficient of determination (R^2): {r2:.2f}')
```

```
Mean squared error: 2582.8
Root mean squared error 50.8
Coefficient of determination (R^2): 0.53
```

Since we estimated a model with intercept, the $R^2 = 0.55$ implies that the model explains 55% of the variance in the test sample.

1.3 Optimizing hyperparameters with cross-validation

1.3.1 Outline of hyperparameter tuning

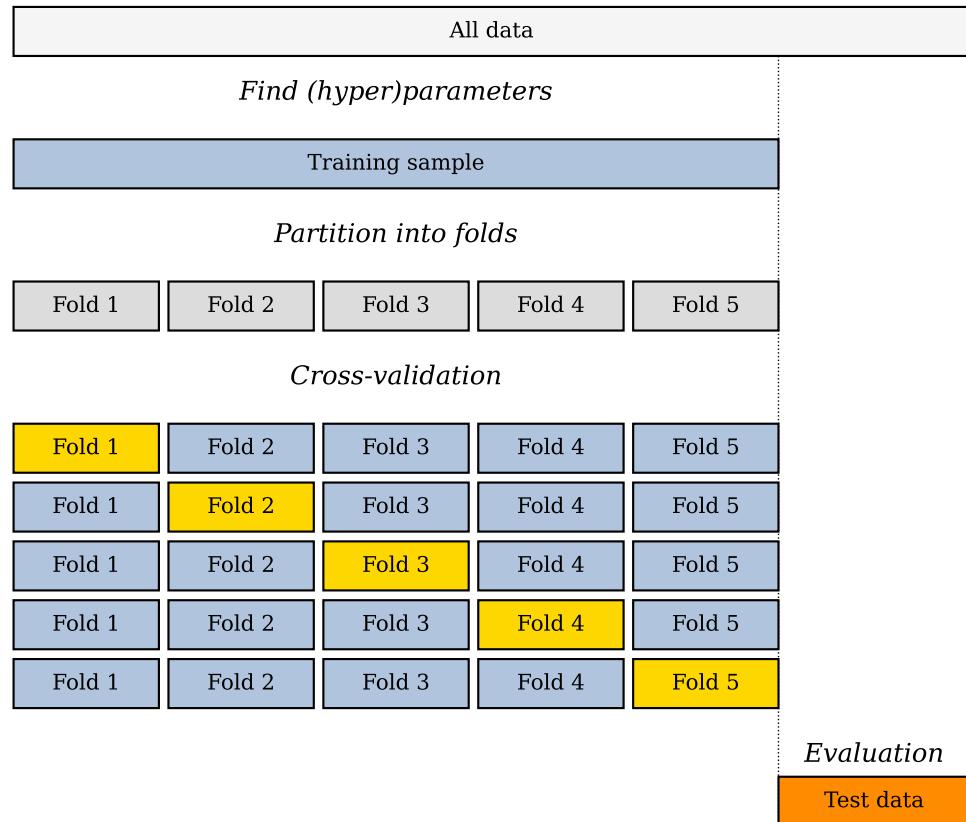
Previously, we discussed some ways to evaluate the model fit (MSE and R^2) but did not specify what to do with this information. In this section, we demonstrate how we can use these measures to tune parameters which govern the estimation process, such as the polynomial degree from the previous section. Such parameters are usually called *hyperparameters* to clearly distinguish them from the parameters that are estimated by the model (e.g., the coefficients of a linear model).

Tuning hyperparameters, estimating the model and evaluating its generalizability cannot be done based on the same data set, as these steps then become interdependent (for the same reason we don't want to evaluate the model fit on the training sample as this would lead to overfitting). To this end, in machine learning we often split the data into *three* parts: the training, validation and test data sets. Models are estimated on the training sample, the choice of hyperparameters is determined based on the validation sample and model generalizability is determined based on the test sample which was not used in estimation or tuning at all.

However, because we might not have enough data to split the sample this way, we usually perform so-called *cross-validation* which is illustrated in the figure below:

1. We split the overall data set into training and test sub-samples.
2. The training data set is further split into K so-called *folds*.
 1. For each $k = 1, \dots, K$, a smaller training sample is formed by excluding the k -th fold and estimating the model on the remaining $K - 1$ folds. We then compute the chosen metric of model fit on the k -th fold and store the result.
 2. After cycling through all K folds, we have K values of our desired metric, which we then average to get our final measure. Hyperparameters are tuned by minimizing (or maximizing) this averaged metric.

- Once hyperparameter tuning is complete, we evaluate the model on the test data set.



In this unit we will skip the final step of assessing generalizability on the test data set. Consequently, when running cross-validation, we will only use training/validation data sets and we will use the terms “validation” and “test” interchangeably.

The [scikit-learn documentation](#) contains a wealth of additional information on cross-validation. Another method to perform hyperparameter tuning is [grid search](#).

1.3.2 Example: Tuning of the polynomial degree

To illustrate the concept, we demonstrate the procedure outlined above by tuning the polynomial degree for the example covered in the previous section by minimizing the root mean squared error (RMSE).

We recreate the data set in the same way as before, using `LivingArea` and `LotArea` as explanatory variables.

```
[36]: # Load Ames housing data
features = ['LivingArea', 'LotArea']
target = 'SalePrice'

X, y = load_ames_data(features, target)
```

We now iterate over the candidate polynomial degrees $d = 1, \dots, 5$ and apply k -fold cross-validation with 10 folds. There is no need to manually split the sample into training and validation/test sub-sets. Instead, we use the [KFold](#) class to automatically create the splits for us. Once we have created an instance with the desired number of folds, e.g., `KFold(n_splits=10)`, we can call the `split()` method which iterates through all possible combinations of training and test data sets and returns the array indices for each.

Note that in the code below, we have to be a little careful on how we create the pipeline for each polynomial degree:

- For $d = 0$, the model only has an intercept. We need to call `PolynomialFeatures` with `include_bias=True`, otherwise there are no explanatory variables and the code fails.
- On the other hand, for $d > 0$, and in particular for large d , the polynomials are so large in magnitude that this can lead to numerical instability. It is therefore good practice to rescale features using the `StandardScaler`. However, the intercept (bias) should never be standardized, so we need to create the polynomials with `include_bias=False` and instead estimate the model with an intercept by creating `LinearRegression(fit_intercept=True)`.

```
[37]: from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold
from sklearn.metrics import root_mean_squared_error

degrees = np.arange(0, 6)

rmse_train_mean = []
rmse_test_mean = []

for d in degrees:
    # Create pipeline to transform and fit the model. Pipeline depends
    # on polynomial degree!
    if d == 0:
        # Intercept-only model
        pipe = make_pipeline(
            PolynomialFeatures(degree=d, include_bias=True),
            LinearRegression(fit_intercept=False),
        )
    else:
        # Model with polynomial terms: need to standardize features for
        # numerical stability
        pipe = make_pipeline(
            PolynomialFeatures(degree=d, include_bias=False),
            StandardScaler(),
            LinearRegression(fit_intercept=True),
        )

    rmse_train_fold = []
    rmse_test_fold = []

    # Create 10 folds
    folds = KFold(n_splits=10)

    # Iterate through all combinations of training/test data
    for itrain, itest in folds.split(X, y):

        # Extract training data for this split
        X_train = X.iloc[itrain]
        y_train = y.iloc[itrain]

        # Extract test (or validation) data for this split
        X_test = X.iloc[itest]
        y_test = y.iloc[itest]

        # Fit model on training data for given degree
        pipe.fit(X_train, y_train)

        # Predict response on train and test data
        rmse_train_fold.append(pipe.predict(X_train).rmean_squared_error(y_train))
        rmse_test_fold.append(pipe.predict(X_test).rmean_squared_error(y_test))

    rmse_train_mean.append(np.mean(rmse_train_fold))
    rmse_test_mean.append(np.mean(rmse_test_fold))
```

```

y_train_hat = pipe.predict(X_train)
y_test_hat = pipe.predict(X_test)

# Compute RMSE as model fit measure
rmse_train = root_mean_squared_error(y_train, y_train_hat)
rmse_test = root_mean_squared_error(y_test, y_test_hat)

# Store RMSE for current split
rmse_train_fold.append(rmse_train)
rmse_test_fold.append(rmse_test)

# Store average MSE for current polynomial degree
rmse_train_mean.append(np.mean(rmse_train_fold))
rmse_test_mean.append(np.mean(rmse_test_fold))

# Convert to NumPy arrays
rmse_train_mean = np.array(rmse_train_mean)
rmse_test_mean = np.array(rmse_test_mean)

# Print average RMSE for all polynomial degrees
rmse_test_mean

```

[37]: array([78.10225405, 54.06796801, 53.7579006 , 53.52612724, 55.57874839, 67.2402846])

This code returns an array of 5 averaged RMSEs, one for each $d = 0, \dots, 5$. We can now find the optimal d by picking the one which has the lowest mean squared error in the test samples using the `argmin()` function which returns the *index* of the smallest array element.

[38]:

```

# Find index of polynomial degree with smallest RMSE
imin = np.argmin(rmse_test_mean)

# RMSE-minimising polynomial degree
dmin = degrees[imin]

print(f'Polynomial degree with minimum RMSE: {dmin}')

```

Polynomial degree with minimum RMSE: 3

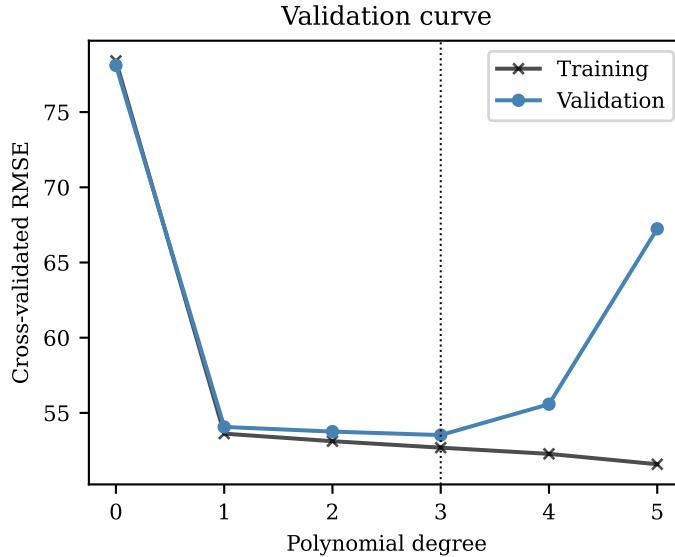
Finally, it is often instructive to visualize how the RMSE evolves as a function of the hyperparameter we want to tune.

[39]:

```

plt.plot(degrees, rmse_train_mean, c='black', marker='x', ms=4, alpha=0.7,
         label='Training')
plt.plot(degrees, rmse_test_mean, c='steelblue', marker='o', ms=4, label='Validation')
plt.xlabel('Polynomial degree')
plt.ylabel('Cross-validated RMSE')
plt.xticks(degrees)
plt.axvline(imin, ls=':', lw=0.75, c='black')
plt.title('Validation curve')
_ = plt.legend()

```



Here we see that for $d = 0$ (the intercept-only model), the model underfits the data leading to a high prediction error. However, higher d 's do not always translate into a better fit. For $d = 5$, the model vastly overfits the data, resulting in poor performance in the test sample and a very large RMSE.

1.3.3 Automating cross-validation

Automatically computing scores on all folds

The code implemented above to run cross-validation was needlessly complex even though we leveraged the `KFold` class to do the sample splitting for us. Fortunately, scikit-learn provides us with even more convenience functions that further simplify this process. For example, if we want to perform tuning based on a single score (such as the root mean squared error), we can instead use `cross_val_score()`. This function requires us to specify an estimator (or a pipeline), the number of folds to use for cross-validation (`cv=10`) and the metric to evaluate. For example, if we want to compute the RMSE, we would pass the argument `scoring='neg_root_mean_squared_error'`. Note that the function returns the *negative* RMSE which we need to correct manually.

See the [documentation](#) for a complete list of valid metrics that can be passed as `scoring` arguments. Alternatively, scikit-learn lists available metrics by running

```
import sklearn.metrics
sklearn.metrics.get_scoring_names()
```

The following code re-implements the k-fold cross-validation from earlier using `cross_val_score()`:

```
[40]: from sklearn.model_selection import cross_val_score

degrees = np.arange(6)

rmse_mean = []
rmse_std = []

for d in degrees:

    # Create pipeline to transform and fit the model. Pipeline depends
    # on polynomial degree!
    if d == 0:
        # Intercept-only model
        pipe = make_pipeline(
            PolynomialFeatures(degree=d, include_bias=True),
```

```

        LinearRegression(fit_intercept=False),
    )
else:
    # Model with polynomial terms: need to standardize features for
    # numerical stability
    pipe = make_pipeline(
        PolynomialFeatures(degree=d, include_bias=False),
        StandardScaler(),
        LinearRegression(fit_intercept=True),
    )

# Compute negative RMSE for all 10 folds
score = cross_val_score(
    pipe,
    X, y,
    scoring='neg_root_mean_squared_error',
    cv=10
)

# Function returns NEGATIVE RMSE, correct this here!
rmse_mean.append(np.mean(-score))

# Convert to NumPy array
rmse_mean = np.array(rmse_mean)

# Print average RMSE for all polynomial degrees
rmse_mean

```

[40]: `array([78.10225405, 54.06796801, 53.7579006 , 53.52612724, 55.57874839,
67.2402846])`

The RMSE-minimizing degree is of course the same as before:

[41]: `imin = np.argmin(rmse_mean)
dmin = degrees[imin]

print(f'Polynomial degree with min. RMSE: {dmin}')`

`Polynomial degree with min. RMSE: 3`

Automatically computing the validation curve

scikit-learn provides an even more automated API to perform the cross-validation using the function `validation_curve()`. To compute some score such as the RMSE for all folds at once, we don't even have to loop over the hyperparameter of interest (in this case the polynomial degree), but can leave this to `validation_curve()`. This process works as follows:

1. We create an instance of an estimator (or estimator pipeline) to be used for the cross-validation.
2. We specify which parameter should be cross-validated (using the argument `param_name`) and the set of candidate values to be used for cross-validation (argument `param_range`).
3. The function `validation_curve()` returns the desired score, which needs to be specified using the `scoring` argument. The list of acceptable scorers is the same as in the previous section.

To demonstrate, we recreate the validation curve from the earlier example, but we restrict our attention to polynomial degrees $d > 0$ as we then don't have to create a separate pipeline for the intercept-only model.

[42]: `from sklearn.model_selection import validation_curve

Candidate polynomial degrees (ignore intercept-only model)`

```

degrees = np.arange(1, 6)

# Create pipeline for polynomial features and linear regression.
# The degree of the polynomial DOES NOT MATTER at this point.
pipe = Pipeline(steps=[
    ('poly', PolynomialFeatures(degree=0, include_bias=False)),
    ('scale', StandardScaler()),
    ('lr', LinearRegression(fit_intercept=True))
])

# Compute the MSEs for each degree and each split, returning arrays of
# size len(degrees) x 10.
train_scores, test_scores = validation_curve(
    estimator=pipe,
    X=X, y=y,
    param_name='poly_degree',
    param_range=degrees,
    scoring='neg_root_mean_squared_error',
    cv=10
)

```

[43]: # Compute mean for each degree (scores returned by function are
NEGATIVE RMSEs)

```

rmse_train_mean = np.mean(-train_scores, axis=1)
rmse_test_mean = np.mean(-test_scores, axis=1)

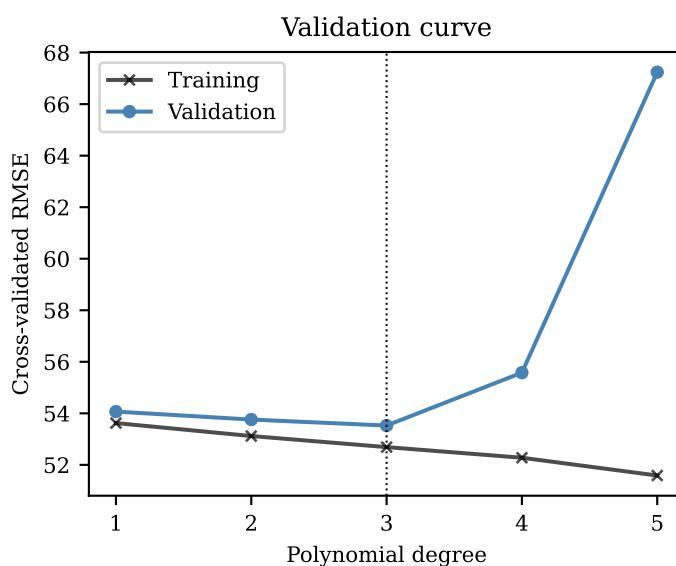
```

[44]:

```

plt.plot(degrees, rmse_train_mean, c='black', marker='x', ms=4, alpha=0.7,
         label='Training')
plt.plot(degrees, rmse_test_mean, c='steelblue', marker='o', ms=4, label='Validation')
plt.xlabel('Polynomial degree')
plt.ylabel('Cross-validated RMSE')
plt.xticks(degrees)
plt.axvline(imin, ls=':', lw=0.75, c='black')
plt.title('Validation curve')
_ = plt.legend()

```



1.4 Advanced pipelines and cross-validation: Principal component regression (PCR)

We close the section on cross-validation and hyperparameter tuning with a more complex example which also demonstrates a more advance use of `scikit-learn` pipelines.

To this end, we implement principal component regression (PCR) with `scikit-learn`. The idea behind PCR is that we can address issues of overfitting and multicollinearity inherent in linear models without regularization by only using a subset of the variation in the explanatory variables. This subset is obtained by using only a few principal components (see [Principal component analysis](#)) which capture most of the variation in the data. You can think of principal component analysis as a method feature extraction, as opposed to feature selection where we manually pick only a few explanatory variables.

```
[45]: # Load Ames data with additional features
features = ['LivingArea', 'LotArea', 'YearBuilt', 'OverallQuality']
target = 'SalePrice'

X, y = load_ames_data(features, target)
```

Fitting the principal component regression

We first create a pipeline with four steps:

1. As before, we create polynomial interactions of all explanatory variables using [PolynomialFeatures](#).
2. We standardize the features created in the first step. This is crucial for principal component analysis as it is very sensitive to data scaling.
3. We extract only a few principal components using [PCA](#). The most important argument here is `n_components`, which can be either an integer specifying the number of components to use, or a floating point number in the open interval (0, 1) which selects the number of components such that the amount of variance explained is greater than the value of `n_components`.
4. The last step is a [LinearRegression](#), as before. It is important to specify `fit_intercept=True` in this step since all variables have been standardized prior to the PCA step, and hence the model wouldn't be able to match the mean of the target variable without an intercept.

Note that we use [Pipeline](#) to create the pipeline instead of the convenience function `make_pipeline()` so that we can explicitly name the pipeline steps. This will come in handy later.

```
[46]: from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.decomposition import PCA

# Create pipeline for principal component regression
pipe_pcr = Pipeline(
    steps=[
        ('poly', PolynomialFeatures(degree=5, include_bias=False)),
        ('scale', StandardScaler()),
        ('pca', PCA(n_components=0.9)),
        ('lr', LinearRegression(fit_intercept=True)),
    ]
)
```

With the pipeline in hand, we can fit the model (for this exercise, we ignore the train-test split):

```
[47]: pipe_pcr.fit(X, y)
```

```
[47]: Pipeline(steps=[('poly', PolynomialFeatures(degree=5, include_bias=False)),
                     ('scale', StandardScaler()), ('pca', PCA(n_components=0.9)),
                     ('lr', LinearRegression())])
```

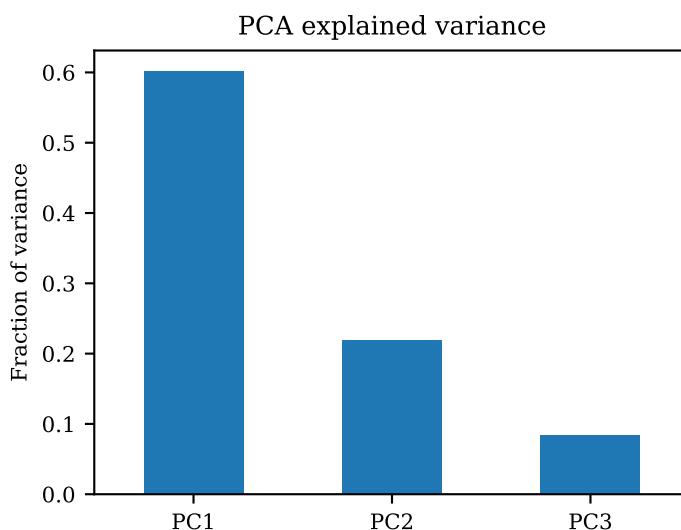
We can use the named pipeline steps to extract information about the data transformation in the [PCA](#) layer. For example, we can extract the fraction of variance explained by each component and plot it as follows:

```
[48]: # Extract fraction of variance explained by each principal component.
# Note that the name of the pipeline step is the one assigned when creating the
# pipeline.
var_expl = pipe_pcr.named_steps['pca'].explained_variance_ratio_
var_expl
```

```
[48]: array([0.60110876, 0.21964345, 0.08389025])
```

```
[49]: # Use pandas plotting functions to create bar plot
pd.Series(var_expl, index=[f'PC{i+1}' for i in range(len(var_expl))]).plot.bar(
    ylabel='Fraction of variance', title='PCA explained variance', rot=0
)
```

```
[49]: <Axes: title={'center': 'PCA explained variance'}, ylabel='Fraction of variance'>
```



Cross-validation for principal component regression

Finally, we can perform hyperparameter tuning for the principal component regression as well. This is slightly complicated by the fact that we now have *two* hyperparameters:

1. The maximum polynomial degree used to create the polynomial features.
2. The number of principal components to use for PCR (or equivalently, the share of explained variance).

For this case, scikit-learn provides the `GridSearchCV` class which performs grid search over a set of parameters which are specified using the `param_grid` argument. This argument is expected to be a dictionary with the parameter names as keys.

Note that the parameter names are composed of two parts:

1. the name of the pipeline step that should be tuned; and
2. the name of the attribute to be varied.

These need to be separated by ' __ ' so that the parameter name is STEP__ATTRIBUTE.

```
[50]: from sklearn.model_selection import GridSearchCV
import numpy as np

# Define grid of hyperparameters
param_grid = {
```

```

'poly_degree': np.arange(1, 6),
'pca_n_components': np.linspace(0.9, 0.999999, 20),
}

# Create GridSearchCV object
pqr_cv = GridSearchCV(
    estimator=pipe_pqr,                                # Estimator pipeline
    param_grid=param_grid,                            # Grid of hyperparameters
    scoring='neg_root_mean_squared_error',           # Scoring criterion
    cv=10,                                         # Number of folds for CV
    n_jobs=-1,                                       # Use all available CPU cores
    refit=True                                       # Refit model with best hyperparameters
)

# Run grid search
pqr_cv.fit(X, y)

```

[50]:

```
GridSearchCV(cv=10,
            estimator=Pipeline(steps=[('poly',
                                         PolynomialFeatures(degree=5,
                                                             include_bias=False)),
                                         ('scale', StandardScaler()),
                                         ('pca', PCA(n_components=0.9)),
                                         ('lr', LinearRegression())]),
            n_jobs=-1,
            param_grid={'pca_n_components': array([0.9      , 0.90526311,
0.91052621, 0.91578932, 0.92105242,
0.92631553, 0.93157863, 0.93684174, 0.94210484, 0.94736795,
0.95263105, 0.95789416, 0.96315726, 0.96842037, 0.97368347,
0.97894658, 0.98420968, 0.98947279, 0.99473589, 0.999999  ]),
            'poly_degree': array([1, 2, 3, 4, 5])},
            scoring='neg_root_mean_squared_error')
```

Once grid search is complete, we can obtain the best parameter combination from the `best_params_` attribute, and the best score (negative RMSE in this case) from the `best_score_` attribute:

[51]:

```
# Extract best cross-validated hyperparameters
pqr_cv.best_params_
```

[51]:

```
{'pca_n_components': np.float64(0.999999), 'poly_degree': np.int64(3)}
```

[52]:

```
# Print RMSE (averaged over folds) for the best model
print(f'Best RMSE: {-pqr_cv.best_score_:.3f}')
```

Best RMSE: 28.961

Moreover, from the PCA step we can read off how many components were selected in the process:

[53]:

```
# Number of total polynomial features in the best model
features_in = pqr_cv.best_estimator_.named_steps['pca'].n_features_in_
# Number of principal components in the best model
n_components = pqr_cv.best_estimator_.named_steps['pca'].n_components_

print(f'Number of polynomial features: {features_in}')
print(f'Number of principal components: {n_components}')
```

Number of polynomial features: 34
Number of principal components: 24

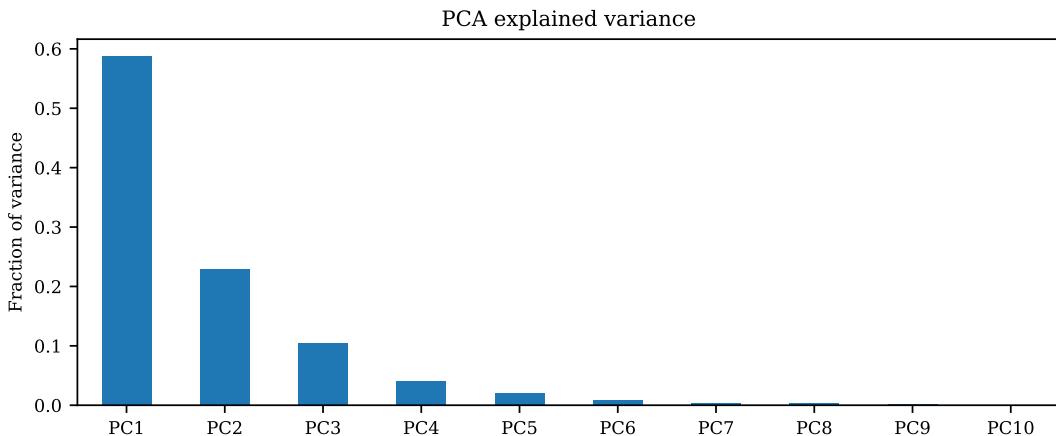
Last, we recreate the bar chart of the variance explained by each component in the best model:

[54]:

```
var_expl = pqr_cv.best_estimator_.named_steps['pca'].explained_variance_ratio_[:10]
pd.Series(var_expl, index=[f'PC{i+1}' for i in range(len(var_expl))]).plot.bar()
```

```
        ylabel='Fraction of variance', title='PCA explained variance', rot=0,
        figsize=(8, 3)
    )
```

```
[54]: <Axes: title={'center': 'PCA explained variance'}, ylabel='Fraction of variance'>
```



Your turn.

1. Add the variables `YearRemodeled`, `Bedrooms`, and `Bathrooms` to the feature set and rerun the cross-validation.
2. What happens to the RMSE and the number of principal components of the best model?

1.5 Dealing with categorical data

So far in this unit, we only dealt with continuous data, i.e., data that can take on (almost any) real value. However, many data sets contain categorical variables which take on a finite number of admissible values or even binary variables (often called dummy or indicator variables) which can be either 0 or 1. Dealing with such data in `scikit-learn` is less straightforward than in most other software packages, so in this section we illustrate how to work with categorical variables.

To do this, we again load the Ames house data which contains several categorical and binary variables.

```
[55]: import pandas as pd
df = pd.read_csv(f'{DATA_PATH}/ames_houses.csv')
```

For example, consider the `BuildingType` variable, which (in this simplified version of the data) takes on three values, 'Single-family', 'Townhouse' and 'Two-family' and has on top a few missing observations. We use the `value_counts()` method to tabulate the number of observations falling into each category. The argument `dropna=False` also includes the number of missing values in the tabulation.

```
[56]: df['BuildingType'].value_counts(dropna=False).sort_index()
```

```
[56]: BuildingType
Single-family    2425
Townhouse       233
Two-family      109
NaN              163
Name: count, dtype: int64
```

There are several ways to deal with such data (see the official [documentation](#) for details). First, since the data are stored as strings, we could use `OrdinalEncoder` to map these strings to integers.

```
[57]: from sklearn.preprocessing import OrdinalEncoder

# Drop rows with NA
df = df.dropna(subset=['BuildingType'])

enc = OrdinalEncoder()

# Transform string variable into integers
bldg_int = enc.fit_transform(df[['BuildingType']])

# Print unique values and histogram
print(f'Unique values: {np.unique(bldg_int)}')
print(f'Histogram: {np.bincount(bldg_int.astype(int).flatten())}')


Unique values: [0. 1. 2.]
Histogram: [2425 233 109]
```

This, however, is still not particularly useful if we want to use these categories as explanatory variables in a `LinearRegression` because scikit-learn will simply treat them as a continuous variable that happens to take on the values 0, 1 or 2. There is nothing to enforce the categorical nature of this data.

An alternative encoding strategy is to create a binary dummy variable for each possible value of a categorical variable. This is achieved using the `OneHotEncoder` transformation as illustrated by the following code:

```
[58]: from sklearn.preprocessing import OneHotEncoder

# Drop rows with NA
df = df.dropna(subset=['BuildingType'])

# List of unique categories
bldg_uniq = list(np.sort(df['BuildingType'].unique()))

# Create dummy variable encoder
enc = OneHotEncoder(categories=[bldg_uniq], sparse_output=False)

# Convert string variable into binary indicator variables
bldg_dummies = enc.fit_transform(df[['BuildingType']])
```

When creating a `OneHotEncoder`, we can optionally pass the list of possible values using the `categories` argument. By default, this transformation returns a sparse matrix since each row will have exactly one element that is 1 while the remaining elements are 0. Using a sparse matrix saves memory but makes interacting with the resulting array more cumbersome. For small data sets, there is no need to use sparse matrices.

The transformed data now looks as follows:

```
[59]: # Print first 5 rows
bldg_dummies[:5]
```

```
[59]: array([[1., 0., 0.],
           [1., 0., 0.],
           [1., 0., 0.],
           [1., 0., 0.],
           [1., 0., 0.]])
```

In this particular example, the first five observations fall into the first category, hence the first column contains ones while the remaining elements are zero.

We can sum the number of ones in each column to verify that the frequency of each category remains the same:

```
[60]: bldg_dummies.sum(axis=0)
```

```
[60]: array([2425., 233., 109.])
```

The category labels taken from the original data are stored in the `categories_` attribute:

```
[61]: enc.categories_[0]
```

```
[61]: array(['Single-family', 'Townhouse', 'Two-family'], dtype=object)
```

Now that we have converted the categories into a dummy matrix, we can append it to the continuous explanatory variables and fit the linear model as usual:

```
[62]: from sklearn.linear_model import LinearRegression
import pandas as pd

# Name of dependent variable
target = 'SalePrice'
# Continuous explanatory variables to include
continuous = ['LivingArea', 'LotArea']

y = df[target].to_numpy()
# Feature matrix with dummies appended
X = np.hstack((df[continuous].to_numpy(), bldg_dummies))

# Create and fit linear model
lr = LinearRegression(fit_intercept=False)
lr.fit(X, y)

# Create DataFrame containing estimated coefficients
labels = continuous + [f'BuildingType: {s}' for s in enc.categories_[0]]

coefs = pd.DataFrame(lr.coef_, index=labels, columns=['Coef'])
coefs
```

```
[62]:          Coef
LivingArea      1.214031
LotArea         0.009499
BuildingType: Single-family  3.979294
BuildingType: Townhouse     36.668665
BuildingType: Two-family    -55.219532
```

Finally, you should be aware that pandas provides the function `get_dummies()` which accomplishes something very similar to OneHotEncoder but may not work as well with the scikit-learn API.

```
[63]: pd.get_dummies(df['BuildingType']).head(5)
```

```
[63]:   Single-family  Townhouse  Two-family
0            True    False     False
1            True    False     False
2            True    False     False
3            True    False     False
4            True    False     False
```

1.6 Optional exercises

Exercise 1: Optimizing hyperparameters with validation curves

In the lecture, we explored several ways to perform cross-validation. In this exercise, you are asked to use one of these methods to tune the polynomial degree of a model estimated on synthetic data.

Consider the following model where y is a non-linear trigonometric function of x and includes an additive error term ϵ ,

$$y_i = f(x_i) + \epsilon_i, \quad f(x) = \sin(2\pi(x + 0.1))$$

and x and ϵ are assumed to be independent.

1. Create a sample of $N = 100$ realizations of (x_i, ϵ_i) using a seed 1234 for the random number generator. Draw the x_i from the standard uniform distribution on $[0, 1]$ and let $\epsilon \stackrel{\text{iid}}{\sim} N(0, 0.5^2)$. Compute y_i according to the equation above.

Hint: The sine function and the constant π are implemented as `np.sin()` and `np.pi` in NumPy.

2. Create a sample scatter plot of (x_i, y_i) and add a line showing the true relationship between x and y (without measurement error).
3. Assume you don't know the true model and you want to fit the target y to a polynomial in x . To this end, create a `Pipeline` that performs the polynomial transformation and the fitting of the linear model. You can use any value for `PolynomialFeatures(degree=...)` at this point.
4. Use the convenience function `validation_curve()` to compute the MSE for each $d = 0, \dots, 15$ and 10 cross-validation splits in a single call:

```
train_scores, test_scores = validation_curve(
    estimator=...,
    ...,
    param_name='polynomialfeatures__degree',
    param_range=degrees,
    scoring='neg_mean_squared_error',
    cv=10
)
```

To do this, the argument `estimator` specifies the model to use (i.e., the pipeline you created in the previous part), `param_name` specifies which parameter needs to be varied, and `param_range` determines the set of parameter values that are evaluated. The `scoring` argument determines which metric is computed and returned.

Since our estimator is a pipeline, the parameter name needs to be specified as '`STEP__ARGUMENT`' where `STEP` is the name of the step in the pipeline we defined (see `Pipeline` for details), and `ARGUMENT` determines the argument name to be varied when re-creating the pipeline, which in our case is `degree`.

Hint: For pipelines created using `make_pipeline()`, the name of each step in the pipeline is the class name converted to lower case, i.e., for `PolynomialFeatures` the step name is `polynomialfeatures`. In this case, `validation_curve()` needs to be called with the argument `param_name='polynomialfeatures__degree'`.

5. `validation_curve()` returns two arrays which contain the scores computed on the training and test sets, respectively. The first dimension of each array corresponds to each parameter from `param_range` and the second dimension corresponds to individual folds used in the cross-validation.

Compute the average MSE for each polynomial degree. Report the polynomial degree at which this metric is minimized, and plot the average MSE on the y -axis against all polynomial degrees on the x -axis to visualize your results.

6. Recreate the graph from Part (2) and add the predicted values from the model with optimal polynomial degree you found.

The scikit-learn [documentation](#) provides additional information on validation curves which you might want to consult.

1.7 Solutions

Exercise 1: Optimizing hyperparameters with validation curves

Part 1 — Creating the sample

We first define the function $f(x)$ that represents the true relationship.

```
[64]: import numpy as np

# True function (w/o errors)
def fcn(x):
    return np.sin(2.0 * np.pi * (x + 0.1))
```

We then draw a sample of size 100 using the respective distributions for x_i and ϵ_i .

```
[65]: from numpy.random import default_rng

# Initialise random number generator
rng = default_rng(seed=1234)

# Sample size
N = 100

# Randomly draw explanatory variable x
x = rng.uniform(size=N)
epsilon = rng.normal(scale=0.5, size=N)
y = fcn(x) + epsilon
```

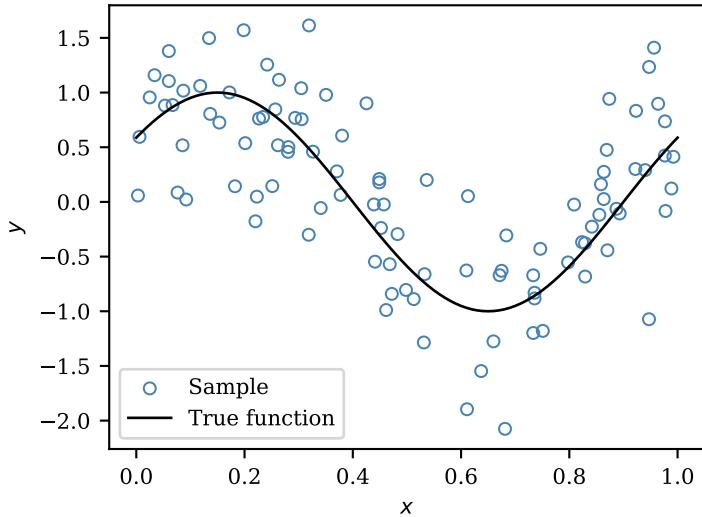
Part 2 — Plotting the sample

```
[66]: import matplotlib.pyplot as plt

# Plot raw data
plt.scatter(x, y, s=20, color='none', edgecolor='steelblue', lw=0.75, label='Sample')

# Plot true function on uniformly spaced grid
xvalues = np.linspace(0.0, 1.0, 101)
plt.plot(xvalues, fcn(xvalues), color='black', lw=1.0, label='True function')

plt.xlabel('$x$')
plt.ylabel('$y$')
_ = plt.legend()
```



Part 3 — Creating a pipeline

We can create a pipeline using two equivalent ways, using either `make_pipeline()` or `Pipeline`.

```
[67]: from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

# Create pipeline. The degree value does not matter at this point, will be
# automatically updated.
pipe = make_pipeline(
    PolynomialFeatures(
        degree=0,
        include_bias=True
    ),
    LinearRegression(fit_intercept=False)
)
```

Alternatively, we could have created the pipeline using `Pipeline` which allows us to explicitly assign names to each pipeline step. We choose names which are the same as the default values assigned by `make_pipeline()`:

```
[68]: from sklearn.pipeline import Pipeline

# Create pipeline, explicitly specifying step names
pipe = Pipeline(
    [
        ('polynomialfeatures', PolynomialFeatures(degree=0, include_bias=True)),
        ('linearregression', LinearRegression(fit_intercept=False)),
    ]
)
```

Part 4 — Computing the MSE for each hyperparameter

To use `validation_curve()`, we need to correctly specify the `param_name` argument. In our case this is given by '`polynomialfeatures__degree`' since we are asking the function to vary the argument `degree` of the pipeline step called `polynomialfeatures`. Note that the `__` serves as a delimiter between the pipeline step name and the argument name.

```
[69]: from sklearn.model_selection import validation_curve

# Set of polynomial degrees to cross-validate
degrees = np.arange(16)

# Compute the MSEs for each degree and each split, returning arrays of
# size 15 x 10.
train_scores, test_scores = validation_curve(
    estimator=pipe,
    X=x[:, None], y=y,
    param_name='polynomialfeatures__degree',
    param_range=degrees,
    scoring='neg_mean_squared_error',
    cv=10
)

# Compute mean and std for each degree (scores returned by function are
# NEGATIVE MSEs)
train_mse_mean = np.mean(-train_scores, axis=1)
train_mse_std = np.std(-train_scores, axis=1)
test_mse_mean = np.mean(-test_scores, axis=1)
test_mse_std = np.std(-test_scores, axis=1)
```

Part 5 — Determine optimal polynomial degree

The optimal cross-validated polynomial degree is the one at which the average MSE is minimized:

```
[70]: imin = np.argmin(test_mse_mean)
dmin = degrees[imin]

print(f'Polynomial degree with min. MSE: {dmin}')
```

Polynomial degree with min. MSE: 5

```
[71]: plt.figure(figsize=(5,3.5))
plt.plot(degrees, train_mse_mean, c='black', lw=0.75, marker='x', ms=4, label='Training')
plt.fill_between(
    degrees,
    train_mse_mean - train_mse_std,
    train_mse_mean + train_mse_std,
    lw=0.25,
    color='black',
    alpha=0.2,
    zorder=-1,
)

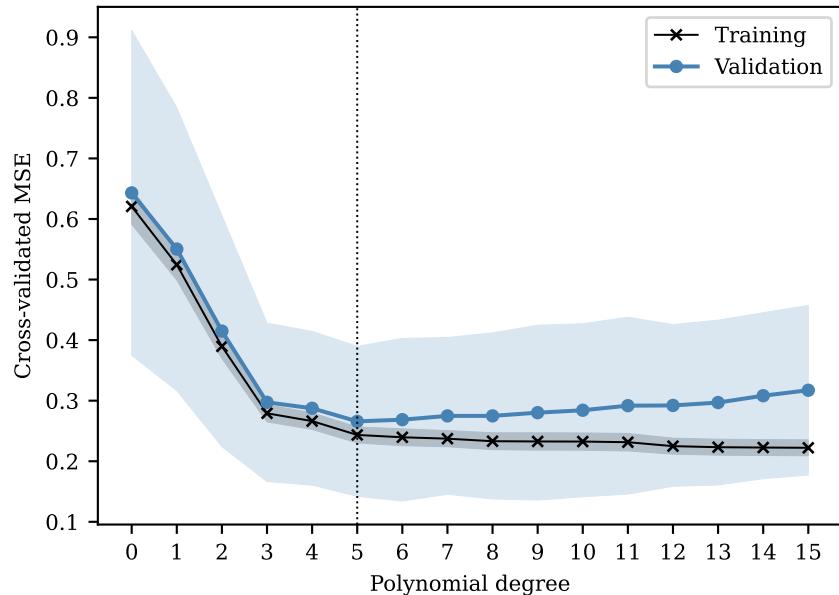
plt.plot(degrees, test_mse_mean, c='steelblue', marker='o', ms=4, label='Validation')
plt.fill_between(
    degrees,
    test_mse_mean - test_mse_std,
    test_mse_mean + test_mse_std,
    lw=0.25,
    color='steelblue',
    alpha=0.2,
    zorder=-1,
)

plt.xlabel('Polynomial degree')
plt.ylabel('Cross-validated MSE')
# plt.scatter(degrees[imin], test_mse_mean[imin], s=15, c='black', zorder=100)
plt.xticks(degrees)
```

```

plt.axvline(imin, ls=':', lw=0.75, c='black')
_ = plt.legend(loc='upper right')

```



Part 6 — Plot predicted values using optimal hyperparameter

We need to re-fit the model using the optimal polynomial degree. For this, we recreate the pipeline, fit the model and compute the model prediction on a grid of x -values.

```

[72]: pipe = make_pipeline(
    PolynomialFeatures(degree=dmin, include_bias=True),
    LinearRegression(fit_intercept=False),
)

pipe.fit(x[:, None], y)

# X-values used for prediction
xvalues = np.linspace(0.0, 1.0, 101)
# Fit model
y_hat = pipe.predict(xvalues[:, None])

# Plot true relationship (w/o errors)
plt.plot(xvalues, fcn(xvalues), color='black', lw=1.0, label='True function')

# Plot best fit
plt.plot(xvalues, y_hat, color='darkorange', lw=1.25, zorder=10, label='Best fit')

# Plot sample
plt.scatter(
    x, y, s=20, color='none', edgecolor='steelblue', lw=0.75, alpha=0.8, label='Sample'
)

plt.xlabel('$x$')
plt.ylabel('$y$')
plt.ylim(-1.5, 2.0)
_ = plt.legend(loc='upper right')

```

