

TECH6: Lecture notes

Richard Foltyn

NHH Norwegian School of Economics

November 2026

See GitHub repository for notebooks and data:

<https://github.com/richardfoltyn/TECH6-H26>

License: This material is licensed under a Creative Commons Attribution 4.0 International License. See <https://creativecommons.org/licenses/by/4.0/> for detailed license terms.

Contents

1	Introduction to scikit-learn	1
1.1	Simple linear regression	1
1.2	Multiple linear regression	14
1.3	Dealing with categorical data	25
1.4	List of functions used in this lecture	31
2	Cross-validation and model selection	33
2.1	Motivation	33
2.2	Optimizing hyperparameters with cross-validation	33
2.3	Principal component regression	44
2.4	List of functions used in this lecture	53
3	Regression models with scikit-learn	55
3.1	Motivation	55
3.2	Synthetic sample and polynomial approximation	55
3.3	Ridge regression	59
3.4	Lasso	67
3.5	Elastic Net	76
3.6	Comparing predictions on new observations	80
3.7	List of functions used in this lecture	81
4	Classification models with Scikit-learn	83
4.1	Motivation	83
4.2	Example data for classification	83
4.3	Logistic regression	87
4.4	Decision trees	101
4.5	Random forest	104
4.6	Comparing predictions on new observations	107
4.7	Support vector machines (SVM)	108
4.8	List of functions and classes used in this lecture	113

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 dataset. 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.

In this lecture, we cover ordinary least squares (OLS), a regression method used in both econometrics and machine learning. It provides a straightforward introduction to scikit-learn using a model most students are familiar with. In the following lectures, we study regularization, which adds a penalty term to the OLS loss function to improve prediction on new data.

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 Simple 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

$$y_i = b + wx_i + \epsilon_i$$

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

$$\hat{\beta} = \frac{\widehat{Cov}(y, x)}{\widehat{Var}(x)}$$

$$\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}$$

where $\widehat{Cov}(\bullet, \bullet)$ and $\widehat{Var}(\bullet)$ 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 multiple linear regression, 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 Simple linear regression 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

$$y_i = 1 + \frac{1}{2}x_i + \epsilon_i$$

$$\epsilon_i \stackrel{\text{iid}}{\sim} N(0, \sigma^2)$$

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 first define a reusable function that can generate samples of a desired size:

```
[1]: %matplotlib inline

# Imports used in the notebook
import matplotlib.pyplot as plt
import numpy as np

# Default plot fonts
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.size'] = 10
```

```
[2]: def create_sample(N, alpha, beta, sigma, rng=None):
    """
    Create sample for simple linear regression

    Parameters
    -----
    N : int
        Sample size
    alpha : float
```

```

    Intercept (or bias)
    beta : float
    Slope (or weight)
    sigma : float
    Standard deviation of the error term
    rng : np.random.Generator, optional
    Random number generator

Returns
-----
x : array
    Predictor variable, uniformly distributed on [0, 10]
y : array
    Outcome variable
"""
from numpy.random import default_rng

# Use a fixed seed when no generator is supplied
if rng is None:
    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

```

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

```

[3]: # Sample size
N = 50

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

# Create sample
x, y = create_sample(N=N, alpha=alpha, beta=beta, sigma=sigma)

```

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 $X = x[:, \text{None}]$.

```

[4]: 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)

```

Linear models in scikit-learn, such as `LinearRegression`, store the fitted slope coefficients 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_`:

```
[5]: # 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
```

Note that the attributes `coef_` and `intercept_` do not exist before the `fit()` method has been called, and attempting to access them will raise an `AttributeError`.

The following function 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 $\hat{\beta}$.

```
[6]: def plot_regression(x, y, alpha, beta, alpha_hat, beta_hat, title=None):
    """
    Plot observations, true and fitted regression lines, and residuals.

    Parameters
    -----
    x : array
        Explanatory variable
    y : array
        Observed response variable
    alpha : float
        True intercept
    beta : float
        True slope
    alpha_hat : float
        Estimated intercept
    beta_hat : float
        Estimated slope
    title : str, optional
        Title for the plot
    """
    import matplotlib.pyplot as plt

    yhat = alpha_hat + beta_hat * x

    _fig, ax = plt.subplots(figsize=(5, 3.5), layout='constrained')

    # Plot observations
    ax.scatter(x, y, s=20, c='none', ec='steelblue', alpha=1.0, lw=0.75, label='y')

    # Connect observed and fitted values
    for i in range(len(x)):
        ax.plot([x[i], x[i]], [y[i], yhat[i]], lw=0.5, ls=':', c='black', alpha=0.9)

    # Plot true relationship
    label = r'$y = \alpha + \beta x$'
    ax.axline((0.0, alpha), slope=beta, lw=1.5, ls='--', c='black', label=label)

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

    # Add annotations
```

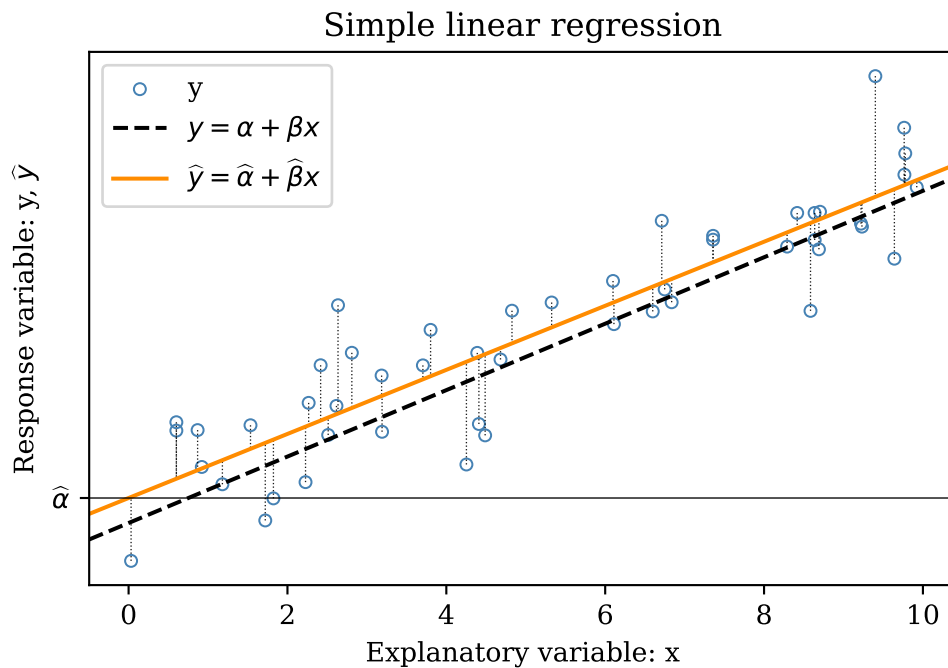
```

ax.set_xlabel('Explanatory variable: x')
ax.set_ylabel(r'Response variable: y, $\widehat{y}$')
ax.axhline(alpha_hat, lw=0.5, c='black')
ax.legend(loc='best')
ax.set_yticks((alpha_hat,), (r'$\widehat{\alpha}$',))
ax.set_title(title or 'Simple linear regression')

```

We use this function to visualize the true relationship and the fitted line:

```
[7]: plot_regression(x, y, alpha=alpha, beta=beta, alpha_hat=alpha_hat, beta_hat=beta_hat)
```



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. Write a loop to 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 Digression: Machine learning versus econometrics

At this point it's instructive to revisit the difference between the machine learning approach in scikit-learn vs. the traditional econometric approach. To illustrate this, we use [statsmodels](#) to run the regression

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

using the same data as above:

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

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

# Create sample
x, y = create_sample(N=N, alpha=alpha, beta=beta, sigma=sigma)

# Refit scikit-learn on the same sample used by statsmodels
_ = lr.fit(x[:, None], y)
```

To use the statsmodels API, we first import it from `statsmodels.api`. We add a constant column for the intercept and fit the same model as above:

```
[9]: import statsmodels.api as sm

# Add constant for intercept
X_with_const = sm.add_constant(x)

# Fit OLS regression
model = sm.OLS(y, X_with_const)
results = model.fit()

print(results.summary())
print('statsmodels coefficients:', results.params)
print('scikit-learn coefficients:', lr.intercept_, lr.coef_[0])
```

```

                        OLS Regression Results
=====
Dep. Variable:          y      R-squared:                0.814
Model:                  OLS      Adj. R-squared:          0.810
Method:                 Least Squares      F-statistic:        209.7
Date:                   Sat, 31 Oct 2026      Prob (F-statistic):    3.86e-19
Time:                   22:15:18      Log-Likelihood:       -54.171
No. Observations:       50      AIC:                  112.3
Df Residuals:           48      BIC:                  116.2
Df Model:               1
Covariance Type:        nonrobust
=====
                        coef      std err          t      P>|t|      [0.025      0.975]
-----
const                1.3762      0.204        6.758      0.000        0.967        1.786
x1                   0.4822      0.033       14.480      0.000        0.415        0.549
=====
Omnibus:                0.167      Durbin-Watson:        2.349
Prob(Omnibus):           0.920      Jarque-Bera (JB):      0.014
Skew:                    0.039      Prob(JB):              0.993
Kurtosis:                2.981      Cond. No.              12.3
=====
```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

statsmodels coefficients: [1.37615964 0.48222474]

scikit-learn coefficients: 1.3761596402147696 0.4822247386261906

As you can see from the regression table, the estimated coefficients for α (labeled const) and β (labeled x1) are identical to those computed by scikit-learn. The difference is that the econometrics package puts more emphasis on inference, i.e., reporting standard errors (column `std err`), t-statistics (column `t`), the F-statistic, and other metrics that allow you to test hypotheses, for example whether any individual coefficient is statistically different from zero. This aspect is completely missing from scikit-learn because there the focus is on fitting models that perform well in out-of-sample prediction.

The fitted result stores the coefficients in `results.params` and standard errors in `results.bse`. Its `results.conf_int(alpha=0.05)` method returns the lower and upper endpoints of 95% confidence intervals. Row 0 is the intercept; row 1 is the slope.

Your turn

Estimate the statsmodels regression for samples of size 10, 20, 50, 100, 1000, and 10000.

1. Use `create_sample()` with $\sigma = 4.0$ and a fixed seed 1234.
2. For each sample, print the slope's standard error (`results.bse[1]`) and 95% confidence interval (`results.conf_int()[1]`).
3. How do the standard errors change, and which intervals exclude zero?

1.1.3 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 datasets, we use a simplified variant of the Ames house price dataset. The original dataset 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 datasets used in introductory machine learning classes. See [here](#) for an overview of the data, [here](#) for detailed documentation of the original variables, and [here](#) for the original data in tab-separated format.

We use the data from `ames_houses_synthetic.csv` which is stored in the `data/` folder in this repository and contains five variables. The house characteristics come from the original Ames dataset, while the variable `SalePrice` contains synthetic prices in thousands of US dollars, predicted using a hedonic regression model plus an error term. These synthetic prices are suited for teaching because we can control the data generating process. Our simplified dataset drops observations with living areas above 300 m² and lot areas above 2,500 m² since we do not want to deal with a handful of extreme outliers in this lecture. The goal is to use the variables in the dataset to predict the house price (or “target” in ML terminology).

First, we read in the CSV and inspect the columns contained in the DataFrame:

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

# Load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/TECH6-H26/main/data'
```

```
[11]: import pandas as pd

file = f'{DATA_PATH}/ames_houses_synthetic.csv'
df_ames = pd.read_csv(file)

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

```
<class 'pandas.DataFrame'>
RangeIndex: 2881 entries, 0 to 2880
Data columns (total 5 columns):
#   Column          Non-Null Count  Dtype
#   :-----
```

```

---  -----  -----
0   SalePrice      2881 non-null   float64
1   LivingArea     2881 non-null   float64
2   LotArea        2881 non-null   float64
3   YearBuilt      2881 non-null   int64
4   BuildingType   2881 non-null   str
dtypes: float64(3), int64(1), str(1)
memory usage: 112.7 KB

```

For our analysis, we extract the target (SalePrice) and scale living area:

$$A_i = \text{LivingArea}_i / 100.$$

In machine learning, it's frequently the case that the features need to be appropriately scaled to avoid numerical problems (it makes no difference for this simple example, though). In later lectures, we'll see how scaling can be implemented using functions provided by scikit-learn.

The coefficient on A measures the price change, in thousands of USD, per 100 m² of living area. We retain the original units in the plots.

```

[12]: # Create a new variable A that is the living area in hundreds of square meters
df_ames['A'] = df_ames['LivingArea'] / 100

# Define features and target variable
features = ['A']
target = 'SalePrice'

# Create separate Series / DataFrames for target and features
y = df_ames[target]
X = df_ames[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 can specify `test_size` or `train_size` to determine how much data goes into each sample. If both are omitted, the test sample contains 25% of the data. The `random_state` argument is used to seed the RNG and get reproducible results:

```

[13]: 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_train1, X_test1, 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
)

```

For the plots below, we create two DataFrames that keep the original features in their original units:

```

[14]: df_train = df_ames.loc[X_train1.index, :]
df_test = df_ames.loc[X_test1.index, :]

```

We use the training sample to explore which features are related to sale price. The `corr()` method computes pairwise correlations between numeric variables:

```

[15]: df_train[['SalePrice', 'LivingArea', 'LotArea', 'YearBuilt']].corr()['SalePrice']

```

```
[15]: SalePrice      1.000000
      LivingArea    0.784587
      LotArea       0.424788
      YearBuilt     0.653875
      Name: SalePrice, dtype: float64
```

Living area (LivingArea) has the highest correlation with sale price in the training sample. We'll start with it, then add construction year and lot area.

1.1.4 Fitting a linear model to the training sample

Once we have split the sample, we estimate **Model 1**, with an intercept and living area, on the training sample:

$$\text{SalePrice}_i = \alpha + \beta_A A_i + \epsilon_i.$$

```
[16]: # Fit model on training sample
      model1 = LinearRegression()
      model1.fit(X_train1, y_train)

      # Extract estimated parameters
      intercept = model1.intercept_
      slope = model1.coef_[0]

      print('Regression coefficients')
      print(f' Intercept: {intercept:.2f}')
      print(f' Slope per 100 m²: {slope:.2f}')
```

```
Regression coefficients
Intercept: 9.38
Slope per 100 m²: 122.15
```

The slope is the predicted price change in thousands of USD for an additional 100 m² of living area. Divide it by 100 to obtain the effect per square meter.

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

```
[17]: def plot_regression_line(X_train, y_train, X_test, y_test, intercept, slope):
      """
      Plot training and test data with fitted regression line.

      Parameters
      -----
      X_train : array-like
          Training sample living areas in m²
      y_train : Series
          Training sample target variable
      X_test : array-like
          Test sample living areas in m²
      y_test : Series
          Test sample target variable
      intercept : float
          Estimated intercept of the regression model
      slope : float
          Estimated slope of the regression model (per 100 m²)
      """
      import matplotlib.pyplot as plt

      fig, axes = plt.subplots(1, 2, figsize=(8, 3.5), layout='constrained')

      # Plot training data
      axes[0].scatter(X_train, y_train, s=15, c='none', ec='steelblue', alpha=0.8, lw=0.5)
```

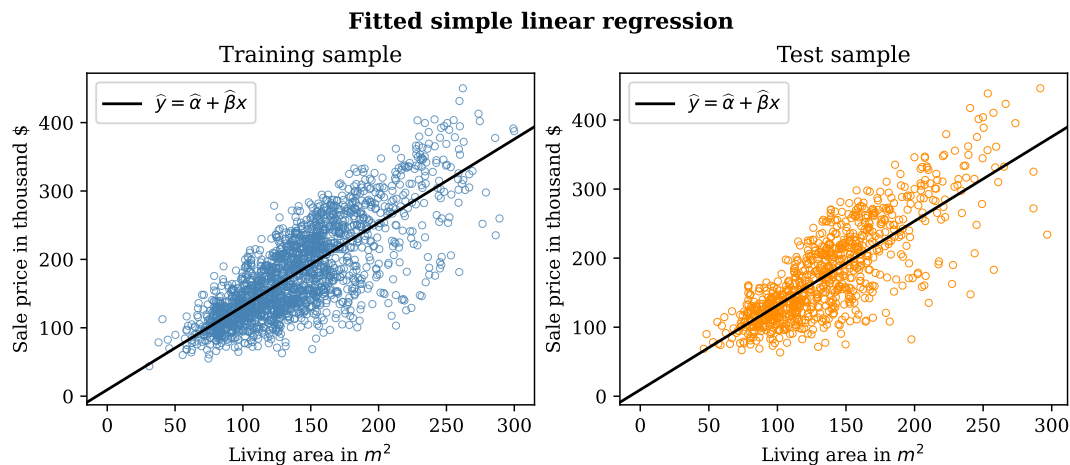
```
# Plot test data
axes[1].scatter(X_test, y_test, s=15, c='none', ec='darkorange', alpha=1.0, lw=0.5)

# Plot regression line and annotations
label = r'$\widehat{y} = \widehat{\alpha} + \widehat{\beta} x$'
titles = ['Training sample', 'Test sample']
for i, ax in enumerate(axes):
    ax.axline((0.0, intercept), slope=slope / 100, lw=1.5, c='black', label=label)

# Add annotations
ax.set_xlabel('Living area in $m^2$')
ax.set_ylabel(r'Sale price in thousand $')
ax.set_title(titles[i])
ax.legend(loc='best')

fig.suptitle('Fitted simple linear regression', fontweight='bold')
```

```
[18]: plot_regression_line(
        df_train['LivingArea'], y_train, df_test['LivingArea'], y_test, intercept, slope
    )
```



Plotting predictions and errors

We compute prediction errors using the estimated coefficients,

$$e_i = y_i - \hat{y}_i = y_i - \hat{\alpha} - \hat{\beta}_A A_i.$$

A positive error means that the model underpredicts the price; a negative error means that it overpredicts. This process can be simplified by using the `predict()` method of the fitted model which returns $\hat{y}_i$. The `predict()` method can be used for both the training and test samples.

The following code computes training predictions and errors for the plots:

```
[19]: # Compute training predictions and errors for the plots
yhat_train1 = model1.predict(X_train1)
errors_train1 = y_train - yhat_train1
```

We use small plotting helpers throughout the lecture: `plot_errors()` plots the prediction errors against an x -variable, and additionally bins the x -variable into eight equal-width intervals and plots the mean error within each bin. The bin means help us see systematic errors in the conditional mean; the spread around them describes error variance.

```
[20]: def plot_errors(x, errors, xlabel, title=None, ylabel=None, ax=None, legend=True):
        """
```

```

Plot errors and bin means.

Parameters
-----
x : array-like
    Variable shown on the x-axis.
errors : array-like
    Prediction error for each observation.
xlabel : str
    Label for the x-axis.
title : str, optional
    Title identifying the model or sample.
ylabel : str, optional
    Label for the y-axis.
ax : axes, optional
    Matplotlib axes on which to draw the plot.
legend : bool, optional
    Whether to show the legend. Default is True.
"""
# Create a new figure and axes if none are provided
if ax is None:
    _fig, ax = plt.subplots(figsize=(5, 3.5), layout='constrained')

# Plot errors for each observation
ax.scatter(x, errors, s=15, c='none', ec='steelblue', alpha=0.4, label='Errors')
# Add horizontal line at y=0 to indicate zero error
ax.axhline(0, c='black', lw=0.75, ls='--')

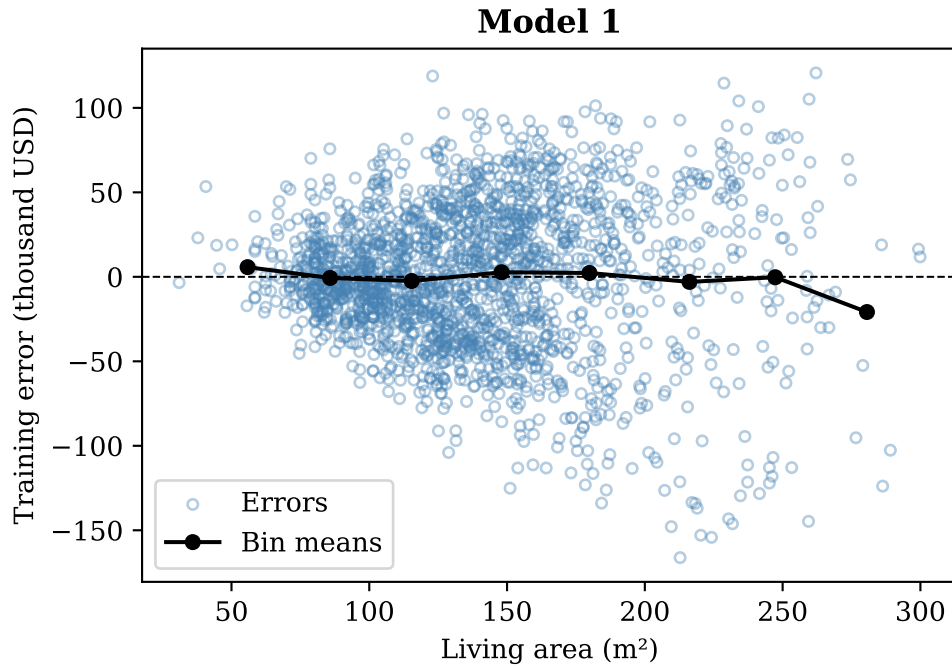
# Convert to arrays so pandas does not align different observation indices.
df_plot = pd.DataFrame({'x': np.asarray(x), 'error': np.asarray(errors)})
bins = pd.cut(df_plot['x'], bins=8)
# Compute means of x and errors within each bin
df_means = df_plot.groupby(bins, observed=True)[['x', 'error']].mean()
# Plot means of x and errors within each bin
xbin, ybin = df_means['x'], df_means['error']
ax.plot(xbin, ybin, marker='o', c='black', lw=1.5, ms=5, label='Bin means')

ax.set_xlabel(xlabel)
ax.set_ylabel(ylabel or 'Training error (thousand USD)')
if title is not None:
    ax.set_title(title, fontweight='bold')
if legend:
    ax.legend(loc='best')

```

We first plot Model 1's training errors against living area. The black line shows mean errors within bins; it does not describe the spread of errors.

```
[21]: plot_errors(df_train['LivingArea'], errors_train1, 'Living area (m²)', 'Model 1')
```



Training errors average to zero, but their bin means can still show patterns. Look at the bin means separately from the spread: a wider spread alone doesn't mean that the model is missing a pattern in average prices.

Your turn

Recreate Model 1's training error plot, but put the predicted price `yhat_train1` on the x-axis instead of living area. Explain why the plot looks almost identical to the one above.

1.1.5 Evaluating the model fit

So far, we've used training errors to spot patterns the model misses. To check predictions on new houses, we compute the following metrics on the test sample. These become particularly relevant for models with hyperparameters, i.e., settings chosen outside coefficient estimation, such as polynomial degree or penalty strength. In later lectures, we'll use these metrics on separate validation data to choose hyperparameters.

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 *coefficient of determination* or R^2 , which is a normalized version of the MSE. The R^2 is defined as

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

where $\widehat{\text{Var}}(y) = N^{-1} \sum_i (y_i - \bar{y})^2$ is the variance of the response y computed with divisor N , matching the MSE convention. 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. For an OLS model with a constant, R^2 is bounded between 0 and 1 in the sample on which the model was estimated (the training sample). On the other hand, it is possible for the R^2 to be negative in a test sample if the model performs worse than a model that predicts the sample mean for all observations.

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 compute the test MSE for Model 1 using the observed and predicted prices. Later comparisons use this same test sample.

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

# Compute test predictions for the performance metrics
yhat_test1 = model1.predict(X_test1)

# Compute Model 1 test MSE
mse = mean_squared_error(y_test, yhat_test1)
print(f'Model 1 test MSE: {mse:.1f} (thousand USD)2)')
```

Model 1 test MSE: 1720.9 (thousand USD)²

Your turn

Compute Model 1's test RMSE with `root_mean_squared_error()` and R^2 with `r2_score()`.

- Interpret both values, including the units of RMSE.
- What benchmark does $R^2 = 0$ represent in this test sample?

We'll work through six models chosen to illustrate different features. To see how each one predicts new prices, we add its test RMSE and R^2 to a comparison table. These scores are for reporting; we won't use them to choose the next model. Here's the first row for Model 1:

```
[23]: # Keep a comparison table and add one row after fitting each model.

# Compute test performance metrics for Model 1
rmse_test1 = root_mean_squared_error(y_test, yhat_test1)
r2_test1 = r2_score(y_test, yhat_test1)

df_results = pd.DataFrame(
    {'RMSE': [rmse_test1], 'R²': [r2_test1]}, index=pd.Index(['Model 1'], name='Model')
)
df_results
```

```
[23]:
```

	RMSE	R ²
Model		
Model 1	41.484218	0.644477

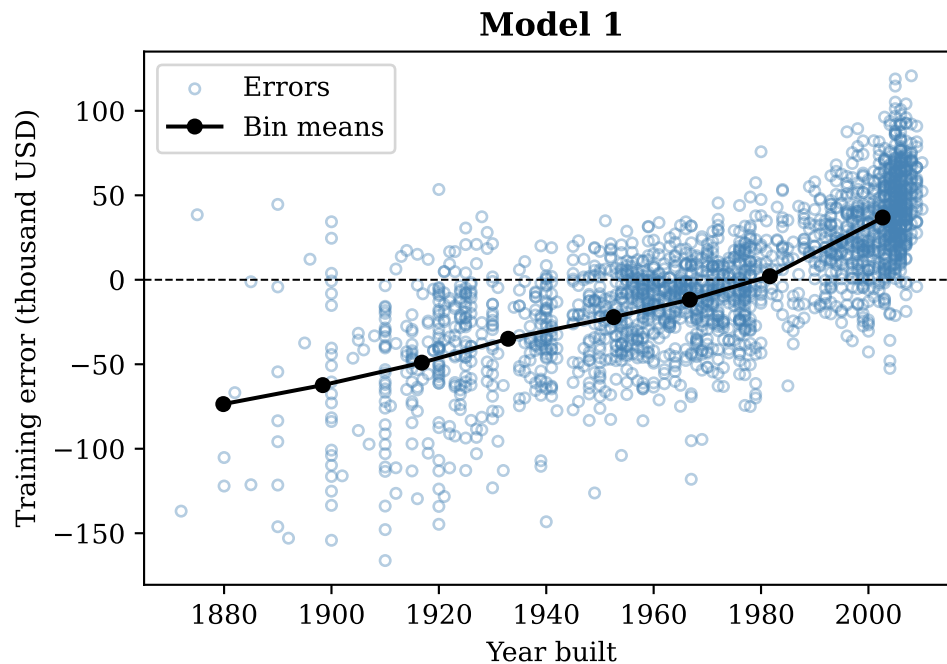
1.2 Multiple linear regression

1.2.1 Data with several explanatory variables

Multiple linear regression extends the simple model to multiple explanatory variables or regressors. To illustrate, we'll use the same Ames houses and the same split with several explanatory variables.

Model 1's training errors show a pattern against year built: houses from different construction years have different mean errors. We can see this using the `plot_errors()` function defined above:

```
[24]: plot_errors(df_train['YearBuilt'], errors_train1, 'Year built', 'Model 1')
```



Adding construction year as an additional regressor

We define **Model 2** as an extension of Model 1 with the construction year as an additional regressor:

$$\text{SalePrice}_i = \alpha + \beta_A A_i + \beta_T T_i + \epsilon_i, \quad T_i = (\text{YearBuilt}_i - 1970)/10.$$

For numerical reasons, we center the construction year around 1970 and scale it by a factor of 10. The variable T is now measured in decades since 1970. Holding living area fixed, β_T is the predicted price difference, in thousands of USD, for a house built a decade later. The coefficient on A now compares houses with the same construction year.

The column T stores the transformed construction year:

```
[25]: # T measures decades since 1970
df_ames['T'] = (df_ames['YearBuilt'] - 1970) / 10
```

We could again use `train_test_split()` to create a new training and test sample, obtaining the same subsamples as before when using the same seed. Instead, we create two new DataFrames for the training and test samples using the indices of the original training and test samples. In principle, both approaches are equivalent, but this one is guaranteed to use the same training and test samples in case we forget to specify the same seed.

```
[26]: # Define features for Model 2
features2 = ['A', 'T']
```

```
# Create new DataFrames for training and test samples with the new features
X_train2 = df_ames.loc[X_train1.index, features2]
X_test2 = df_ames.loc[X_test1.index, features2]
```

We estimate the model as before, using the same API for simple and multiple linear regression:

```
[27]: # Fit Model 2 on training sample
model2 = LinearRegression()
model2.fit(X_train2, y_train)

print(f'Model 2 intercept: {model2.intercept_:.2f}')
print('Model 2 coefficients (A, T):', model2.coef_)
```

Model 2 intercept: 34.59

Model 2 coefficients (A, T): [103.01161897 10.87476896]

The coefficient array `coef_` follows the feature order `['A', 'T']`. The estimated slope on *A* says that an additional 100 m² of living area is associated with a predicted price increase of around 103 thousand USD, holding construction year fixed. On the other hand, the estimated slope on *T* says that a house built a decade later is associated with a predicted price increase of around 10.8 thousand USD, holding living area fixed.

Next, we compare training errors against both regressors for Models 1 and 2, i.e., before and after adding the feature *T*. For this purpose, we define the function `plot_error_comparison()` which allows us to compare training errors against several explanatory variables across several models.

```
[28]: def plot_error_comparison(models, model_names, X_trains, y_train, variables, ylim=None):
    """
    Compare training errors across models and (explanatory) variables.

    Parameters
    -----
    models : list
        Fitted models or pipelines.
    model_names : list of str
        Titles for the model rows.
    X_trains : list of array-like
        Training feature matrices for the models.
    y_train : array-like
        Observed training prices in thousands of USD.
    variables : dict
        Axis labels mapped to numeric training variables.
    ylim : tuple, optional
        Common vertical error limits; autoscale when omitted.
    """
    xlabels = list(variables)
    xvalues = list(variables.values())
    nrows = len(models)
    ncols = len(xlabels)

    fig, axes = plt.subplots(
        nrows,
        ncols,
        figsize=(4 * ncols, 3 * nrows),
        sharex='col', # Share x-axis only within columns
        sharey=True, # Share y-axis across all subplots
        squeeze=False, # Keep axes two-dimensional even with one row or column
        layout='constrained',
    )

    for i, model in enumerate(models):
        yhat = model.predict(X_trains[i])
```

```

err = np.asarray(y_train) - yhat
for j in range(ncols):
    ax = axes[i, j]
    plot_errors(
        xvalues[j], err, xlabel=j, model_names[i], ax=ax, legend=False
    )

if ylim is not None:
    axes[0, 0].set_ylim(ylim)

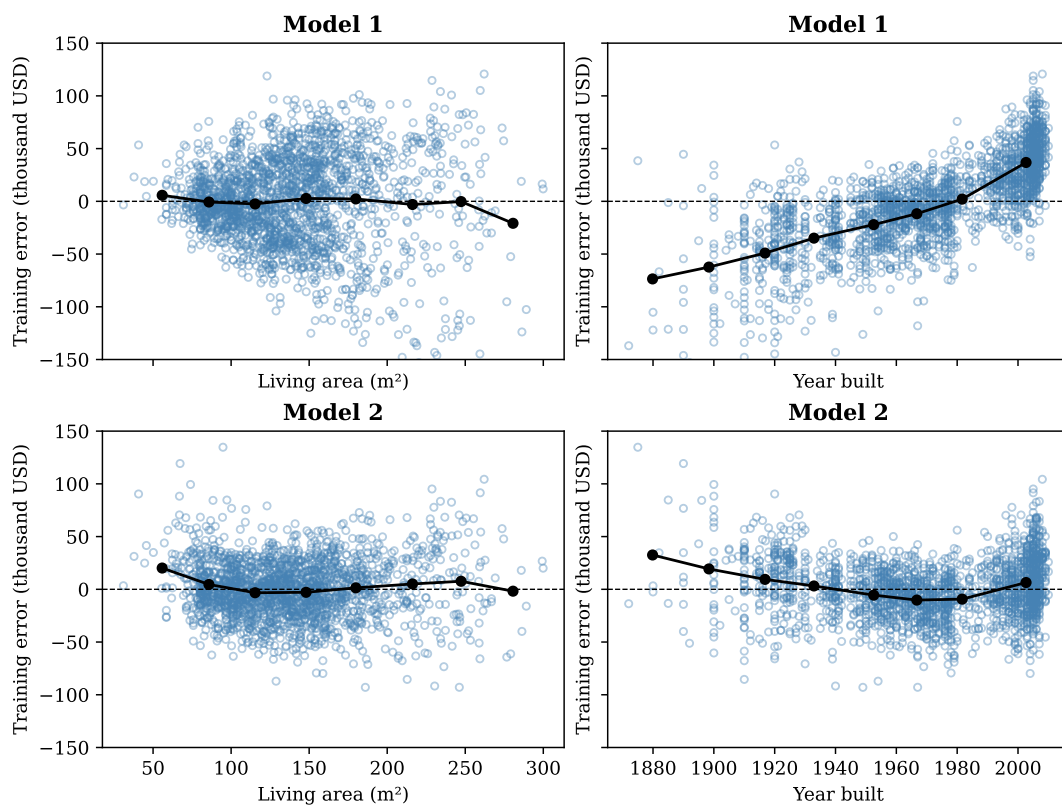
return fig, axes

```

```

[29]: fig, axes = plot_error_comparison(
    [model1, model2],
    ['Model 1', 'Model 2'],
    [X_train1, X_train2],
    y_train,
    variables={
        'Living area (m²)': df_train['LivingArea'],
        'Year built': df_train['YearBuilt'],
    },
    ylim=(-150, 150),
)

```



As the figure shows, the clearly visible upward pattern of prediction errors against construction year is no longer present in Model 2.

Finally, we compute the test predictions and evaluate Model 2's performance on the test sample:

```

[30]: # Compute model 2 predictions for test sample
yhat_test2 = model2.predict(X_test2)

# Compute metrics for Model 2
rmse_test2 = root_mean_squared_error(y_test, yhat_test2)

```

```

r2_test2 = r2_score(y_test, yhat_test2)

# Add model 2 results to the results DataFrame
df_results.loc['Model 2'] = [rmse_test2, r2_test2]
df_results

```

```

[30]:
      RMSE      R²
Model
Model 1  41.484218  0.644477
Model 2  27.896622  0.839230

```

The table confirms that adding the construction year substantially decreases the test RMSE and increases R^2 .

Adding lot area as an additional regressor

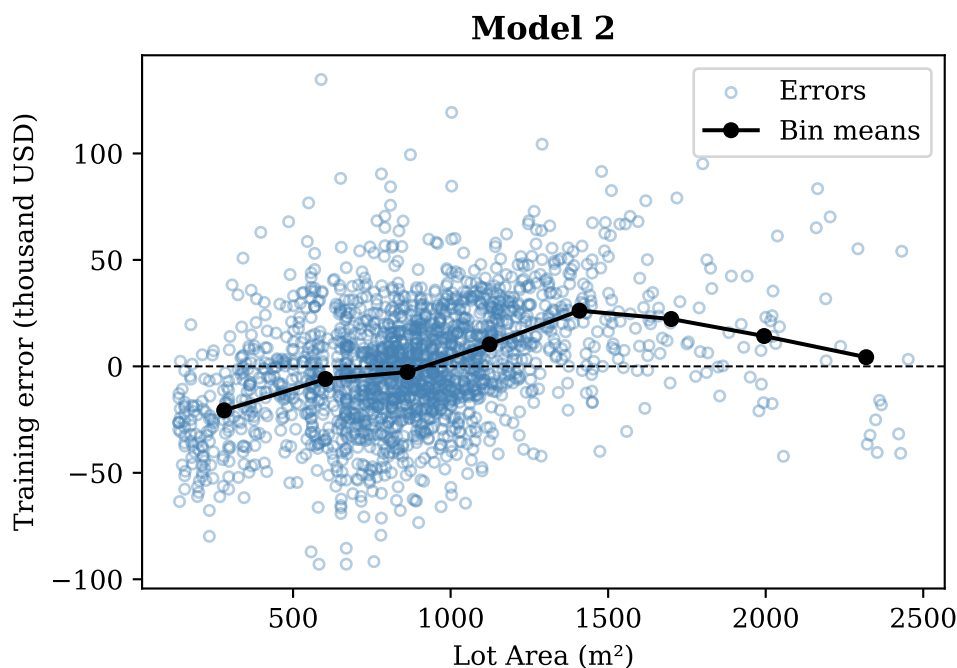
Next, we plot Model 2's training errors against lot area. The bin means rise over much of the range, suggesting that lot area could help explain the remaining errors:

```

[31]: # Compute training errors for the plots
errors_train2 = y_train - model2.predict(X_train2)

plot_errors(df_train['LotArea'], errors_train2, 'Lot Area (m²)', 'Model 2')

```



We therefore proceed to define **Model 3** which includes lot area as an additional explanatory variable:

$$\text{SalePrice}_i = \alpha + \beta_A A_i + \beta_T T_i + \beta_L L_i + \epsilon_i, \quad L_i = \text{LotArea}_i / 1000.$$

For numerical reasons, we scale the lot area by a factor of 1,000. The coefficient on L is the predicted price change, in thousands of USD, for an additional 1,000 m² of lot area, holding living area and construction year fixed.

Fitting the model on the training data is the same as in the simple case. We select the additional feature using the indices of X_{train1} and X_{test1} :

```

[32]: # Add (scaled) lot area as an additional regressor
df_ames['L'] = df_ames['LotArea'] / 1000

```

```
# Define features for Model 3
features3 = ['A', 'T', 'L']

# Create training and test DataFrames for Model 3
X_train3 = df_ames.loc[X_train1.index, features3]
X_test3 = df_ames.loc[X_test1.index, features3]

# Fit Model 3 on training sample
model3 = LinearRegression()
model3.fit(X_train3, y_train)

print(f'Model 3 intercept: {model3.intercept_:.2f}')
print('Model 3 coefficients (A, T, L):', model3.coef_)
```

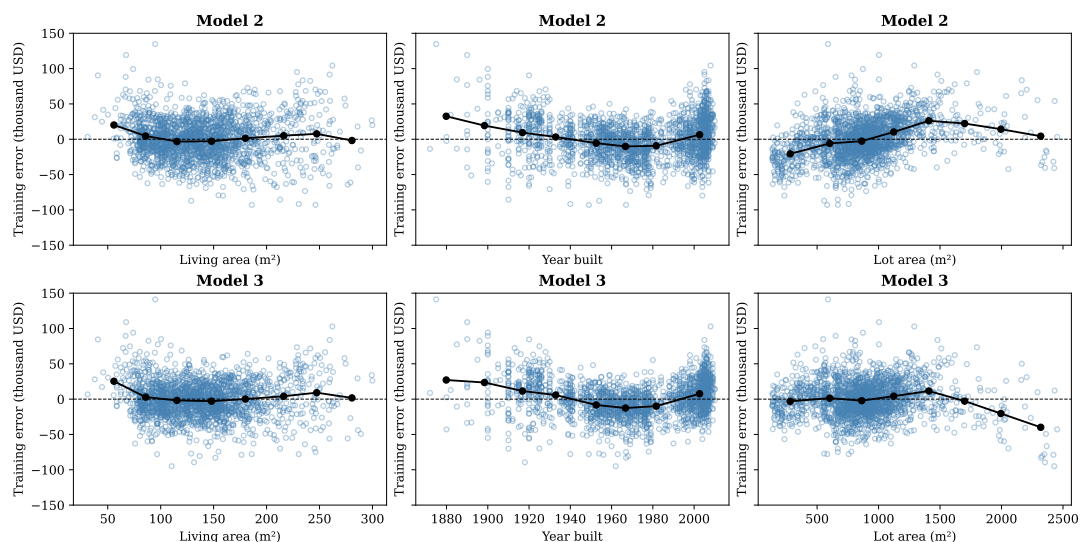
Model 3 intercept: 19.28

Model 3 coefficients (A, T, L): [92.87443441 10.9893512 33.21838988]

The coefficient array `coef_` now stores three values in the same order as the features in the feature matrix passed when fitting the model. The results say that an additional 1,000 m² of lot area is associated with a predicted price increase of about 33.2 thousand USD, holding living area and construction year fixed.

We compare training errors against lot area before and after adding *L*. The living-area and year-built panels show how the remaining patterns change against the other features.

```
[33]: fig, axes = plot_error_comparison(
    [model2, model3],
    ['Model 2', 'Model 3'],
    [X_train2, X_train3],
    y_train,
    variables={
        'Living area (m²)': df_train['LivingArea'],
        'Year built': df_train['YearBuilt'],
        'Lot area (m²)': df_train['LotArea'],
    },
    ylim=(-150, 150),
)
```



To assess the improvement of Model 3 over the previous two models on the test sample, we compute the RMSE and the R^2 :

```
[34]: # Compute test predictions for Model 3
yhat_test3 = model3.predict(X_test3)
```

```
# Compute test performance metrics for Model 3
rmse_test3 = root_mean_squared_error(y_test, yhat_test3)
r2_test3 = r2_score(y_test, yhat_test3)

# Tabulate test performance metrics for Models 1-3
df_results.loc['Model 3'] = [rmse_test3, r2_test3]
df_results
```

```
[34]:
```

	RMSE	R ²
Model		
Model 1	41.484218	0.644477
Model 2	27.896622	0.839230
Model 3	25.457015	0.866120

As you see, the addition of lot area as an explanatory variable improves the model fit, even though the improvement is not as large as when we added construction year.

Your turn

Fit a model using living area A_i and lot area L_i as regressors, keeping the same training and test samples.

- Compute its test RMSE and compare it with Models 2 and 3.
- Use `plot_error_comparison()` to compare its training errors against year built with those of Model 3.
- Does lot area compensate for leaving out construction year? Which construction periods does the reduced model tend to overpredict or underpredict?

1.2.2 Creating polynomial features

Adding construction year and lot area reduces some systematic mean errors, but these additive models require the effect of each variable to be constant. Curvature or patterns that differ across construction periods suggest trying polynomial terms and interactions.

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 some feature 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 modeled 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.

Applying this approach to the Ames housing data, we extend Model 3 with the interaction between living area A_i and construction year T_i . **Model 4** thus adds $A_i T_i$ as a feature to the regression:

$$\text{SalePrice}_i = \alpha + \beta_A A_i + \beta_T T_i + \beta_L L_i + \beta_{AT} A_i T_i + \epsilon_i.$$

Holding lot area fixed, the area slope is $\beta_A + \beta_{AT} T$ per 100 m². Thus, the price effect of additional living area can differ by construction year.

For now, we create the interaction manually and we'll study in the next section how to automate this process in scikit-learn.

```
[35]: # Interaction term: AT = A times T.
df_ames['AT'] = df_ames['A'] * df_ames['T']
```

```
# Features for Model 4: A, T, L, and AT
features4 = ['A', 'T', 'L', 'AT']

# Create training and test DataFrames for Model 4
X_train4 = df_ames.loc[X_train1.index, features4]
X_test4 = df_ames.loc[X_test1.index, features4]

# Fit Model 4 on training sample
model4 = LinearRegression()
model4.fit(X_train4, y_train)
print('Model 4 coefficients (A, T, L, AT):', model4.coef_)
```

```
Model 4 coefficients (A, T, L, AT): [93.46883063 -2.49358347 30.96397712
9.32750407]
```

We can use the estimated coefficients to evaluate the marginal effect of an increase in living area for houses built at different points in time. For example, for the years 1930, 1970, and 2010, the marginal effects can be computed as follows:

```
[36]: years = np.array([1930, 1970, 2010])
# Compute slopes (marginal effects) for given years using the
# formula: slope = beta_A + beta_AT * (YearBuilt - 1970) / 10
slopes = model4.coef_[0] + model4.coef_[3] * (years - 1970) / 10

df_slopes = pd.DataFrame(
    {'Year built': years, 'Price increase per 100 m² (thousand USD)': slopes}
)
df_slopes.round(1)
```

```
[36]:   Year built  Price increase per 100 m² (thousand USD)
0         1930                                56.2
1         1970                                93.5
2         2010                               130.8
```

The table shows the predicted price increase for an additional 100 m² of living area, holding construction year and lot area fixed. For each construction year, the slope is constant across living areas. The interaction $A \cdot T$ allows this slope to differ by construction year. For example, an additional 100 m² of living area for a house built in 2010 is associated with a predicted price increase of 130.8 thousand USD, while for a house built in 1930 the predicted price increase is only 56.2 thousand USD.

We compute test predictions and evaluate Model 4's RMSE and R^2 :

```
[37]: # Compute test predictions for Model 4
yhat_test4 = model4.predict(X_test4)

# Compute test performance metrics for Model 4
rmse_test4 = root_mean_squared_error(y_test, yhat_test4)
r2_test4 = r2_score(y_test, yhat_test4)

# Add model 4 results to the results DataFrame
df_results.loc['Model 4'] = [rmse_test4, r2_test4]
df_results
```

```
[37]:   RMSE      R²
Model
Model 1  41.484218  0.644477
Model 2  27.896622  0.839230
Model 3  25.457015  0.866120
Model 4  22.387663  0.896457
```

The table shows yet another improvement in the model fit compared to Model 3.

Your turn

Compare the already fitted Models 3 and 4 for houses built before 1950.

- Compute each model's RMSE for these houses in the test sample.
- Use `plot_error_comparison()` to plot both models' errors against living area for these houses in the training sample. Does adding the interaction reduce the trend in the bin means?

Creating polynomial features automatically

We next extend Model 4 to **Model 5**, a polynomial of degree 2 in A , L , and T (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 $\beta_0, \dots, \beta_5$. With three input variables, Model 5 contains ten terms: $1, A, L, T, A^2, AL, AT, L^2, LT, T^2$.

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 task. 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.

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

# Continuous variables used to build polynomial features
CONT_VARIABLES = ['A', 'L', 'T']

# Create training and test DataFrames for Model 5
df_train5 = df_ames.loc[X_train1.index, CONT_VARIABLES]
df_test5 = df_ames.loc[X_test1.index, CONT_VARIABLES]

# Create polynomials of degree 2 or less, including one intercept (bias).
poly = PolynomialFeatures(degree=2, include_bias=True)

# Create 10 features for Model 5: 1, A, L, T, A^2, AL, AT, L^2, LT, T^2
X_train5 = poly.fit_transform(df_train5)
```

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

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

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

The first feature is constructed as $\mathbf{X}_{(1)}^0 \cdot \mathbf{X}_{(2)}^0 \cdot \mathbf{X}_{(3)}^0$: the first row of exponents is $[0, 0, 0]$, so this is the intercept. The second row, $[1, 0, 0]$, gives the first input column A . Here $\mathbf{X}_{(i)}$ refers to the i -th column of the input matrix.

Now that we have transformed the input matrix, we can estimate the linear model on the expanded feature matrix as before, with `fit_intercept=False` so that there is exactly one constant term:

```
[40]: # Fit linear regression to polynomial features.
model5 = LinearRegression(fit_intercept=False)
model5.fit(X_train5, y_train)

# Tabulate fitted coefficients in a DataFrame
labels = poly.get_feature_names_out(CONT_VARIABLES)
pd.DataFrame({'Coef': model5.coef_, index=labels})
```

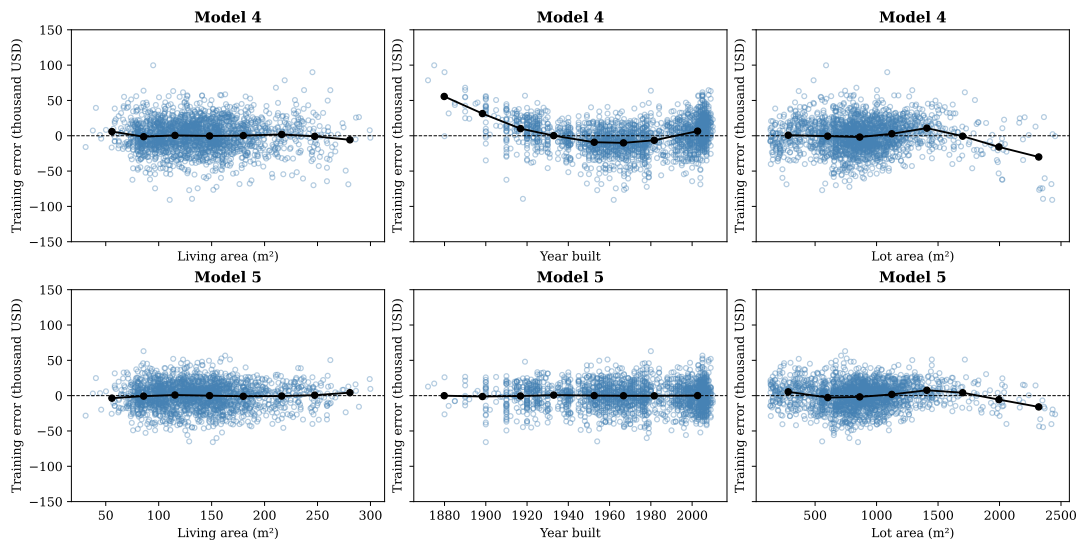
```
[40]:      Coef
1      48.846243
A      52.342747
L      11.942999
T      -0.761637
A^2    -4.626760
A L     46.300756
A T      8.040642
L^2    -20.181195
L T      2.658046
T^2      1.178710
```

The fitted model has ten coefficients, including the constant feature. The `LinearRegression` object's `intercept_` is zero because it did not add a separate intercept.

We compare Models 4 and 5 against living area, year built, and lot area, using the same training houses and error limits. Look for changes in the black bin means, especially against year built and lot area:

```
[41]: # Compute training errors for the plots
errors_train5 = y_train - model5.predict(X_train5)
```

```
[42]: fig, axes = plot_error_comparison(
    [model4, model5],
    ['Model 4', 'Model 5'],
    [X_train4, X_train5],
    y_train,
    variables={
        'Living area (m²)': df_train['LivingArea'],
        'Year built': df_train['YearBuilt'],
        'Lot area (m²)': df_train['LotArea'],
    },
    ylim=(-150, 150),
)
```



Before computing test predictions, we apply the same fitted polynomial transformation to the test features using the `transform()` method of the `poly` object stored from earlier:

```
[43]: # Apply polynomial transformation to test sample
X_test5 = poly.transform(df_test5)

# Compute predicted values for Model 5
yhat_test5 = model5.predict(X_test5)

# Compute test performance metrics for Model 5
rmse_test5 = root_mean_squared_error(y_test, yhat_test5)
r2_test5 = r2_score(y_test, yhat_test5)

# Add model 5 results to the results DataFrame
df_results.loc['Model 5'] = [rmse_test5, r2_test5]
df_results
```

```
[43]:
```

	RMSE	R^2
Model		
Model 1	41.484218	0.644477
Model 2	27.896622	0.839230
Model 3	25.457015	0.866120
Model 4	22.387663	0.896457
Model 5	17.830097	0.934324

Model 5 includes squared terms that can represent curvature as well as interactions. This is reflected in yet another substantial drop in RMSE and increase in R^2 for the test sample.

Your turn

Fit a separate polynomial model with `interaction_only=True`, keeping `degree=2` and `include_bias=True`.

- Reuse `df_train5`, `df_test5`, `y_train`, and `y_test` so that the training and test samples stay the same.
- Fit the regression with `LinearRegression(fit_intercept=False)`.
- Use `get_feature_names_out()` on the new transformer to inspect its features. Which terms have disappeared compared with Model 5?
- Compute the new model's test RMSE and compare it with Model 5. Do the squared terms improve prediction beyond the pairwise interactions?

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 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 `make_pipeline()` 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 proceed as follows:

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

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

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

In an interactive notebook, evaluating `pipe` as the last expression in a cell 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

```
[45]: # transform and fit in a single step
pipe.fit(df_train5, y_train)

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

```
Coefficients: [ 48.84624317  52.34274693  11.94299902 -0.76163673 -4.62676023
  46.30075595  8.04064183 -20.18119455  2.65804571  1.17870965]
```

We use a pipeline’s `named_steps` attribute to access the individual steps. As shown above, we can access the fitted coefficients of the linear model step with `pipe.named_steps.lr.coef_` or `pipe.named_steps['lr'].coef_`. Similarly, we can extract the polynomial feature names from the `poly` step as follows:

```
[46]: # Get feature names for polynomial features step in the pipeline
labels = pipe.named_steps.poly.get_feature_names_out()
# Tabulate fitted coefficients in a DataFrame
pd.DataFrame({'Coef': pipe.named_steps.lr.coef_, index=labels})
```

```
[46]:      Coef
1    48.846243
A    52.342747
L    11.942999
T    -0.761637
A^2  -4.626760
A L   46.300756
```

```

A T    8.040642
L^2 -20.181195
L T    2.658046
T^2    1.178710

```

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:

```

[47]: from sklearn.pipeline import make_pipeline

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

# transform and fit model in single step
pipe.fit(df_train5, y_train)

# print default names assigned to each step
pipe.named_steps

```

```

[47]: {'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).

To demonstrate that the pipeline gives the same results as the manual implementation, we compare its test predictions with Model 5's predictions using `np.allclose()`. The pipeline fits its transformation on the training data and applies the fitted transformation to test data through `predict()`.

```

[48]: # Predict test responses and compare with the manual implementation.
yhat_test_pipe = pipe.predict(df_test5)
print('Same predictions as manual Model 5:', np.allclose(yhat_test_pipe, yhat_test5))

```

Same predictions as manual Model 5: True

Your turn

Change the degree-2 pipeline to `include_bias=False` and `fit_intercept=True`.

- Fit it on the same training data and use `np.allclose()` to compare its test predictions with `yhat_test_pipe`. This function checks equality within a numerical tolerance.
- Why do both intercept conventions give the same predictions?

1.3 Dealing with categorical data

1.3.1 Creating dummy variables

So far in this unit, we only dealt with continuous data, i.e., data that can take on (almost any) real value. However, many datasets 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.

For example, consider the `BuildingType` variable, which (in this simplified version of the data) takes on three values: 'Single-family', 'Townhouse', and 'Two-family'. We use the `value_counts()` method to count the training observations in each category.

```
[49]: df_train['BuildingType'].value_counts().sort_index()
```

```
[49]: BuildingType
Single-family    1657
Townhouse       243
Two-family       116
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.

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

enc = OrdinalEncoder()

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

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

```

```
Unique values: [0. 1. 2.]
Histogram: [1657 243 116]
```

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. Building types have no meaningful numeric order. Integer codes would impose an arbitrary ordering and equal spacing between the three categories.

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:

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

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

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

# Convert string variable into binary indicator variables
building_dummies = enc.fit_transform(df_train[['BuildingType']])

```

When creating a `OneHotEncoder`, we can optionally specify the possible values using the `categories` argument. The encoder can process several input columns at once, so `categories` takes a list of array-like collections (such as lists or NumPy arrays), one per input column. The collections are matched to input columns by position: `categories[0]` applies to the first column, `categories[1]` to the second, and so on. Here, we encode only `BuildingType`, so we wrap its category list `building_uniq` in an outer list: `categories=[building_uniq]`.

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 datasets, there is no need to use sparse matrices.

We inspect the first five rows and sum the ones in each column to verify that category frequencies are preserved. The original category labels are stored in the `categories_` attribute:

```
[52]: # Display the first 5 rows
building_dummies[:5]
```

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

```
[53]: # Print category frequencies
       building_dummies.sum(axis=0)
```

```
[53]: array([1657., 243., 116.]])
```

```
[54]: # Print category labels
       print(enc.categories_[0])
```

```
['Single-family' 'Townhouse' 'Two-family']
```

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.

```
[55]: # Create dummy variables for `BuildingType` using pandas
       pd.get_dummies(df_train['BuildingType']).head(5)
```

```
[55]:
```

	Single-family	Townhouse	Two-family
2674	False	False	True
63	True	False	False
2169	True	False	False
308	True	False	False
2670	True	False	False

1.3.2 Adding building-type dummies to the polynomial model

Model 6 adds building-type dummies to Model 5's quadratic terms. We use single-family houses as the reference: townhouse and two-family coefficients are price differences relative to single-family houses, holding A, L, T fixed. The dummies remain outside the polynomial transformation. Its constant feature supplies the only intercept, so we set `fit_intercept=False`.

The `drop` argument specifies one category to omit per input column, in the same column order as categories. Here, `drop=['Single-family']` is a flat list with one entry because we encode one input column.

We reuse Model 5's polynomial features and add the building-type dummies:

```
[56]: # Drop one category because the polynomial already contains an intercept.
       enc6 = OneHotEncoder(
           categories=[building_uniq], drop=['Single-family'], sparse_output=False
       )

       # Building-type columns on training and test samples
       building_type_train = df_ames.loc[X_train1.index, ['BuildingType']]
       building_type_test = df_ames.loc[X_test1.index, ['BuildingType']]

       # Fit-transform training BuildingType into dummies_train
       dummies_train = enc6.fit_transform(building_type_train)
       # Transform test BuildingType into dummies_test
       dummies_test = enc6.transform(building_type_test)

       # Reuse Model 5's polynomial features and add building-type dummies.
       X_train6 = np.hstack((X_train5, dummies_train))
       X_test6 = np.hstack((X_test5, dummies_test))
```

We fit the model on the training data and tabulate the estimated coefficients:

```
[57]: # Fit Model 6 on training sample (intercept is included in polynomial features)
model6 = LinearRegression(fit_intercept=False)
model6.fit(X_train6, y_train)

# Get coefficient labels for polynomial features
labels6 = list(poly.get_feature_names_out())
# Get coefficient labels for building-type dummies
labels6 += list(enc6.get_feature_names_out())
pd.DataFrame({'Coef': model6.coef_}, index=labels6)
```

```
[57]:
```

	Coef
1	11.162224
A	54.416710
L	73.174060
T	-3.857738
A ²	-3.627468
A L	42.865109
A T	7.725070
L ²	-40.376615
L T	4.986909
T ²	1.093566
BuildingType_Townhouse	33.067420
BuildingType_Two-family	-29.451434

Note that changing the reference category changes coefficient interpretation while preserving the fitted model.

We compare Model 5 and Model 6 training errors by building type, with group means before and after adding dummies using the following function:

```
[58]: def plot_errors_by_building_type(errors_by_model, model_names, df_train):
    """
    Compare training errors across models and building types.

    Parameters
    -----
    errors_by_model : list of array-like
        Training errors for each model.
    model_names : list of str
        Titles for the model columns.
    df_train : DataFrame
        Training dataset.
    """
    ncols = len(errors_by_model)
    building_uniq = list(np.sort(df_train['BuildingType'].unique()))
    ngroups = len(building_uniq)

    _fig, axes = plt.subplots(
        1, ncols, figsize=(8, 3), sharey=True, layout='constrained'
    )
    for i, ax in enumerate(axes):
        means = []
        # Loop over building types and plot errors for each type
        for j in range(ngroups):
            keep = df_train['BuildingType'] == building_uniq[j]
            err = errors_by_model[i][keep]
            x = np.full(len(err), j)
            label = 'Errors' if j != 0 else None
            ax.scatter(x, err, s=15, c='none', ec='steelblue', alpha=0.8, label=label)
            # Compute and store mean error for each building type
            means.append(err.mean())

        ax.plot(range(ngroups), means, 'o-', c='black', lw=1.5, label='Mean error')
```

```

ax.axhline(0, c='black', lw=0.75, ls='--', zorder=-1)
ax.set_xticks(range(len(building_uniq)), building_uniq)
ax.set_title(model_names[i])
ax.set_xlim(-0.5, ngroups - 0.5)

axes[0].legend()
axes[0].set_ylabel('Training error (thousand USD)')

```

Model 5 tends to underpredict townhouses and overpredict two-family houses. After adding the dummies, each building type has a mean training error of zero.

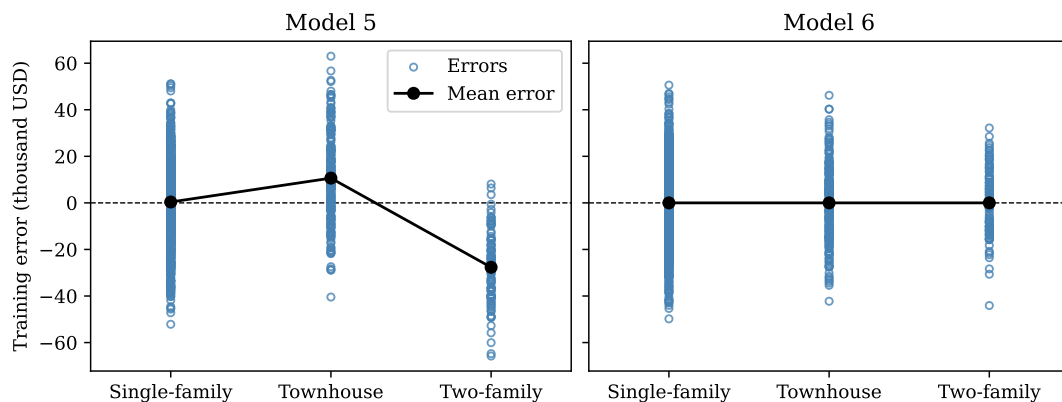
```

[59]: # Compute training errors for the plots
errors_train6 = y_train - model6.predict(X_train6)

# Compare training errors across models and building types.
errors_by_model = [errors_train5, errors_train6]
model_names = ['Model 5', 'Model 6']

plot_errors_by_building_type(errors_by_model, model_names, df_train)

```



We compare Model 6's test RMSE and R^2 with Model 5 to assess the effect of adding the dummies. The comparison table now contains all six models:

```

[60]: # Compute test predictions for Model 6
yhat_test6 = model6.predict(X_test6)

# Compute test performance metrics for Model 6
rmse_test6 = root_mean_squared_error(y_test, yhat_test6)
r2_test6 = r2_score(y_test, yhat_test6)

# Add model 6 results to the results DataFrame
df_results.loc['Model 6'] = [rmse_test6, r2_test6]
df_results

```

```

[60]:
      RMSE      R²
Model
Model 1  41.484218  0.644477
Model 2  27.896622  0.839230
Model 3  25.457015  0.866120
Model 4  22.387663  0.896457
Model 5  17.830097  0.934324
Model 6  15.254890  0.951925

```

We close this section by plotting observed versus predicted prices for Model 6.

```

[61]: def plot_predictions(ax, y, yhat, title=None):
    """
    Plot observed and predicted prices on the supplied axes.
    """

```

```

Parameters
-----
ax : axes
    Matplotlib axes on which to draw the plot.
y : array-like
    Observed prices in thousands of USD.
yhat : array-like
    Predicted prices in thousands of USD.
title : str, optional
    Title for the plot.
"""
ax.scatter(yhat, y, s=15, c='none', ec='darkorange', alpha=0.6)
ax.axline((0, 0), slope=1, c='black', lw=0.75, ls='--')

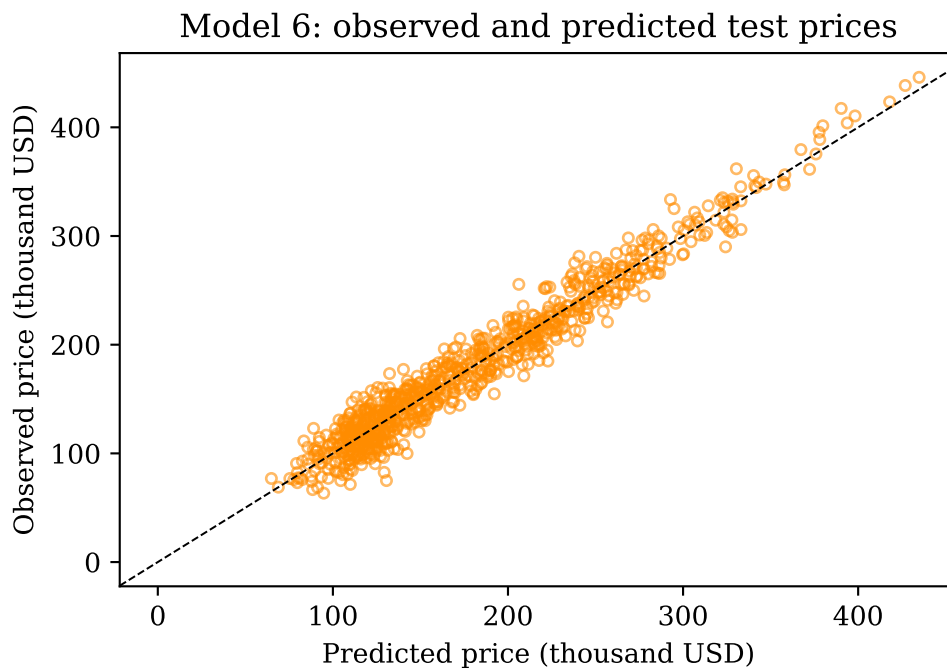
xlabel = 'Predicted price (thousand USD)'
ylabel = 'Observed price (thousand USD)'
ax.set_xlabel(xlabel)
ax.set_ylabel(ylabel)
if title is not None:
    ax.set_title(title)

```

```

[62]: fig, ax = plt.subplots(figsize=(5, 3.5), layout='constrained')
      title = 'Model 6: observed and predicted test prices'
      plot_predictions(ax, y_test, yhat_test6, title=title)

```



The 45-degree line represents perfect predictions. Points above it are houses whose prices the model underpredicts; points below it are houses whose prices the model overpredicts. This gives an overall view of Model 6's test predictions, while the error plots show which house characteristics are associated with systematic prediction errors.

Your turn

Change `enc6` to use `townhouse` as the reference category and refit Model 6 on the same data. Keep the dummies outside the polynomial and include only one intercept.

1. Use `np.allclose()` to compare the new test predictions with `yhat_test6`.

2. How does the new single-family coefficient relate to the old townhouse coefficient?

1.4 List of functions used in this lecture

The following list contains the functions, classes, and methods covered in this lecture. Official API links give details on arguments and return values; the [\[go to section\]](#) links lead to their use here. Least-squares solvers below are discussed alternatives, not executed examples.

1.4.1 NumPy functions and methods

- `default_rng()` Create a random number generator. [\[go to section\]](#)
 - `Generator.uniform()` Draw samples from a uniform distribution. [\[go to section\]](#)
 - `Generator.normal()` Draw samples from a normal distribution. [\[go to section\]](#)
- `unique()` Find unique array elements. [\[go to section\]](#)
- `bincount()` Count occurrences of nonnegative integers. [\[go to section\]](#)
- `hstack()` Stack arrays horizontally. [\[go to section\]](#)
- `full()` Create an array filled with a specified value. [\[go to section\]](#)
- `allclose()` Check whether arrays are equal within a tolerance. [\[go to section\]](#)
- `lstsq()` Compute a least-squares solution to a linear system. [\[go to section\]](#)

1.4.2 SciPy functions

- `lstsq()` Compute a least-squares solution to a linear system. [\[go to section\]](#)

1.4.3 pandas functions and methods

- `DataFrame.corr()` Compute pairwise column correlations. [\[go to section\]](#)
- `cut()` Bin values into discrete intervals. [\[go to section\]](#)
- `Series.value_counts()` Count occurrences of unique values. [\[go to section\]](#)
- `get_dummies()` Convert categorical values into dummy variables. [\[go to section\]](#)

1.4.4 scikit-learn classes and functions

Regression models (`sklearn.linear_model`)

- `LinearRegression` Fit ordinary least squares linear regression. [\[go to section\]](#)
 - `fit()` Fit a linear regression model. [\[go to section\]](#)
 - `predict()` Predict response values. [\[go to section\]](#)

Splitting data (`sklearn.model_selection`)

- `train_test_split()` Split arrays into random training and test subsets. [\[go to section\]](#)

Model evaluation (`sklearn.metrics`)

- `mean_squared_error()` Compute the mean squared prediction error. [\[go to section\]](#)
- `root_mean_squared_error()` Compute the root mean squared prediction error. [\[go to section\]](#)
- `r2_score()` Compute the coefficient of determination (R^2). [\[go to section\]](#)

Feature preprocessing (`sklearn.preprocessing`)

- `PolynomialFeatures` Generate polynomial and interaction features. [\[go to section\]](#)
 - `fit_transform()` Fit the transformation and transform the data. [\[go to section\]](#)
 - `transform()` Apply the fitted transformation to data. [\[go to section\]](#)
 - `get_feature_names_out()` Return names of the transformed features. [\[go to section\]](#)
- `OrdinalEncoder` Encode categorical features as integers. [\[go to section\]](#)
 - `fit_transform()` Fit the transformation and transform the data. [\[go to section\]](#)
- `OneHotEncoder` Encode categorical features as binary indicators. [\[go to section\]](#)
 - `fit_transform()` Fit the transformation and transform the data. [\[go to section\]](#)
 - `transform()` Apply the fitted transformation to data. [\[go to section\]](#)
 - `get_feature_names_out()` Return names of the transformed features. [\[go to section\]](#)

Pipelines (`sklearn.pipeline`)

- `Pipeline` Chain transformers and a final estimator. [\[go to section\]](#)
 - `fit()` Fit the transformers and final estimator. [\[go to section\]](#)
 - `predict()` Transform data and predict response values. [\[go to section\]](#)
- `make_pipeline()` Build a pipeline with automatically assigned step names. [\[go to section\]](#)

1.4.5 statsmodels classes and methods

- `OLS` Specify an ordinary least squares regression model. [\[go to section\]](#)
 - `fit()` Fit the model and return regression results. [\[go to section\]](#)
- `add_constant()` Add a column of ones to an array. [\[go to section\]](#)
- `summary()` Summarize regression results. [\[go to section\]](#)
- `conf_int()` Compute confidence intervals for model parameters. [\[go to section\]](#)

2 Cross-validation and model selection

2.1 Motivation

In Lecture 1, we fitted regression models and evaluated their predictions on a single train-test split. This was done in an ad hoc manner and didn't reflect the best practices in machine learning for two reasons. First, the estimated test error depends on this single split, which can be problematic, especially in small samples. Second, repeatedly choosing model complexity using that test set weakens its role as an independent benchmark.

Cross-validation (CV) is a more systematic approach to choosing model complexity and hyperparameters that is designed to address these issues. In K -fold cross-validation, we divide the original training data into K folds. We repeatedly fit the model on $K - 1$ folds and evaluate its performance on the remaining validation fold. Model selection is then based on the average performance across all K folds, alleviating the dependence on one single validation set. We use the test set for final evaluation only after model selection is complete.

Cross-validation thus helps select model complexity by evaluating predictions on held-out observations, reducing the risk of overfitting. We saw in Workshop 1 that models with higher-degree polynomials can fit the training data more closely, but they may not generalize well to new observations and can lead to larger prediction errors.

In this lecture, we apply cross-validation to two settings: selecting the degree of a polynomial regression model and selecting the number of principal components in principal component regression.

Additional details about cross-validation are available in the scikit-learn [cross-validation guide](#).

2.2 Optimizing hyperparameters with cross-validation

2.2.1 Modeling workflow

A *hyperparameter*, such as polynomial degree, governs model complexity or the estimation procedure. Model *coefficients* are instead estimated from the data once the hyperparameters are fixed.

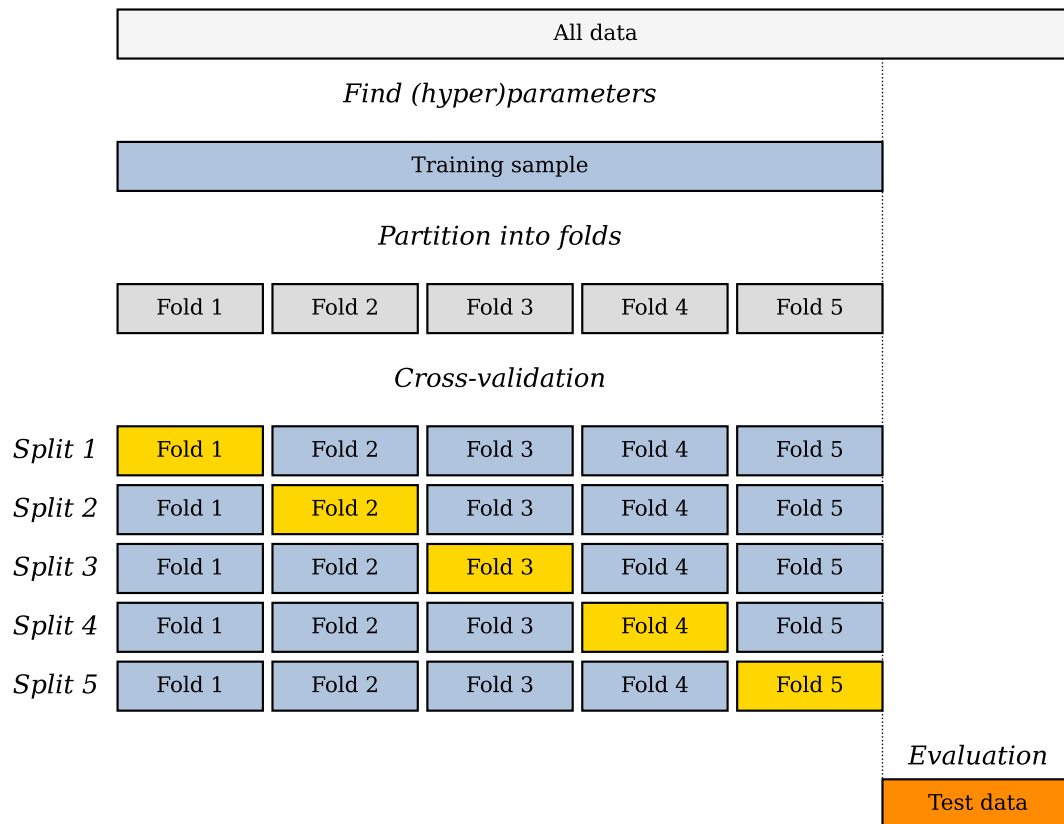
We distinguish three roles for observations: *training* observations estimate the model, *validation* observations guide hyperparameter selection, and an independent *test* sample evaluates the selected prediction rule.

With limited data, we can use K -fold cross-validation within the training sample. This algorithm can be summarized as follows:

1. Set aside a test sample for the final evaluation.
2. Divide the remaining observations into K folds. For each fold, fit the model on the other $K - 1$ folds and score its predictions on the held-out validation fold. Average the K validation scores to compare hyperparameters.
3. Select the hyperparameters and refit on all observations in the training set.
4. Evaluate the refitted model once on the independent test sample.

2 Cross-validation and model selection

The following figure illustrates this workflow for $K = 5$ folds. The yellow boxes represent the validation folds used in each iteration of cross-validation, while the blue boxes represent the training folds.



2.2.2 Ames data and cross-validation setup

We use the Ames housing data from `ames_houses.csv`. Unlike the synthetic data in Lecture 1, this dataset uses the actual sale prices and is therefore more dispersed and harder to model than in Lecture 1. As before, `SalePrice` is measured in thousands of USD, and our main evaluation metric is root mean squared error (RMSE) measured in the same units.

```
[1]: %matplotlib inline

# Imports used in the notebook
import matplotlib.pyplot as plt
import numpy as np

# Default plot fonts
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.size'] = 10
```

Data preparation

We define the following function to load the data and apply the sample restrictions that we'll use throughout this lecture:

```
[2]: def load_ames_data(file, features, target):
    """
    Load filtered Ames observations for selected features and a target.
```

```

Parameters
-----
file : str
    Path to the Ames CSV file
features : list of str
    Column names used as predictors
target : str
    Column name used as the response

Returns
-----
X : DataFrame
    Selected predictors
y : Series
    Selected response
"""
import pandas as pd

df_ames = pd.read_csv(file)
df_ames = df_ames.query('LivingArea <= 350 & LotArea <= 5000')
df_ames = df_ames.dropna(subset=[*features, target])
return df_ames[features], df_ames[target]

```

We start with `LivingArea` and `YearBuilt` as predictors. We retain houses with a living area of at most 350 m² and a lot area of at most 5,000 m², and drop observations with missing values in the selected predictors or target.

After loading, we use `train_test_split()` to reserve 30% of the observations for the final test. A fixed random seed makes the split reproducible. All cross-validation and model fitting use the remaining 70%.

```

[3]: from sklearn.model_selection import train_test_split

# Relative path to the data folder in the git repository
DATA_PATH = '../data'

# Path to Ames housing data
file = f'{DATA_PATH}/ames_houses.csv'

# Features and target variables
features = ['LivingArea', 'YearBuilt']
target = 'SalePrice'

# Load the Ames housing data
X_ames, y_ames = load_ames_data(file, features, target)

# Set aside 30% of the observations for the independent test sample
X_train, X_test, y_train, y_test = train_test_split(
    X_ames, y_ames, test_size=0.3, random_state=1234
)

print(f'Number of training observations: {len(y_train):,d}')
print(f'Number of test observations: {len(y_test):,d}')

```

```

Number of training observations: 2,041
Number of test observations: 876

```

Cross-validation setup

The next few sections explore various ways to run cross-validation and plot validation curves within the training sample. Before doing so, we set up a common splitter using the `KFold` class.

The argument `KFold(n_splits=K)` controls the number of folds. By default, `KFold` splits the data deterministically into contiguous folds of approximately equal size, with about N/K observations in each fold.

Whether to shuffle the data before splitting depends on the modeling task. If the observations are already in random order, shuffling is unnecessary. If we explicitly model the time series dimension, we need to respect chronological order and thus avoid shuffling. In our analysis of the Ames housing data, however, we ignore the year a house was sold and instead treat the data as a pooled cross-section, where observation order is irrelevant.

The CSV is ordered by sale year, but the initial train-test split shuffles the observations, so this ordering is not an issue here. We nevertheless use `shuffle=True` in `KFold` to make the random assignment to CV folds explicit. A fixed random seed makes these folds reproducible.

The following code creates a `KFold` splitter with 5 folds, shuffling the data and using a fixed random seed:

```
[4]: from sklearn.model_selection import KFold

# Set up a KFold cross-validation splitter
cv_splitter = KFold(n_splits=5, shuffle=True, random_state=1234)
```

Finally, we choose a scoring rule to evaluate predictions. We use the *negative* root mean squared error (RMSE) because scikit-learn's cross-validation functions follow a "larger is better" convention. Maximizing negative RMSE is equivalent to minimizing RMSE. See the official [scoring documentation](#) for supported scorer names.

```
[5]: # Set up a scoring rule for cross-validation
scoring = 'neg_root_mean_squared_error'
```

2.2.3 Evaluating a quadratic model

We start by evaluating a quadratic model that uses living area and construction year to predict prices. Later, we will use cross-validation to choose the optimal polynomial degree.

In Lecture 1, we scaled the living area by 100 and centered the construction year around 1970 and expressed it in decades. While we could continue to use this manual approach, it is more convenient to use `StandardScaler`, which computes means and standard deviations from the training observations, centering and scaling both inputs before we take powers and interactions.

Before we run the cross-validation, we build one named `Pipeline` with four steps:

1. `input_scale` centers and scales the original inputs.
2. `poly` uses `PolynomialFeatures` to create powers and interactions, initially of degree 2.
3. `scale` standardizes the expanded columns.
4. `lr` fits `LinearRegression` with an intercept.

Centering the inputs before creating polynomial features can reduce near-collinearity between their powers, especially for inputs such as construction year. Omitting the initial input scaling can change fitted predictions through numerical error. Moreover, powers and interactions can have different variances, so the second scaler standardizes the expanded columns. For OLS, the latter standardization is usually not required, but it can be useful for numerical stability. For methods like principal component regression, introduced below, and for Ridge and Lasso, introduced in the next lecture, this second standardization affects the fitted model because these methods are sensitive to the scale of the features.

We need to construct the `PolynomialFeatures` object with `include_bias=False` to avoid a redundant constant column. Additionally, there is no need to standardize the intercept, and attempting to do so with `StandardScaler` produces a column of zeros. The correct approach therefore is to let `LinearRegression` handle the intercept and only standardize the other columns.

In each CV split, we fit the entire pipeline on the training observations. The scalers compute their means and standard deviations from these observations and reuse them to transform the validation observations. Scaling the full dataset before cross-validation would let validation observations influence model fitting. This is *data leakage* and can make validation scores overly optimistic.

Reusing the training statistics also keeps the meaning of each predictor consistent. For example, a standardized living area of zero always represents the training mean. Computing a separate mean for the validation sample would change that meaning. After model selection, we refit the pipeline on all training observations and apply it unchanged to the test sample. Scikit-learn's [common pitfalls guide](#) explains both data leakage and inconsistent preprocessing.

The following code constructs the pipeline:

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

# Build a pipeline for a quadratic model with input standardization
pipe = Pipeline(
    steps=[
        ('input_scale', StandardScaler()),
        ('poly', PolynomialFeatures(degree=2, include_bias=False)),
        ('scale', StandardScaler()),
        ('lr', LinearRegression(fit_intercept=True)),
    ]
)
```

To evaluate our quadratic pipeline with cross-validation, we use `cross_val_score()`. It fits a fresh copy of the entire pipeline on each fold's training observations and scores its predictions on the held-out validation observations. It returns an array containing one validation score per fold.

The following Python pseudocode illustrates what `cross_val_score()` does under the hood:

```
# Set up an empty list to store the validation scores
scores = []

# Loop over the K folds
for k in range(K):
    # Split the data into training and validation folds
    X_train, X_validate = X[train_index], X[validate_index]
    y_train, y_validate = y[train_index], y[validate_index]

    # Initialize a fresh copy of the estimator for this fold
    estimator = clone(estimator)

    # Fit the estimator on the training fold
    estimator.fit(X_train, y_train)

    # Score the estimator on the validation fold
    score = scoring(estimator, X_validate, y_validate)

    # Store the score for this fold
    scores.append(score)
```

Instead of writing this loop ourselves, we use `cross_val_score()`. Its main arguments are:

- `estimator`: the model or pipeline to evaluate, here our quadratic pipeline pipe.
- `X` and `y`: the training predictors and target, here `X_train` and `y_train`.
- `cv`: a CV splitter or number of folds; here we use the shared splitter `cv_splitter`.
- `scoring`: the scoring rule, here `scoring='neg_root_mean_squared_error'`.

We pass the pipeline directly without fitting it first. Each CV fit estimates the scaling parameters and regression coefficients using only that fold's training observations.

We set `n_jobs=4` to fit and score models in up to four parallel jobs. More jobs generally speed up cross-validation, but they also use more memory because several models are fitted at the same time.

```
[7]: import pandas as pd
      from sklearn.model_selection import cross_val_score

      # Evaluate the quadratic model with cross-validation
      quadratic_scores = cross_val_score(
          pipe, # estimator or pipeline
          X_train, # training predictors
          y_train, # training target
          cv=cv_splitter, # cross-validation splitter
          scoring=scoring, # scoring rule
          n_jobs=4, # number of parallel jobs
      )

      # Print the quadratic scores for each of the 5 folds
      pd.DataFrame(
          {'Score': quadratic_scores},
          index=pd.Index(np.arange(len(quadratic_scores)) + 1, name='Fold'),
      )
```

```
[7]:      Score
Fold
1    -40.650216
2    -43.669891
3    -42.719426
4    -43.025824
5    -38.443521
```

The returned scores are *negative* RMSEs. We average the scores across the five folds and negate the result to obtain the quadratic model's mean validation RMSE. All RMSEs below are in thousands of USD.

```
[8]: quadratic_rmse = -quadratic_scores.mean()
      print(f'Quadratic model CV RMSE: {quadratic_rmse:.3f}')
```

Quadratic model CV RMSE: 41.702

The quadratic model's mean validation RMSE is about 41.7 thousand USD. This averages prediction errors over five held-out folds within the training sample. The independent test sample remains reserved for the final evaluation.

Comparing training and validation errors

`cross_val_score()` returns only the validation scores, as the name implies. If we want to recover additional information from the cross-validation such as the training scores, we can use `cross_validate()`. It takes the same main arguments, and we set `return_train_score=True` to obtain the training scores as well. These can be useful to assess overfitting on the training data.

The function returns a dictionary with the keys `train_score`, `test_score`, `fit_time`, and `score_time`. Each value is an array containing one value per fold. We now evaluate the quadratic model again to compare its mean training and validation errors.

The terminology used by scikit-learn can be confusing here. The values returned under the key `test_score` are *validation* scores from the held-out CV folds, not scores from an independent test sample.

```
[9]: from sklearn.model_selection import cross_validate
```

```
# Run cross-validation to obtain both training and validation scores for the
# quadratic model
scores_quadratic = cross_validate(
    pipe, # estimator or pipeline
    X_train, # training predictors
    y_train, # training target
    cv=cv_splitter, # cross-validation splitter
    scoring=scoring, # scoring rule
    n_jobs=4, # number of parallel jobs
    return_train_score=True, # include training scores
)
```

To compute the mean training and validation RMSEs, we average the negative scores across folds and negate the result:

```
[10]: quadratic_train_rmse = -scores_quadratic['train_score'].mean()
quadratic_validation_rmse = -scores_quadratic['test_score'].mean()

print(f'Mean training RMSE: {quadratic_train_rmse:.3f}')
print(f'Mean validation RMSE: {quadratic_validation_rmse:.3f}')
```

```
Mean training RMSE: 41.387
Mean validation RMSE: 41.702
```

Training RMSE measures fit to observations used to estimate the model. Validation RMSE measures predictions on observations excluded from that fit. A large gap can signal overfitting; the two errors need not coincide even when model complexity is fixed.

Your turn

1. Repeat the quadratic-model cross-validation using only `LivingArea`. Keep the same training observations, folds, and scoring rule.
2. Compare the mean training and validation RMSEs with the two-predictor model. How much does including `YearBuilt` improve prediction?

2.2.4 Choosing the polynomial degree

So far, we have evaluated one fixed quadratic model. We now want to compare degrees 1 through 9 using the convenience function `validation_curve()`. It handles the candidate values and folds without the caller having to explicitly loop over the polynomial degree.

The following pseudocode illustrates what `validation_curve()` does under the hood:

```
# Set up empty lists to store the training and validation scores
train_scores = []
validation_scores = []
# Loop over the candidate degrees
for degree in degrees:
    # Set the polynomial degree in the pipeline
    pipe.set_params(poly__degree=degree)
    # Run cross-validation to obtain training and validation scores
    scores = cross_validate(
        pipe,
        X_train,
        y_train,
        cv=cv_splitter,
        scoring=scoring,
        return_train_score=True
```

```
)
# Store the training and validation scores for this degree
train_scores.append(scores['train_score'])
validation_scores.append(scores['test_score'])
```

Instead of writing this loop ourselves, we use `validation_curve()`.

The only aspect that requires special attention is the parameter name that needs to be specified so that `validation_curve()` knows which parameter to vary. `validation_curve()` accepts estimator objects such as `LinearRegression` or pipelines, and the parameter name we want to cross-validate must be specified depending on this estimator type:

- For a single estimator, the parameter name is simply the name of the corresponding attribute.
- For a pipeline, the parameter name is constructed by combining the pipeline step name and the parameter name, separated by two underscores, `step__parameter`. For example, to vary the polynomial degree in our pipeline, we use `poly__degree` as the parameter name since we assigned the name `poly` to the `PolynomialFeatures` step in our pipeline.

The call to `validation_curve()` therefore looks like this:

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

# Sequence of polynomial degrees to evaluate
degrees = np.arange(1, 10)

# Compute training and validation scores for polynomial degrees
train_scores, validation_scores = validation_curve(
    pipe, # estimator or pipeline
    X_train, # predictors
    y_train, # target
    param_name='poly__degree', # parameter name to loop over
    param_range=degrees, # candidate values for the parameter
    cv=cv_splitter, # cross-validation splitter
    scoring=scoring, # scoring rule
    n_jobs=4, # number of parallel jobs
)
```

Both returned score arrays have shape `(len(degrees), K)`: rows correspond to degrees and columns to folds. For example, `train_scores[0, 0]` returns the training score for the first degree and the first fold, while `validation_scores[0, 0]` returns the corresponding validation score.

We can average the scores across folds to obtain mean training and validation RMSEs for each degree:

```
[12]: # Compute average RMSE across folds for each degree
rmse_train = -train_scores.mean(axis=1)
rmse_validation = -validation_scores.mean(axis=1)

# Find the degree with the minimum validation RMSE
imin = np.argmin(rmse_validation)
dmin = degrees[imin]
print(f'Degree with minimum validation RMSE: {dmin}')
print(f'CV selection RMSE: {rmse_validation[imin]:.3f}')
```

```
Degree with minimum validation RMSE: 5
CV selection RMSE: 39.087
```

The following function plots the validation curve for both the training and validation RMSEs against the polynomial degree.

```
[13]: def plot_validation_curve(degrees, rmse_train, rmse_validation, dmin):
    """
    Plot mean training and validation RMSEs against polynomial degree.
```

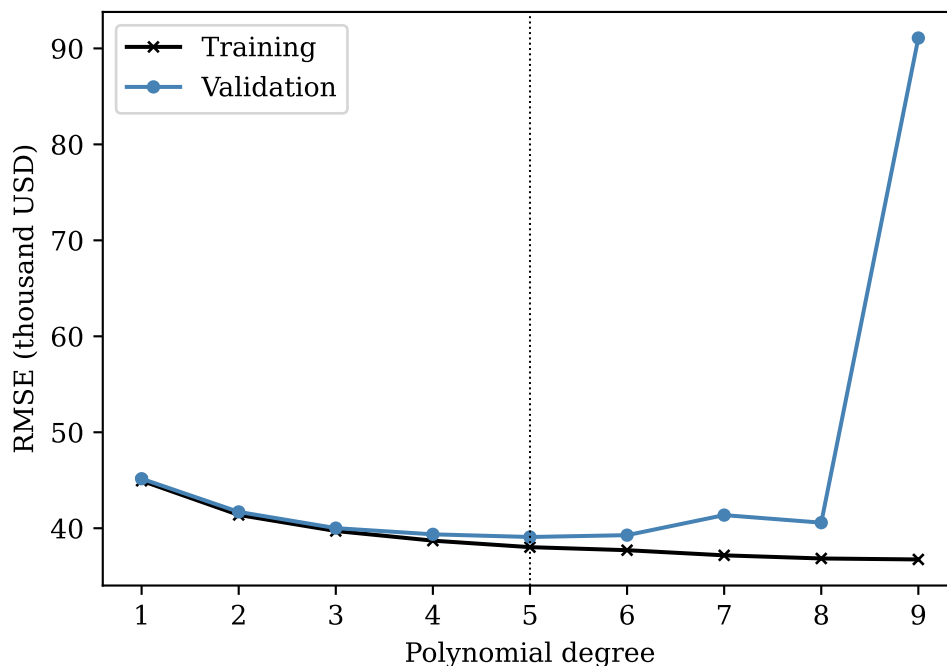
```

Parameters
-----
degrees : array-like
    Candidate polynomial degrees
rmse_train : array-like
    Mean training RMSE for each degree
rmse_validation : array-like
    Mean validation RMSE for each degree
dmin : int
    Degree with the lowest mean validation RMSE
"""
import matplotlib.pyplot as plt

_fig, ax = plt.subplots(figsize=(5, 3.5), layout='constrained')
ax.plot(degrees, rmse_train, c='black', marker='x', ms=4, label='Training')
ax.plot(
    degrees, rmse_validation, c='steelblue', marker='o', ms=4, label='Validation'
)
ax.axvline(dmin, ls=':', lw=0.75, c='black')
ax.set(xlabel='Polynomial degree', ylabel='RMSE (thousand USD)', xticks=degrees)
ax.legend()

```

```
[14]: plot_validation_curve(degrees, rmse_train, rmse_validation, dmin)
```



The smallest mean validation RMSE occurs at degree 5, about 39.1 thousand USD. The training error generally falls as the model becomes more flexible, but higher degrees have larger validation errors, indicating overfitting.

A convenience display of the same scores

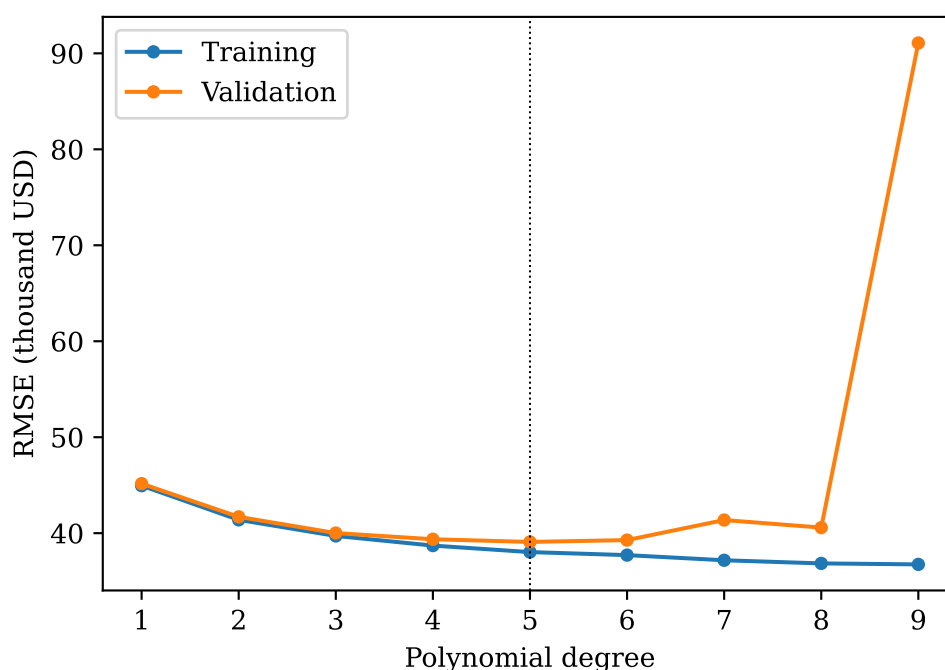
If all we are interested in is plotting the validation curve, we can use `ValidationCurveDisplay` as an alternative. We construct it from the score arrays already computed; there is no need to rerun CV. With the original negative scores, `negate_score=True` plots positive RMSE.

```
[15]: from sklearn.model_selection import ValidationCurveDisplay
```

```

# Create display object from the training and validation scores
display = ValidationCurveDisplay(
    param_name='poly__degree',
    param_range=degrees,
    train_scores=train_scores,
    test_scores=validation_scores,
    score_name='RMSE (thousand USD)',
)
# Plot the training and validation curves against the polynomial degree
fig, ax = plt.subplots(figsize=(5, 3.5), layout='constrained')
display.plot(
    ax=ax,
    negate_score=True,
    std_display_style=None,
    line_kw={'marker': 'o', 'markersize': 4},
)
ax.axvline(dmin, ls=':', lw=0.75, c='black')
ax.set(xlabel='Polynomial degree', xticks=degrees)
_ = ax.legend(['Training', 'Validation'])

```



The downside is that this approach is less flexible, for example if we want to control the plot style of the training and validation lines separately.

2.2.5 Selecting and evaluating the final model

In the previous section, we only determined the optimal polynomial degree. The next step in the CV workflow is to refit the selected model on the entire training set. Both steps can be combined and automated using `GridSearchCV`, which can optionally refit the selected model on all training observations after CV selection. `GridSearchCV` can cross-validate multiple hyperparameters simultaneously, and we pass these as a dictionary of parameter names and candidate values.

The following code illustrates this for the polynomial degree, using the same pipeline, folds, and scorer. We pass `refit=True` to refit the selected model on all training observations after CV selection.

```
[16]: from sklearn.model_selection import GridSearchCV
```

```
# Define hyperparameter grid for polynomial degree
param_grid = {'poly__degree': degrees}

# Create a GridSearchCV object
poly_cv = GridSearchCV(
    pipe, # estimator or pipeline
    param_grid=param_grid, # dictionary of candidate parameter values
    cv=cv_splitter, # cross-validation splitter
    scoring=scoring, # scoring rule
    n_jobs=4, # number of parallel jobs
    refit=True, # refit the selected model on all training observations
)

# Run cross-validation and final fit on the training sample
poly_cv.fit(X_train, y_train)
```

After the fit completes, the attribute `best_params_` gives the chosen parameter values. `best_score_` is their mean CV score, a *selection score*. `best_estimator_` contains the fitted pipeline refitted on the entire training sample, ready to predict new observations. Before doing that, we can inspect the selected polynomial degree and its mean CV score.

```
[17]: print(poly_cv.best_params_)
      print(f'CV selection RMSE: {-poly_cv.best_score_:.3f}')
```

```
{'poly__degree': np.int64(5)}
CV selection RMSE: 39.087
```

The above output confirms that the selected degree is 5, with a mean CV RMSE of about 39.1 thousand USD.

We now use the refitted pipeline to predict prices in `X_test`. We evaluate these predictions with `root_mean_squared_error()` for RMSE in thousands of USD and `r2_score()` for R^2 . These test scores evaluate the selected model on observations excluded from fitting and cross-validation.

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

```
# Reference to the refitted best estimator
estimator = poly_cv.best_estimator_

# Evaluate the refitted model on the independent test sample
y_test_pred = estimator.predict(X_test)

# Compute RMSE and  $R^2$  on the test sample
test_rmse = root_mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

print(f'Test RMSE (thousand USD): {test_rmse:.3f}')
print(f'Test  $R^2$ : {test_r2:.3f}')
```

```
Test RMSE (thousand USD): 37.626
```

```
Test  $R^2$ : 0.744
```

Your turn

1. Repeat the polynomial-degree grid search using R^2 as the scoring criterion. Keep the same training observations, candidate degrees, and CV folds. Set `scoring='r2'`; larger scores indicate better predictions.
2. Report the selected degree and its mean validation R^2 . Does the selected degree differ from the one chosen using RMSE?

2.3 Principal component regression

2.3.1 A geometric introduction to principal component analysis

Principal component regression (PCR) is based on principal component analysis (PCA), which finds perpendicular directions that successively capture the greatest remaining variation in the data. Geometrically, PCA rotates the original axes to align with the directions of greatest variation in the predictors. PCA itself is an unsupervised machine learning method that does not use the response variable. However, we'll later combine PCA with regression to form PCR, a supervised learning method. We will use cross-validation to select the number of principal components in PCR.

To make the rotation visible, we simulate 100 observations directly from a bivariate normal distribution with zero means, standard deviations of 1.5 for x_1 and 1 for x_2 , and a correlation of 0.8. This bivariate normal sample is generated by the following function:

```
[19]: def generate_bivariate_normal(N=100, sigma1=1.5, rho=0.8):
    """
    Generate a synthetic bivariate normal dataset for PCA demonstration.

    Parameters
    -----
    N : int
        Number of observations to generate
    sigma1 : float
        Standard deviation of the first variable
    rho : float
        Correlation coefficient between the two variables

    Returns
    -----
    X : array of shape (N, 2)
        Generated synthetic dataset with two centered variables
    """
    import numpy as np

    rng = np.random.default_rng(seed=1234)
    cov = np.array([[sigma1**2, sigma1 * rho], [sigma1 * rho, 1]])
    X = rng.multivariate_normal(mean=[0, 0], cov=cov, size=N)
    # Center the variables to have mean zero
    X = X - X.mean(axis=0)
    return X
```

We then fit a two-component PCA, which centers the sample without rescaling the variables. In scikit-learn, this is implemented in the `PCA` class. As usual in scikit-learn, the `fit()` method estimates the principal components from the training data, and the `transform()` method projects the original data onto these components. These two steps can be combined in a single call to `fit_transform()`.

```
[20]: from sklearn.decomposition import PCA

# Create a bivariate normal demo dataset
X = generate_bivariate_normal(N=100, rho=0.8)

# Set up PCA with 2 components
pca = PCA(n_components=2)

# Perform PCA on the synthetic dataset
X_pc = pca.fit_transform(X)

print('Explained variance fractions:', pca.explained_variance_ratio_)
```

Explained variance fractions: [0.91430695 0.08569305]

As reported above, for this synthetic sample, the first principal component captures about 91% of the variation in the predictors, while the second captures about 9%. This is to be expected given the high correlation between x_1 and x_2 .

We use the following function `plot_pca_demo()` to visualize the original data, the principal components, and the rotated data. The first plot shows the original data with the principal components as arrows. The second plot shows the data projected onto the principal components, illustrating how PCA captures the directions of greatest variance.

```
[21]: def plot_pca_demo(X, X_pc, pca):
    """
    Plot the original and PCA-transformed datasets with principal axes.

    Parameters
    -----
    X : array of shape (N, 2)
        Original dataset
    X_pc : array of shape (N, 2)
        PCA-transformed dataset
    pca : PCA object
        Fitted PCA object containing mean and components
    """
    import matplotlib.pyplot as plt
    import numpy as np

    _fig, axes = plt.subplots(
        1, 2, figsize=(6.5, 3.5), layout='constrained', sharex=True, sharey=True
    )

    # Data used for plotting the principal axes and arrows
    origins = [pca.mean_, np.zeros(2)]
    directions = [pca.components_, np.eye(2)]

    colors = ['black', 'red']
    lims = (-6, 6)

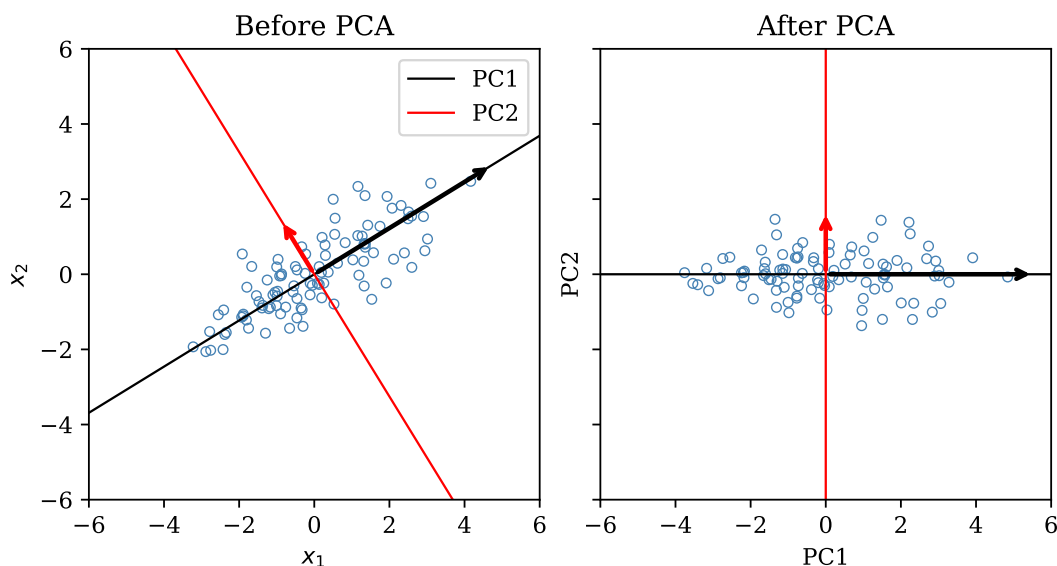
    for i, ax in enumerate(axes):
        # Plot the original or PCA-transformed data
        data = X if i == 0 else X_pc
        ax.scatter(data[:, 0], data[:, 1], s=18, c='none', ec='steelblue', lw=0.6)

        ax.set_xlabel('$x_1$' if i == 0 else 'PC1')
        ax.set_ylabel('$x_2$' if i == 0 else 'PC2')
        ax.set_title('Before PCA' if i == 0 else 'After PCA')
        ax.set(xlim=lims, ylim=lims, aspect=1)

        # Draw each principal axis and an arrow
        for j in range(2):
            # Scale the arrow length by 3 std. dev. to make it visible in the plot
            arrow = 3 * np.sqrt(pca.explained_variance_[j]) * directions[i][j]
            endpoint = origins[i] + arrow
            ax.axline(origins[i], endpoint, c=colors[j], lw=1, label=f'PC{j + 1}')
            ax.annotate(
                '',
                xy=endpoint,
                xytext=origins[i],
                arrowprops={'arrowstyle': '->', 'lw': 2, 'color': colors[j]},
            )

    axes[0].legend()
```

```
[22]: plot_pca_demo(X, X_pc, pca)
```



Before PCA, the cloud is diagonal. The black PC1 axis follows the direction with the greatest variation; the red PC2 axis is perpendicular to it. After PCA, the same cloud lies along the horizontal PC1 axis. Both panels use identical axis scales, so the change is a rotation. The arrows show the same principal directions before and after rotation, with lengths equal to three times the component standard deviations to improve visibility.

Keeping both components preserves all information in the centered data. Keeping only PC1 would replace each point with its horizontal coordinate in the right panel, discarding the smaller vertical variation along PC2.

Your turn

1. Use `generate_bivariate_normal()` with `rho=0`, keeping the other arguments unchanged. Repeat PCA and its visualization.
2. Compare the principal directions and explained variance fractions with the original example. Why do uncorrelated variables not necessarily contribute equal variance?

2.3.2 Principal component regression with polynomial features

We now return to the Ames data and use five predictors: `LivingArea`, `LotArea`, `YearBuilt`, `OverallQuality`, and `Fireplaces`. We treat `quality`, which is measured on a scale of 1-10, as a single continuous covariate here; Workshop 2 compares its numeric and categorical representations. As before, we load the data with the selected predictors and reserve 30% of the observations for testing, using the same random seed.

```
[23]: # Features used for PCR
features_pcr = ['LivingArea', 'LotArea', 'YearBuilt', 'OverallQuality', 'Fireplaces']

# Load the Ames housing data for PCR
X_pcr, y_pcr = load_ames_data(file, features_pcr, target)

# Create the same 70/30 train-test split for PCR
X_pcr_train, X_pcr_test, y_pcr_train, y_pcr_test = train_test_split(
    X_pcr, y_pcr, test_size=0.3, random_state=1234
)
```

To turn our polynomial regression pipeline into PCR, we insert PCA before the regression step. The two scalers have different purposes: scaling the inputs improves the numerical behavior of powers and interactions, while scaling the expanded columns gives them comparable weight in PCA. We use `PCA(..., svd_solver='full')` because the covariance-based solver is more sensitive to numerical errors when the polynomial columns are highly correlated.

Our scaling choices affect which directions PCA identifies as having the most variation. If we scale the polynomial features differently, PCA may retain different components, and the resulting PCR model may make different predictions. Scaling can help make the calculations more stable, but we still need cross-validation to check how accurately the model predicts house prices.

We also fit PCA only on the training observations in each CV split. For validation, we reuse both the training scaling statistics and the fitted PCA directions. We measure the validation observations along these existing directions; we do not fit a new PCA to them. The same applies to the test sample after the final refit.

The transformed validation or test predictors need not have mean zero and standard deviation one. PCA can still project them onto the fitted directions. Keeping these directions fixed gives the regression the same component definitions it learned during training.

For now, we set `n_components=0.9`. A fractional value tells PCA to retain enough components to explain more than that fraction of predictor variance. However, components that explain little predictor variance may still help predict prices. We will therefore later use predictive CV scores to choose this threshold.

```
[24]: # Target fraction of variance to retain in PCA
pca_variance = 0.9

# Build a pipeline for degree-5 PCR with PCA
pipe_pcr = Pipeline(
    steps=[
        ('input_scale', StandardScaler()),
        ('poly', PolynomialFeatures(degree=5, include_bias=False)),
        ('scale', StandardScaler()),
        ('pca', PCA(n_components=pca_variance, svd_solver='full')),
        ('lr', LinearRegression(fit_intercept=True)),
    ]
)
```

2.3.3 PCR versus simple polynomial regression

We first compare polynomial regression with PCR while holding the degree fixed at 5. Both models use the same five predictors, observations, scaling, scorer, and folds, so the only difference is the PCA step. This lets us see how reducing the expanded predictor space affects an overly flexible polynomial model.

For convenience, we repeat the simple polynomial pipeline code from the previous section:

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

# Build a pipeline for the simple polynomial regression with degree 5
poly5 = Pipeline(
    steps=[
        ('input_scale', StandardScaler()),
        ('poly', PolynomialFeatures(degree=5, include_bias=False)),
        ('scale', StandardScaler()),
        ('lr', LinearRegression(fit_intercept=True)),
    ]
)
```

We can now run cross-validation on both pipelines and compare their mean validation RMSEs.

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

# Compute cross-validation scores for the simple polynomial
poly5_scores = cross_val_score(
    poly5, # estimator or pipeline
    X_pcr_train, # training predictors
    y_pcr_train, # training target
    cv=cv_splitter, # cross-validation splitter
    scoring=scoring, # scoring rule
    n_jobs=4, # number of parallel jobs
)

# Compute cross-validation scores for the PCR model
pcr5_scores = cross_val_score(
    pipe_pcr, # estimator or pipeline
    X_pcr_train, # training predictors
    y_pcr_train, # training target
    cv=cv_splitter, # cross-validation splitter
    scoring=scoring, # scoring rule
    n_jobs=4, # number of parallel jobs
)

# Compare the mean validation RMSEs for both models
df_degrees5 = pd.DataFrame(
    {
        'Model': ['Degree 5 polynomial', 'Degree 5 PCR'],
        'Mean validation RMSE': [(-poly5_scores).mean(), (-pcr5_scores).mean()],
    }
)
df_degrees5
```

```
[26]:
```

	Model	Mean validation RMSE
0	Degree 5 polynomial	246.254881
1	Degree 5 PCR	39.528431

At degree 5, PCR reduces mean validation RMSE from about 246.3 to 39.5 thousand USD. Since both models use the same features, we can attribute this difference to the PCA step.

To see which components PCA retains, we fit the PCR pipeline on the entire training sample and inspect its attributes. This is necessary because the CV functions fitted their own copies of the pipeline. We can access each fitted transformation through the `named_steps` dictionary.

```
[27]: # Refit the PCR pipeline on the entire training sample
pipe_pcr.fit(X_pcr_train, y_pcr_train)

# Report number of expanded features and retained components
pca_step = pipe_pcr.named_steps['pca']
print(f'Expanded features: {pca_step.n_features_in_}')
print(f'Retained components: {pca_step.n_components_}')
```

```
Expanded features: 251
Retained components: 21
```

The polynomial expansion creates 251 features, but PCA retains only 21 components. Together, these components explain at least 90% of the variation in the expanded predictors, as specified by `pca_variance`.

We can also look at how much variance each component explains. We extract these fractions from the fitted estimator as follows:

```
[28]: # Obtain the explained variance fractions for the retained components
var_expl = pca_step.explained_variance_ratio_
```

We can then use these statistics to create the bar chart:

```
[29]: def plot_pcr_variance(var_expl, n_comp=10, target_var=0.9):
    """
    Plot the explained variance fractions for the retained components in PCR.

    Parameters
    -----
    var_expl : array of shape (n_components,)
        Explained variance fractions for the retained components
    n_comp : int
        Number of components to plot
    target_var : float
        Target variance to indicate on the plot
    """
    import matplotlib.pyplot as plt
    import numpy as np

    _fig, ax = plt.subplots(figsize=(6, 3.5), layout='constrained')

    # Component number shown on the x-axis
    n_comp = min(n_comp, len(var_expl))
    components = np.arange(1, n_comp + 1)

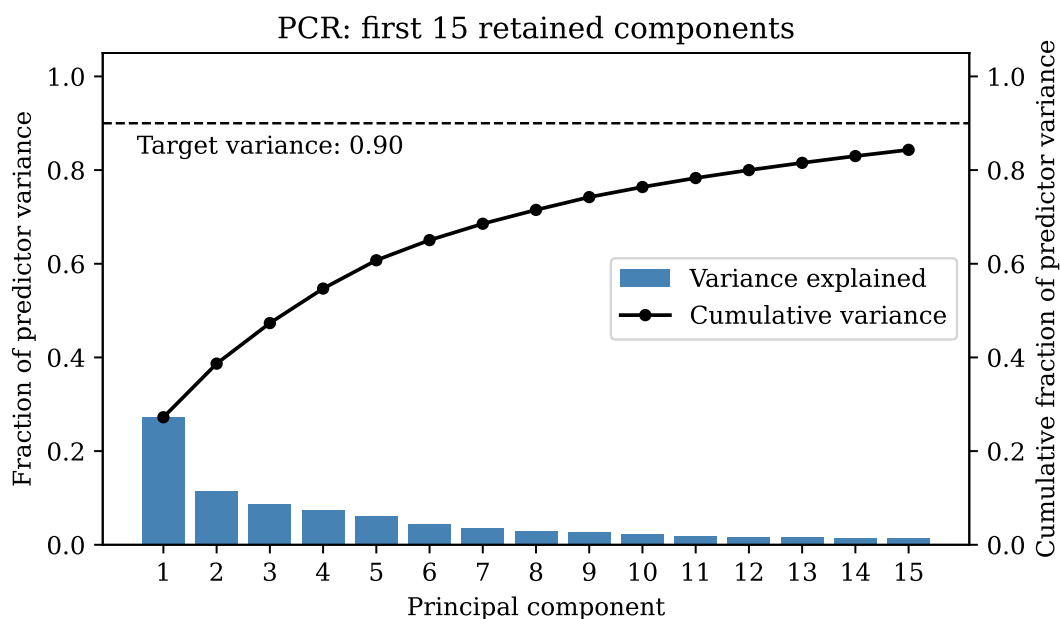
    bars = ax.bar(components, var_expl[:n_comp], color='steelblue')
    ax.set_ylabel('Fraction of predictor variance')
    ax.set_xlabel('Principal component')
    ax.set_xticks(components)
    ax.set_ylim(0, 1.05)

    # Cumulative explained variance on a second y-axis
    ax_cum = ax.twinx()
    cum_var_expl = np.cumsum(var_expl)
    cumul = ax_cum.plot(components, cum_var_expl[:n_comp], c='black', marker='o', ms=4)
    ax_cum.set_ylabel('Cumulative fraction of predictor variance')
    ax_cum.set_ylim(0, 1.05)
    if target_var is not None:
        ax_cum.axhline(target_var, c='black', ls='--', lw=1)
        ax.text(
            0.5,
            target_var - 0.025,
            f'Target variance: {target_var:.2f}',
            c='black',
            va='top',
            ha='left',
            fontsize=10,
        )

    ax.legend(
        [bars, cumul[0]],
        ['Variance explained', 'Cumulative variance'],
        loc='center right',
    )
    ax.set_title(f'PCR: first {n_comp} retained components')
```

```
[30]: # Number of components to plot
n_comp = 15

plot_pcr_variance(var_expl, n_comp=n_comp, target_var=pca_variance)
```



As you can see from the plot, the first component explains about 27% of the variance and the second about 11%. The black line shows the cumulative fraction explained by the first k components. The first 15 components together explain about 84%, so the curve has not yet reached the dashed 90% threshold. We need 21 components to reach that threshold, which is why PCA retains 21 components in the fitted pipeline.

Your turn

1. Increase the PCA variance threshold from 0.9 to 0.95, keeping polynomial degree 5. Refit using the existing training data and repeat cross-validation with the same folds and scoring rule.
2. Compare the retained component count and mean validation RMSE with the 90% model. Does retaining more predictor variance improve prediction?

2.3.4 Cross-validating multiple hyperparameters

We can now use the grid search introduced earlier to choose two parameters jointly: polynomial degree and the fraction of predictor variance retained by PCA. The names `poly__degree` and `pca__n_components` follow the same `step__parameter` convention as before.

With four degrees and five variance thresholds, we have 20 combinations to compare. We use the same shuffled five-fold splitter and negative-RMSE scorer. With `refit=True`, `GridSearchCV` then refits the selected pipeline on the entire five-predictor training sample.

We redefine the CV splitter and scoring metric here for clarity, but they are the same as before since we are using the same seed.

```
[31]: from sklearn.model_selection import GridSearchCV, KFold

# Candidate degrees and variance thresholds (20 combinations)
param_grid = {
    'poly__degree': [2, 3, 4, 5],
    'pca__n_components': [0.9, 0.95, 0.975, 0.99, 0.999],
}

# Define the CV splitter
```

```

cv_splitter = KFold(n_splits=5, shuffle=True, random_state=1234)

# Define the scoring metric
scoring = 'neg_root_mean_squared_error'

# Create a GridSearchCV object for PCR
pcr_cv = GridSearchCV(
    pipe_pcr, # estimator or pipeline
    param_grid, # candidate parameter values
    cv=cv_splitter, # cross-validation splitter
    scoring=scoring, # scoring rule
    n_jobs=4, # number of parallel jobs
    refit=True, # refit on all training observations
)

pcr_cv.fit(X_pcr_train, y_pcr_train)

```

Since we are now cross-validating two hyperparameters, we can tabulate the RMSE by both polynomial degree and PCA variance threshold. GridSearchCV calls the model for each combination of hyperparameters and returns the results in the `cv_results_` attribute. We convert this dictionary to a DataFrame for easier inspection:

```

[32]: # Tabulate RMSEs by polynomial degree and PCA variance threshold
df_pcr_rmse = pd.DataFrame(
    {
        'CV RMSE': -pcr_cv.cv_results_['mean_test_score'],
        'Poly degree': pcr_cv.cv_results_['param_poly__degree'],
        'PCA variance': pcr_cv.cv_results_['param_pca__n_components'],
    }
).pivot(index='Poly degree', columns='PCA variance', values='CV RMSE').round(3)
df_pcr_rmse

```

```

[32]: PCA variance    0.900    0.950    0.975    0.990    0.999
Poly degree
2          29.063   29.088   28.852   28.860   28.891
3          28.872   28.710   28.088   27.897   28.000
4          32.814   28.425   28.298   28.557   31.277
5          39.528   30.910   31.313   30.102   45.827

```

As the table shows, the optimal variance threshold varies with the polynomial degree. For example, for degree 2, the best threshold is 0.975, whereas for degree 3 it's 0.99, which also is the overall smallest RMSE for all possible parameter combinations.

As before, the attribute `best_params_` tells us which parameter values were selected. Since our scorer returns negative RMSE, we negate `best_score_` to report the CV selection RMSE. This is the mean validation RMSE for the selected PCR candidate.

```

[33]: pcr_cv.best_params_

```

```

[33]: {'pca__n_components': 0.99, 'poly__degree': 3}

```

```

[34]: print(f'PCR CV selection RMSE: {-pcr_cv.best_score_:.3f}')

```

```

PCR CV selection RMSE: 27.897

```

We can also inspect the refitted PCA step to see how many expanded polynomial features it received and how many components it retained. A fractional variance threshold can lead to different numbers of components in different folds. The counts below refer to the final refit on the entire training sample.

```

[35]: # Extract the PCA step from the best estimator
best_pca = pcr_cv.best_estimator_.named_steps['pca']

# Store number of features entering the model

```

```

features_in = best_pca.n_features_in_
# Store number of retained components
n_components = best_pca.n_components_

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

```

Number of expanded polynomial features: 55
 Number of retained principal components: 40

The search selects degree 3 with a variance threshold of 0.99 and a CV RMSE of about 27.9 thousand USD. The polynomial expansion creates 55 features, and the refitted PCR model retains 40 components. PCA therefore removes 15 components while retaining at least 99% of the variation in the expanded predictors.

2.3.5 Evaluating the selected models on the test sample

Our earlier polynomial degree search used only two predictors. Since we now have five predictors, we need to select the polynomial degree again for this feature set. We use the same degrees from 2 to 5, CV folds, and scoring metric as in the PCR search. This lets each method choose its own model complexity: polynomial regression selects the degree, while PCR also selects how much predictor variance to retain.

With `refit=True`, each search refits its selected model on the entire training sample.

```

[36]: # Select among degrees 2 to 5 using the same CV setup as PCR
poly_param_grid = {'poly__degree': [2, 3, 4, 5]}

poly_cv = GridSearchCV(
    poly5, # estimator or pipeline
    poly_param_grid, # candidate parameter values
    cv=cv_splitter, # cross-validation splitter
    scoring=scoring, # scoring rule
    n_jobs=4, # number of parallel jobs
    refit=True, # refit on all training observations
)
poly_cv.fit(X_pcr_train, y_pcr_train)

print(poly_cv.best_params_)
print(f'Polynomial CV selection RMSE: {-poly_cv.best_score_:.3f}')

```

```

{'poly__degree': 3}
Polynomial CV selection RMSE: 27.776

```

We can now compute the predictions for both models on the same independent test sample. We then compute `root_mean_squared_error()` for RMSE in thousands of USD and `r2_score()` for R^2 to compare the prediction accuracy.

```

[37]: from sklearn.metrics import r2_score, root_mean_squared_error

# Predict on test sample and compute RMSE and R^2 for simple polynomial model
y_poly_pred = poly_cv.predict(X_pcr_test)
poly_test_rmse = root_mean_squared_error(y_pcr_test, y_poly_pred)
poly_test_r2 = r2_score(y_pcr_test, y_poly_pred)

# Predict on test sample and compute RMSE and R^2 for PCR model
y_pcr_pred = pcr_cv.predict(X_pcr_test)
pcr_test_rmse = root_mean_squared_error(y_pcr_test, y_pcr_pred)
pcr_test_r2 = r2_score(y_pcr_test, y_pcr_pred)

# Compare the selected degrees, model dimensions, and test scores
poly_features = poly_cv.best_estimator_.named_steps['poly'].n_output_features_
df_comparison = pd.DataFrame(

```

```

{
    'Model': ['Selected polynomial', 'Selected PCR'],
    'Degree': [
        poly_cv.best_params_['poly__degree'],
        pcr_cv.best_params_['poly__degree'],
    ],
    'Regressors': [poly_features, n_components],
    'Test RMSE': [poly_test_rmse, pcr_test_rmse],
    'Test R²': [poly_test_r2, pcr_test_r2],
}
)
print(df_comparison.to_string(index=False))

```

	Model	Degree	Regressors	Test RMSE	Test R²
	Selected polynomial	3	55	28.588499	0.852468
	Selected PCR	3	40	28.097610	0.857491

The polynomial and PCR cross-validations both select degree 3. Their CV selection RMSEs are 27.776 and 27.897 thousand USD, respectively. On the test sample, polynomial regression has RMSE of about 28.6 thousand USD, compared with 28.1 for PCR. The selected polynomial model uses 55 features; PCA reduces PCR's 55 expanded features to 40 components.

This result depends on the sample and model-selection setup, including the train-test split, CV folds, random seed, and candidate values. You should not conclude that PCR always performs better on the test set. In fact, the results here suggest that both methods perform very similarly, but PCR achieves this result with fewer regressors.

Your turn

1. Add `OverallCondition` to the five PCR predictors. Keep the same training observations in the same order and reuse `cv_splitter`.
2. Rerun the same 20-combination grid search. Compare the CV RMSE, selected degree, variance threshold, and number of retained components with the five-predictor results. Use only the training sample for this comparison.
3. Does adding `OverallCondition` improve validation accuracy, and does the selected regression use fewer or more regressors? Explain how this relates to the selected polynomial degree.

2.4 List of functions used in this lecture

Each function name links to its official API documentation. The [\[go to section\]](#) links point to the relevant discussion in this notebook.

2.4.1 NumPy functions

- [argmin\(\)](#) Return the indices of minimum values along an axis. [\[go to section\]](#)

2.4.2 scikit-learn classes and functions

Regression models (`sklearn.linear_model`)

- [LinearRegression](#) Fit ordinary least squares linear regression. [\[go to section\]](#)

Model selection and cross-validation (**sklearn.model_selection**)

- `train_test_split()` Split observations into training and test samples. [\[go to section\]](#)
- `KFold` Divide observations into folds for cross-validation. [\[go to section\]](#)
- `cross_val_score()` Evaluate a single metric for an estimator using cross-validation. [\[go to section\]](#)
- `cross_validate()` Evaluate one or more metrics using cross-validation and return timing data. [\[go to section\]](#)
- `validation_curve()` Compute training and validation scores over a range of parameter values. [\[go to section\]](#)
- `ValidationCurveDisplay` Plot training and validation scores as a function of a parameter value. [\[go to section\]](#)
- `GridSearchCV` Search specified hyperparameter combinations exhaustively using cross-validation. [\[go to section\]](#)

Evaluation metrics (**sklearn.metrics**)

- `root_mean_squared_error()` Compute prediction RMSE. [\[go to section\]](#)
- `r2_score()` Compute the coefficient of determination. [\[go to section\]](#)

Feature preprocessing (**sklearn.preprocessing**)

- `PolynomialFeatures` Generate polynomial and interaction features. [\[go to section\]](#)
- `StandardScaler` Standardize features by removing their mean and scaling to unit variance. [\[go to section\]](#)

Dimensionality reduction (**sklearn.decomposition**)

- `PCA` Perform linear dimensionality reduction using principal component analysis. [\[go to section\]](#)

Pipelines (**sklearn.pipeline**)

- `Pipeline` Chain transformers and a final estimator. [\[go to section\]](#)

3 Regression models with scikit-learn

3.1 Motivation

In the previous lecture, we studied linear models of the form

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

where μ is the intercept (or bias in machine learning lingo), $\mathbf{x}$ is a vector of explanatory variables, and ϵ_i is an additive error term that captures any residual variance in y_i not accounted for by the explanatory variables.

We estimated the coefficients using ordinary least squares (OLS), which is equivalent to minimizing the sum of squared errors,

$$L(\mu, \boldsymbol{\beta}) = \underbrace{\sum_{i=1}^N (y_i - \mu - \mathbf{x}_i' \boldsymbol{\beta})^2}_{\text{Sum of squared errors}} \quad (1)$$

The estimates $(\hat{\mu}, \hat{\boldsymbol{\beta}})$ are then the values that minimize $L(\mu, \boldsymbol{\beta})$,

$$(\hat{\mu}, \hat{\boldsymbol{\beta}}) = \arg \min_{\mu, \boldsymbol{\beta}} L(\mu, \boldsymbol{\beta})$$

While OLS is the best linear unbiased estimator (BLUE) under the usual assumptions about the errors ϵ_i , the estimated $(\hat{\mu}, \hat{\boldsymbol{\beta}})$ may not be the best model parameters when it comes to minimizing *out-of-sample* prediction errors. Whenever our focus is on *prediction* rather than on *inference* about population parameters, we can often do better than using the loss function from (1). To this end, we augment the loss function with penalty terms that can reduce overfitting and improve out-of-sample prediction.

3.2 Synthetic sample and polynomial approximation

3.2.1 Synthetic sample

We use a highly stylized example to study polynomial approximation and overfitting. We want to approximate a (non-linear) trigonometric function using a high-order polynomial, a setting that is notoriously susceptible to overfitting (see also the previous lecture and workshop for illustrations). Assume that our model is given by

$$y_i = \cos\left(\frac{3\pi}{200} x_i\right) + \epsilon_i$$
$$\epsilon_i \stackrel{\text{iid}}{\sim} \mathcal{N}(0, 0.5^2)$$

where ϵ_i is an additive, normally distributed measurement error term with mean 0 and variance $\frac{1}{4}$. For this exercise, we assume that x is uniformly distributed on the interval $[0, 100]$.

The true values of y (without measurement error) are computed using the function `compute_true_y()`, which returns y for a given x .

```
[1]: import numpy as np
```

```
def compute_true_y(x):
    """
    True trigonometric function (without errors)
    """
    return np.cos(3 * np.pi / 200.0 * x)
```

The function `create_trig_sample()` defined below is used to generate the training and test samples.

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

def create_trig_sample(N=100, sigma=0.5, rng=None):
    """
    Create trigonometric relationship sample data.

    Parameters
    -----
    N : int
        Sample size.
    sigma : float
        Standard deviation of the normal error term.
    rng : np.random.Generator, optional
        Random number generator.
    """

    # Initialize random number generator
    if rng is None:
        rng = default_rng(seed=12345)

    # Randomly draw explanatory variable x uniformly distributed on [0, 100]
    x = rng.uniform(0, 100, size=N)

    # Draw errors from normal distribution
    epsilon = rng.normal(scale=sigma, size=N)

    # Compute y, add measurement error
    y = compute_true_y(x) + epsilon

    return x, y
```

This lecture uses a training sample with $N = 100$ observations and the fixed seed 12345. We create `x_train`, `y_train`, and the feature matrix `X_train` once for all models.

We define `DEGREE = 15` which controls the maximum degree of the polynomial features used in all models in this lecture. Additionally, we define a common prediction grid, `xvalues`, which we use for plotting the model predictions. Its matrix form, `X_grid`, is used as the input for a model's `predict()` method.

```
[3]: # Polynomial degree used throughout the notebook
DEGREE = 15

# Training sample and two-dimensional feature matrix
x_train, y_train = create_trig_sample()
X_train = x_train[:, None]

# Common grid for prediction plots within the training range
xvalues = np.linspace(x_train.min(), x_train.max(), 500)
# Matrix of features used for predict()
X_grid = xvalues[:, None]
```

The next graph visualizes the true relationship and the sample points (x_i, y_i) . For this purpose, we first define the function `plot_trig_sample()`, which creates the baseline plot used throughout the next few sections.

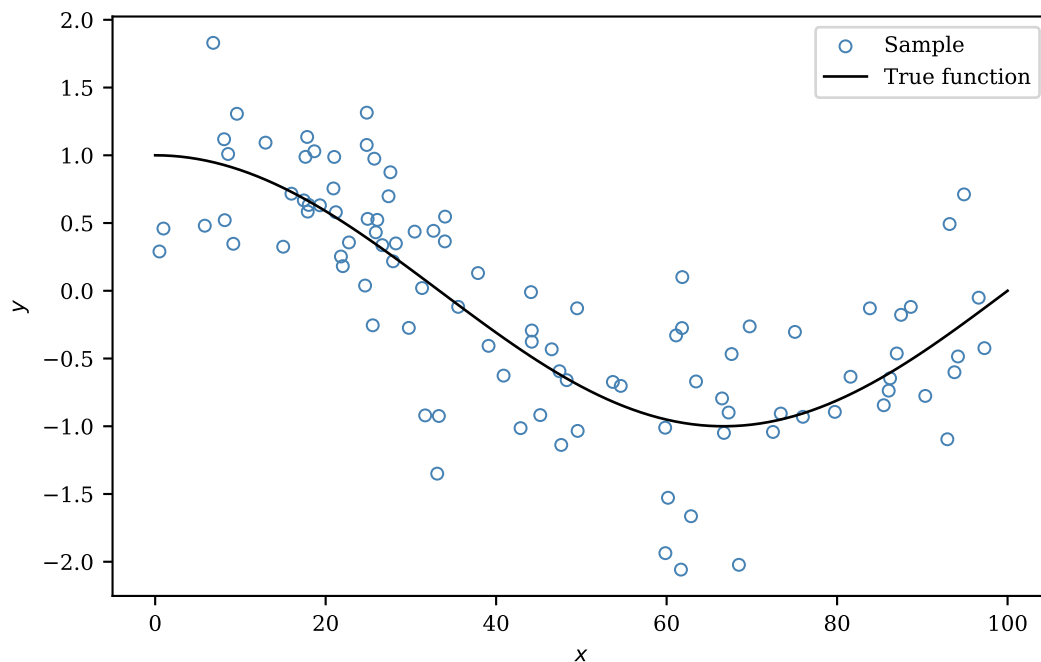
```
[4]: def plot_trig_sample(x, y):
    """
    Plot the trigonometric relationship and sample.
    """
    import matplotlib.pyplot as plt

    # Sample scatter plot
    _, ax = plt.subplots(figsize=(5.5, 3.5), layout='constrained')
    ax.scatter(x, y, s=20, c='none', edgecolor='steelblue', lw=0.75, label='Sample')

    # Plot true relationship
    x_true = np.linspace(0.0, 100.0, 101)
    y_true = compute_true_y(x_true)
    ax.plot(x_true, y_true, c='black', lw=1.0, label='True function')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.legend(loc='upper right')

    return ax
```

```
[5]: _ = plot_trig_sample(x_train, y_train)
```



3.2.2 Polynomial approximation

We now estimate a polynomial approximation where we assume that y is a degree- K polynomial in x , i.e.,

$$y_i \approx \mu + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_K x_i^K$$

For this example, we choose an unconventionally high $K = 15$, set by the constant `DEGREE`, since we anticipate that this leads to problems with OLS. These problems are further amplified by the small sample size of $N = 100$ observations. We make these choices for pedagogical reasons so overfitting is visually more apparent.

As in Lecture 2, we use two scaling steps before fitting the model. The complete pipeline is as follows:

1. We center and scale the original input using `StandardScaler()`. Writing

$$z_i = \frac{x_i - \bar{x}}{s_x},$$

we subtract the training mean $\bar{x}$ and divide by the training standard deviation s_x . Centering can reduce correlations between powers, while scaling keeps their magnitudes manageable.

2. We then create powers of z with `PolynomialFeatures()`. We use `include_bias=False` because the regression step fits the intercept.
3. A second `StandardScaler()` standardizes the expanded columns. Even when z has variance one, its powers generally have different variances. This second scaler puts them on comparable scales.
4. Finally, we fit a regression model to the transformed features.

The two scalers have different purposes. With our training sample, the correlation between x and x^2 is about 0.972, while the correlation between z and z^2 is about 0.383. High powers can still be strongly correlated after input standardization.

We build a `Pipeline` that combines these transformations with the estimation step. We use the `make_pipeline()` convenience function to simplify this step.

3.2.3 OLS benchmark

We first estimate the polynomial using ordinary least squares, with `LinearRegression()` as the final step of the pipeline:

```
[6]: from sklearn.linear_model import LinearRegression
      from sklearn.pipeline import make_pipeline
      from sklearn.preprocessing import PolynomialFeatures, StandardScaler

      # Create pipeline for linear model
      pipe_ols = make_pipeline(
          StandardScaler(),
          PolynomialFeatures(degree=DEGREE, include_bias=False),
          StandardScaler(),
          LinearRegression(fit_intercept=True),
      )

      # Fit model
      pipe_ols.fit(X_train, y_train)
```

We can use the fitted model to predict the outcomes for the fixed grid which we use for plotting the model predictions:

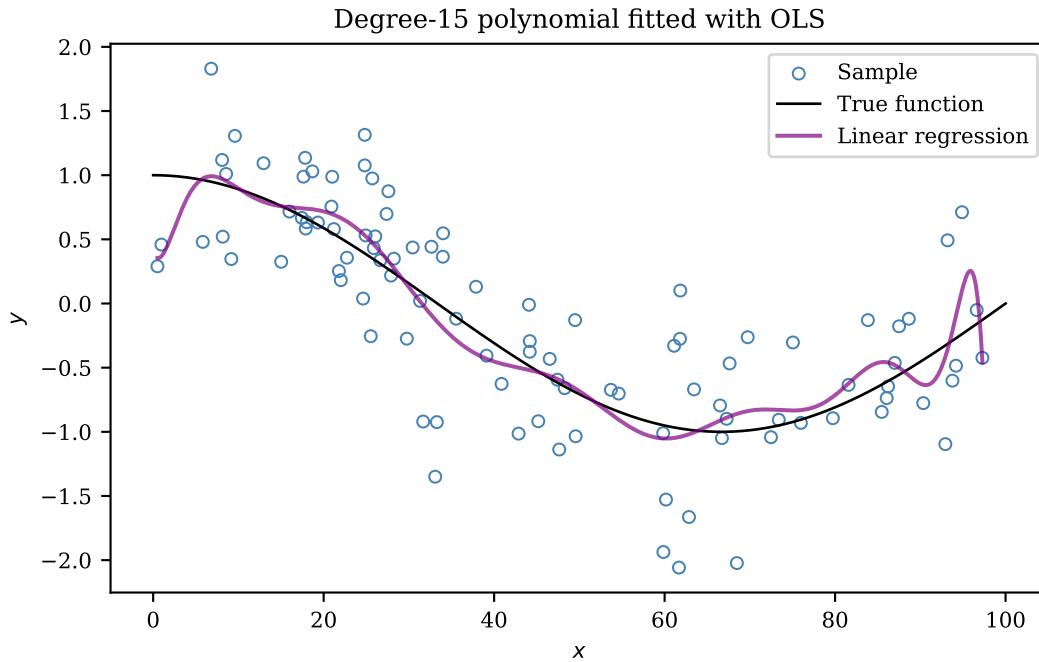
```
[7]: # OLS predictions on the common grid
      yhat_grid_ols = pipe_ols.predict(X_grid)
```

The following cell plots these predicted values together with the true function and the training sample:

```
[8]: # Plot sample and true relationship
      ax = plot_trig_sample(x_train, y_train)

      # OLS prediction
      ax.plot(xvalues, yhat_grid_ols, c='purple', alpha=0.7, label='Linear regression')

      ax.set_title(f'Degree-{DEGREE} polynomial fitted with OLS')
      _ = ax.legend()
```



In theory, for OLS with an intercept and a complete polynomial expansion, centering and scaling leave the fitted polynomial unchanged in exact arithmetic. The estimated coefficients change to describe the transformed features. In practice, highly correlated polynomial columns can make the calculations unstable. The solver may discard nearly dependent directions, changing the predictions. In our degree-15 example, omitting the input scaler can produce this kind of numerical truncation even when we standardize the expanded columns.

However, as the figure above shows, more stable calculations do not prevent overfitting: the fitted polynomial can still follow the noise in our small sample. We now turn to regularization methods, which address this problem by explicitly penalizing the coefficients.

3.3 Ridge regression

Our first regularized model is Ridge regression, which adds a penalty term to the OLS loss function:

$$L(\mu, \beta) = \underbrace{\sum_{i=1}^N (y_i - \mu - \mathbf{x}_i' \beta)^2}_{\text{Sum of squared errors}} + \underbrace{\alpha \sum_{k=1}^K \beta_k^2}_{\text{L2 penalty}}$$

Throughout this lecture, we write the objectives using scikit-learn's conventions. Other formulations are possible, for example using the mean rather than the sum of squared errors, but this changes the interpretation of the penalty parameter.

Compared to ordinary least squares (OLS), the penalty term increases the loss when the estimated coefficients β are large in absolute value. This term is called an L2 penalty because it corresponds to the squared L2 vector norm. In many textbooks, we see it written as $\alpha \|\beta\|_2^2$, which is equivalent to the formula above. For any $\alpha > 0$, the penalty increases with the squared coefficient values. The estimated coefficients therefore balance fitting the data against keeping their overall size small. We call this *regularization* or *shrinkage*. The intercept μ is not included in the penalty term, so it is not directly penalized.

As α approaches zero, the estimates approach the OLS solution. Conversely, for large α , the coefficients approach zero. Individual coefficients need not decrease steadily in absolute value, especially when the explanatory variables are highly correlated. In this setting, α is a hyperparameter, and we can use cross-validation to select its value.

Why would we use Ridge regression given that OLS is the best linear unbiased estimator? Regularization can help when we have many, potentially highly correlated, regressors, in which case OLS is prone to overfitting.

Importance of scaling and centering

For Ridge, the second StandardScaler after the polynomial expansion puts the expanded features on comparable scales for the coefficient penalty. High polynomial powers can still be strongly correlated, so input standardization does not eliminate the need for regularization.

Centering also changes what regularization penalizes. Powers of x and powers of z can represent the same polynomial functions, but they use different coefficients. Penalizing these coefficients therefore gives different fitted models, even after we standardize the expanded columns. While input standardization can improve numerical stability, it need not improve prediction accuracy.

3.3.1 Estimating the Ridge model

We use the same preprocessing as in the OLS pipeline, with `Ridge()` as the final step.

To run Ridge regression, we also need to specify the regularization strength α , which we set to $\alpha = 3$ for illustration.

```
[9]: from sklearn.linear_model import Ridge

# Penalty strength to use for Ridge
alpha = 3

# Build pipeline of transformations and Ridge estimation.
# We create the polynomial transformation w/o the intercept so we
# need to include an intercept in the fitting step.
pipe_ridge = make_pipeline(
    StandardScaler(), # center and scale the original input
    PolynomialFeatures(degree=DEGREE, include_bias=False),
    StandardScaler(), # standardize expanded features
    Ridge(alpha=alpha, fit_intercept=True), # Fit Ridge regression
)

# Fit model
pipe_ridge.fit(X_train, y_train)
```

We now add the Ridge predictions to the OLS plot to compare the two fitted polynomials.

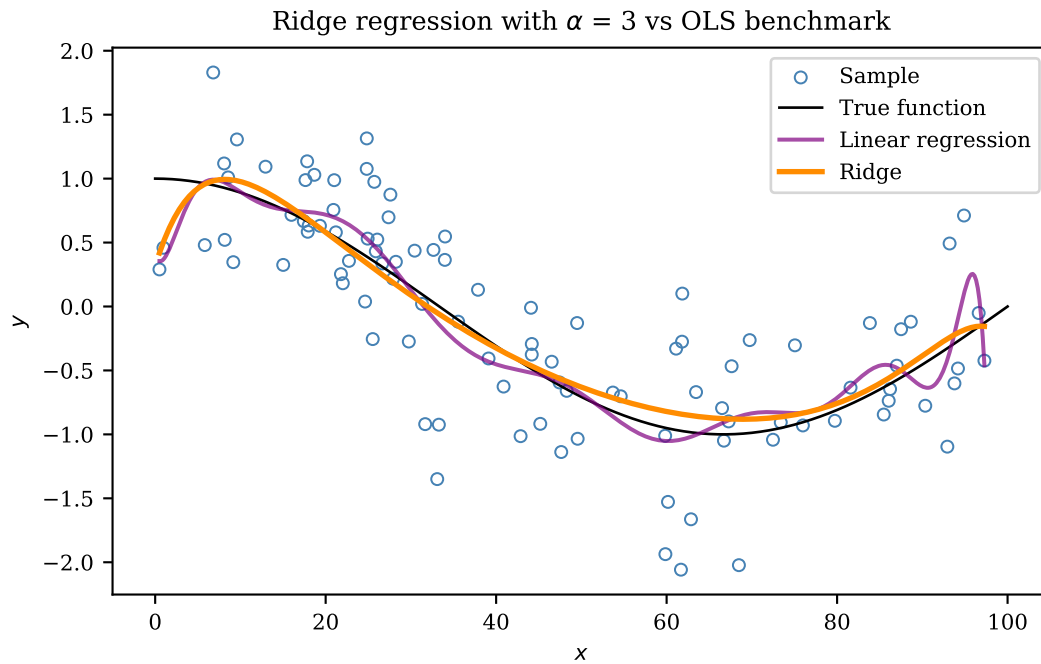
```
[10]: # Ridge predictions on the common grid
yhat_grid_ridge = pipe_ridge.predict(X_grid)

[11]: # Plot sample and true relationship
ax = plot_trig_sample(x_train, y_train)

# Linear regression prediction
ax.plot(xvalues, yhat_grid_ols, c='purple', alpha=0.7, label='Linear regression')

# Ridge prediction
ax.plot(xvalues, yhat_grid_ridge, c='darkorange', lw=2.0, label='Ridge')

ax.set_title(rf'Ridge regression with $\alpha$ = {alpha:.4g} vs OLS benchmark')
_ = ax.legend()
```



The OLS prediction follows sample fluctuations and oscillates near the right edge of the training sample. The Ridge prediction is smoother, although it also follows some of the noise. Later, we will compare prediction errors on new observations to assess how well the models generalize.

Your turn

1. Comment out the first `StandardScaler()` in the Ridge and OLS pipelines above. Rerun the fits, predictions, and graph.
2. Restore the first scaler, then comment out the `StandardScaler()` after `PolynomialFeatures`. Rerun and inspect the graph again.

How does removing each scaler affect the Ridge predictions relative to OLS? Why does feature scaling matter when we penalize coefficients?
Restore both scaling steps before continuing.

3.3.2 Ridge path

To gain some intuition for what is going on, it is instructive to plot the estimated Ridge coefficients for a whole range of α values as we do in the code below. This plot is called the *coefficient path* or *Ridge path* in the context of Ridge regression.

Note that we choose the grid of α to be uniformly spaced on a logarithmic scale since we want to zoom in on what happens when α is small. This is accomplished by using `np.logspace()` instead of `np.linspace()`.

We reuse the Ridge pipeline defined above. The `set_params()` method updates its regularization strength through `ridge__alpha`, combining the step name and parameter name with two underscores. We then refit the pipeline and recover the fitted coefficients from `named_steps['ridge']`.

Each call to `fit()` also refits the preprocessing steps. Since we use the same sample for every value of α , the transformed features remain the same.

```
[12]: # Create grid of alphas spaced uniformly in logs on [1e-4, 100]
      alphas = np.logspace(start=np.log10(1.0e-4), stop=np.log10(100), num=200)
```

3 Regression models with scikit-learn

```
# Array to store one coefficient for each polynomial feature and alpha
coefs = np.empty((len(alphas), DEGREE))

# Reuse the Ridge pipeline, updating alpha before each fit
for i, alpha in enumerate(alphas):
    # Update the penalty in the Ridge step
    pipe_ridge.set_params(ridge__alpha=alpha)

    # Refit the pipeline on the original input
    pipe_ridge.fit(X_train, y_train)

    # Store the coefficients for this alpha
    coefs[i] = pipe_ridge.named_steps['ridge'].coef_
```

The following plot shows the coefficient on each standardized power of the centered and scaled input z for different values of α . These coefficients refer to the transformed features, rather than to the original powers of x . The x-axis uses a $\log_{10}$ scale so that we can see the behavior at small values of α more clearly.

We define `plot_coefficient_path()` to draw this plot. The `model_name` argument lets us reuse the function for both Ridge and Lasso:

```
[13]: def plot_coefficient_path(alphas, coefs, model_name):
    """
    Plot coefficients for different regularization strengths.

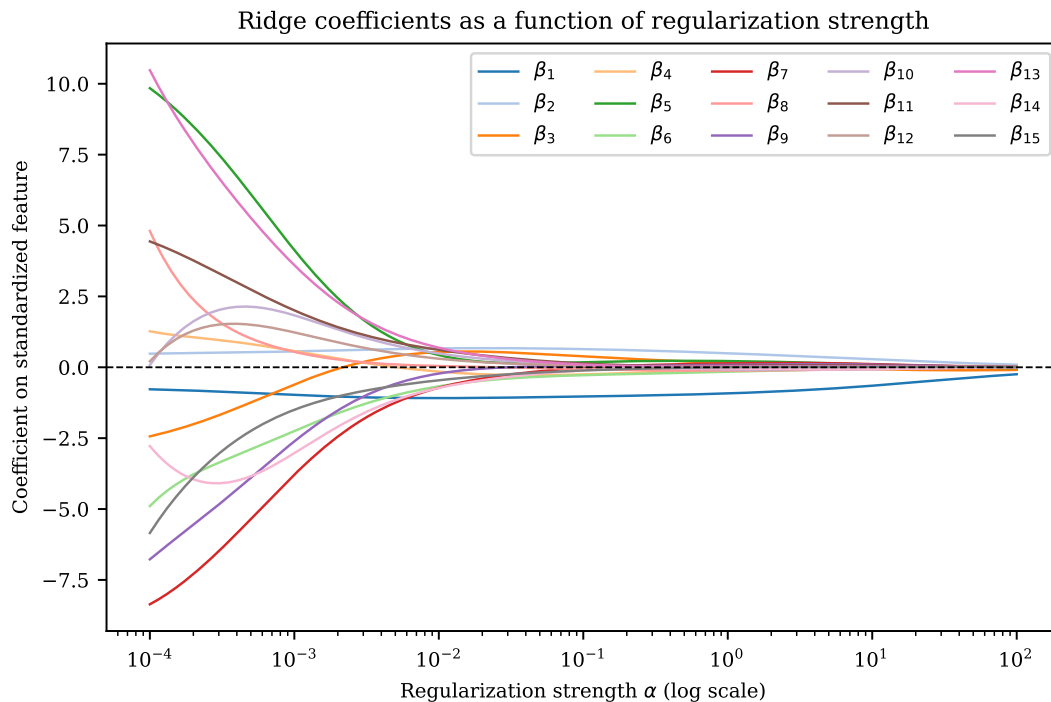
    Parameters
    -----
    alphas : array
        Regularization strengths
    coefs : array
        Coefficients, with one row per strength and one column per feature
    model_name : str
        Model name, either 'Ridge' or 'Lasso'
    """
    import matplotlib.pyplot as plt

    _, ax = plt.subplots(figsize=(6, 4), layout='constrained')
    colors = plt.get_cmap('tab20').colors

    # Use a separate color for each standardized polynomial feature
    for i in range(coefs.shape[1]):
        ax.plot(alphas, coefs[:, i], lw=1.0, c=colors[i], label=rf'$\beta_{{{i + 1}}}$')

    ax.set_xscale('log', base=10)
    ax.axhline(0.0, ls='--', lw=0.75, c='black')
    ax.set_xlabel(r'Regularization strength $\alpha$ (log scale)')
    ax.set_ylabel('Coefficient on standardized feature')
    title = f'{model_name} coefficients as a function of regularization strength'
    ax.set_title(title)
    _ = ax.legend(ncols=5)
```

```
[14]: plot_coefficient_path(alphas, coefs, 'Ridge')
```



The coefficients become smaller overall as α increases, although individual paths need not move steadily toward zero. At the right endpoint, the coefficients are substantially smaller but are not exactly zero. Conversely, as α approaches zero, Ridge approaches OLS.

Your turn

1. Find the largest absolute coefficient in the fitted OLS model `pipe_ols`.
Hint: The OLS coefficients are stored in the attribute `coef_` in the 'linearregression' step of the pipeline.
2. Find the largest absolute Ridge coefficient at the left endpoint of the path, where the penalty is $\alpha = 10^{-4}$. These coefficients are stored in `coefs[0]`.
3. Compare the two values. Does the smallest penalty in our path give coefficients close to OLS? What does this tell us about the shrinkage produced by a small positive penalty?

3.3.3 Tuning the regularization parameter via cross-validation

In the previous example, we picked an arbitrary regularization strength α . We now use cross-validation to select it. We search a broad range from 10^{-4} to 10^2 : a narrow range can miss a better minimum elsewhere in the validation curve (recall from lecture 2 that the validation curve traces the performance of the model as we vary the hyperparameter, here α).

Our preprocessing needs to be part of cross-validation. In each split, we estimate the means and standard deviations for both scalers using only the training observations. We then apply those same transformations to the validation observations. Scaling the entire sample before splitting would let information from the validation observations enter training. This is called *data leakage*.

Scikit-learn provides `RidgeCV()`, a class that selects the Ridge regularization strength α by cross-validation. However, it handles only the regression step. Our example also requires fitting the scalers separately within each training fold.

We therefore use `GridSearchCV()` around the complete pipeline. It tries each candidate α and refits both the preprocessing and the regression model within each training fold.

We define five shuffled folds with `KFold()` and fix the random seed so that we can reproduce the splits. We use the same folds for Lasso and Elastic Net later.

```
[15]: from sklearn.model_selection import GridSearchCV, KFold

# Use the same reproducible splits for all models
cv = KFold(n_splits=5, shuffle=True, random_state=1234)

# Scoring metric for cross-validation
scoring = 'neg_mean_squared_error'

# Candidate regularization strengths on a log scale from 1e-4 to 1e2
ridge_alphas = np.logspace(-4, 2, num=200)

# Keep all preprocessing inside the pipeline
pipe_ridge = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(degree=DEGREE, include_bias=False),
    StandardScaler(),
    Ridge(),
)
```

Once we have set up the CV splitter, the scoring metric, the candidate α values, and the Ridge pipeline, we create a `GridSearchCV` object and run the cross-validation:

```
[16]: # Define the grid search for Ridge regression with cross-validation
ridge_cv = GridSearchCV(
    pipe_ridge,
    param_grid={'ridge__alpha': ridge_alphas}, # parameters to search over
    scoring=scoring, # scoring metric for cross-validation
    cv=cv, # CV splitting strategy
    refit=True, # refit the best model on the entire training set
    n_jobs=4, # parallelize over 4 CPU cores
)

# Run the cross-validation
ridge_cv.fit(X_train, y_train)
```

The parameter name `ridge__alpha` combines the pipeline step name (`ridge`) and its parameter name (`alpha`), separated by two underscores. The search receives the training inputs `X_train` and target `y_train`, so each fold gets its own fitted input scaler and expanded-feature scaler.

Scikit-learn's search tools maximize scores. We specify `scoring='neg_mean_squared_error'` so that maximizing the score minimizes MSE. The selected parameter is stored in `best_params_`, and the corresponding average validation score is stored in `best_score_`.

```
[17]: # Recover the selected alpha of the best model
alpha_best = ridge_cv.best_params_['ridge__alpha']

# Average validation MSE of the best model
mse_ridge_cv = -ridge_cv.best_score_

print(f'Best alpha: {alpha_best:.4g}')
print(f'CV MSE: {mse_ridge_cv:.4g}')
```

```
Best alpha: 1.351
CV MSE: 0.2489
```

For our demo sample, the search selects $\alpha \approx 1.351$ with a CV MSE of about 0.249.

The search also stores results for every candidate in the dictionary `cv_results_`. We recover each α and its average validation score to plot the validation curve. These scores are averaged over the five folds.

We define `plot_validation_curve()` to handle these steps for both Ridge and Lasso. It takes the fitted search and the model name. Converting the name to lowercase gives the pipeline step name, so the function reads either `param_ridge__alpha` or `param_lasso__alpha`. Both searches use negative MSE, which we convert back to MSE for the plot.

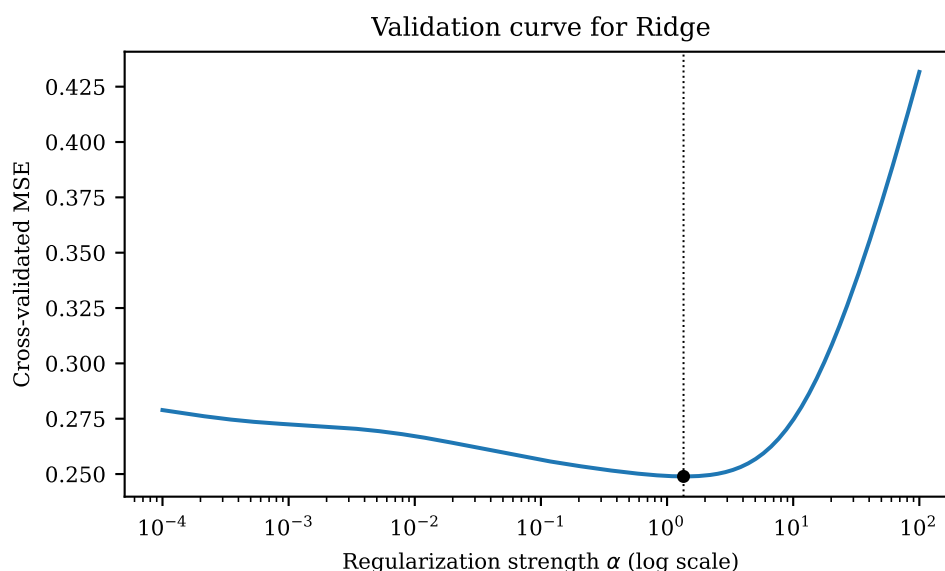
```
[18]: def plot_validation_curve(search, model_name):
    """
    Plot validation MSE against alpha and mark the selected value.

    Parameters
    -----
    search : object
        Fitted GridSearchCV search over the model's alpha
    model_name : str
        Model name, either 'Ridge' or 'Lasso'
    """
    import matplotlib.pyplot as plt

    # Recover candidate alphas, mean validation errors, and the selected index
    param_name = f'param_{model_name.lower()}__alpha'
    alphas = search.cv_results_[param_name]
    mse_mean = -search.cv_results_['mean_test_score']
    imin = search.best_index_

    # Plot the validation curve and highlight the selected alpha
    _, ax = plt.subplots(figsize=(5, 3), layout='constrained')
    ax.plot(alphas, mse_mean)
    ax.scatter(alphas[imin], mse_mean[imin], s=15, c='black', zorder=100)
    ax.axvline(alphas[imin], ls=':', lw=0.75, c='black')
    ax.set_xscale('log')
    ax.set_xlabel(r'Regularization strength  $\alpha$  (log scale)')
    ax.set_ylabel('Cross-validated MSE')
    _ = ax.set_title(f'Validation curve for {model_name}')
```

```
[19]: plot_validation_curve(ridge_cv, 'Ridge')
```



By default, `GridSearchCV` refits the selected pipeline on the entire training sample after choosing α . This fitted pipeline is stored in `best_estimator_`. We can use it directly to predict from the original x values; the pipeline applies the fitted transformations for us.

```
[20]: # Recover the pipeline already fitted with the selected alpha
pipe_ridge = ridge_cv.best_estimator_

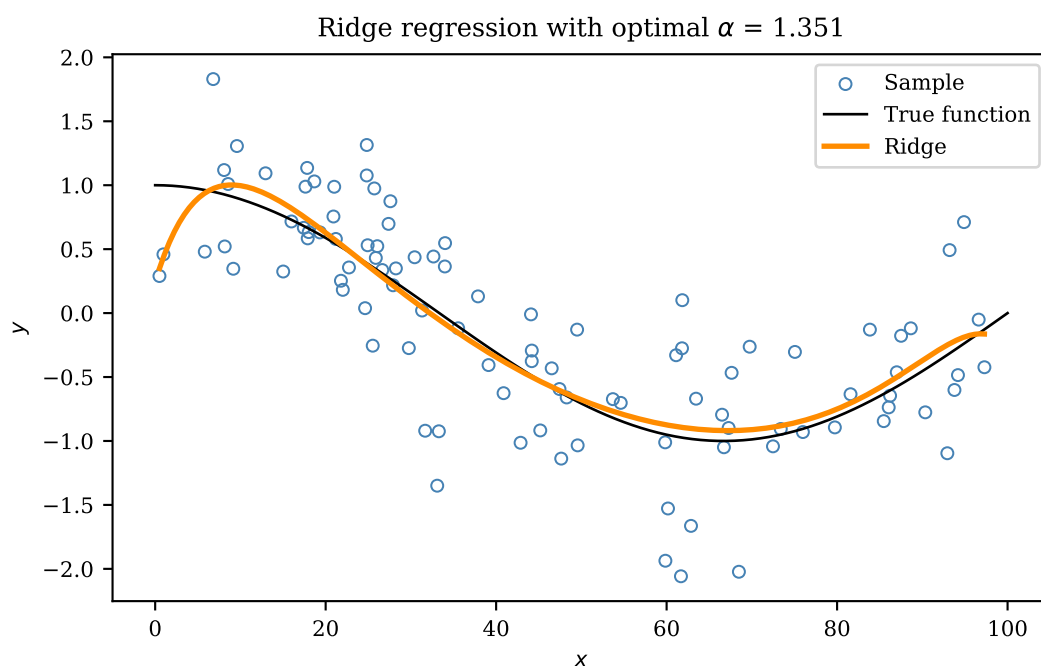
# Compute grid predictions using the selected Ridge model
yhat_grid_ridge = pipe_ridge.predict(X_grid)
```

Finally, we plot the predicted values together with the training sample and the true relationship:

```
[21]: # Plot sample and true relationship
ax = plot_trig_sample(x_train, y_train)

# Plot predicted values from cross-validated Ridge regression
ax.plot(xvalues, yhat_grid_ridge, c='darkorange', lw=2.0, label='Ridge')

ax.set_title(rf'Ridge regression with optimal $\alpha$ = {alpha_best:.4g}')
_ = ax.legend()
```



As the figure shows, the optimally selected Ridge model is smoother than the OLS model and follows the true relationship more closely, except at the left boundary of the training sample where we have few observations.

Your turn

1. Repeat the OLS and Ridge analysis with $N=200$. How do the predictions change?
2. What happens to the selected α ? Also compare α/N : dividing Ridge's objective by N gives mean squared error plus a penalty with weight α/N . This helps us compare penalty strengths across sample sizes.

Restore $N=100$ and rerun the analysis before continuing.

3.3.4 Dedicated cross-validation with RidgeCV

The dedicated `RidgeCV()` class is useful when the input features are already on suitable scales without learning preprocessing from the current sample. With its default `cv=None`, it uses an efficient calculation

of leave-one-out errors, avoiding a separate model fit for each omitted observation.

For a short illustration, we center and scale x by the known mean of 50 and upper bound of 100 before creating polynomial features. Knowing the data-generating process gives us a natural choice of fixed centering and scaling constants. In applications, suitable values may be less obvious. This fixed conversion uses no sample means or standard deviations, so it can be done before cross-validation. This illustration uses centered powers of $z = (x - 50)/100$ without either learned scaler. Both the polynomial basis and its scales differ from our main pipeline, so its fitted model and selected α are not directly comparable. For our main model with learned standardization, we keep the GridSearchCV approach.

We pass candidate values through alphas and recover the selected value from `alpha_`.

```
[22]: from sklearn.linear_model import RidgeCV

# Fixed feature construction; no scales are estimated from the sample
poly = PolynomialFeatures(degree=DEGREE, include_bias=False)
X_train_fixed = poly.fit_transform((X_train - 50) / 100)

# Candidate strengths for these fixed feature scales
ridge_alphas_dedicated = np.logspace(-5, 2, num=200)

# Create a RidgeCV object with candidate alphas
ridge_cv_dedicated = RidgeCV(alphas=ridge_alphas_dedicated)

# Run the cross-validation
ridge_cv_dedicated.fit(X_train_fixed, y_train)

print(f'RidgeCV selected alpha: {ridge_cv_dedicated.alpha_:.4g}')
```

RidgeCV selected alpha: 0.0003002

The selected α is *not directly comparable* to the earlier pipeline value because the polynomial basis and feature scales differ.

3.4 Lasso

Another widely used estimator with shrinkage is Lasso (least absolute shrinkage and selection operator). Its objective combines a scaled squared-error term with an L1 penalty,

$$L(\mu, \beta) = \underbrace{\frac{1}{2N} \sum_{i=1}^N (y_i - \mu - \mathbf{x}_i' \beta)^2}_{\text{Half the mean squared error}} + \underbrace{\alpha \sum_{k=1}^K |\beta_k|}_{\text{L1 penalty}}$$

where we have used the formulation implemented in scikit-learn. You might see slightly different but equivalent formulations in other textbooks. As with Ridge regression, the additional term penalizes large coefficient values, while the intercept is not penalized. This term is called an L1 penalty because it corresponds to the L1 vector norm, which we can also write as $\alpha \|\beta\|_1$.

Using the L1 instead of the squared L2 norm can produce much more parsimonious models: many coefficients can be exactly zero, eliminating the corresponding features from the model. We will see this in the example below.

3.4.1 Estimating the Lasso and linear regression models

We apply Lasso to the same training sample, `X_train` and `y_train`, that we used for Ridge. This allows us to compare the two methods.

As with Ridge, we standardize the input before creating polynomial features, then standardize the expanded columns before fitting Lasso. We use the `Lasso` class to perform the model estimation. For

now, we set the regularization strength to $\alpha = 0.0075$ and use cross-validation to find the optimal value later.

For Lasso, we may need to increase `max_iter` from its default of 1,000. We also set `tol=1e-6`, a smaller stopping tolerance than the default of `1e-4`, to obtain more accurate solutions with our correlated polynomial features.

```
[23]: from sklearn.linear_model import Lasso
      from sklearn.pipeline import make_pipeline
      from sklearn.preprocessing import PolynomialFeatures, StandardScaler

      # Penalty strength to use for Lasso
      alpha = 0.0075

      # Build pipeline of transformations and Lasso estimation.
      # We create the polynomial transformation w/o the intercept so we
      # need to include an intercept in the fitting step.
      pipe_lasso = make_pipeline(
          StandardScaler(),
          PolynomialFeatures(degree=DEGREE, include_bias=False),
          StandardScaler(),
          Lasso(alpha=alpha, fit_intercept=True, max_iter=10_000, tol=1e-6),
      )

      pipe_lasso.fit(X_train, y_train)
```

For comparison, we reuse the fitted OLS pipeline `pipe_ols` and its grid predictions `yhat_grid_ols` from the Ridge section.

The following code plots the sample data, the true functional relationship, and the predictions from the linear regression and Lasso models.

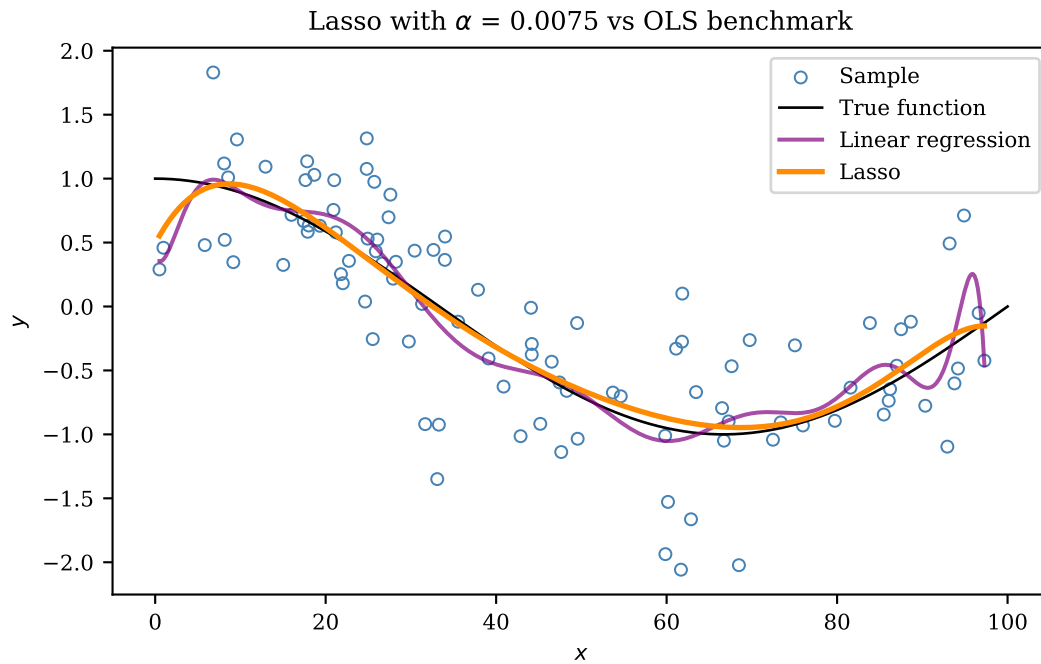
```
[24]: # Grid predictions from the illustrative Lasso model
      yhat_grid_lasso = pipe_lasso.predict(X_grid)

[25]: ax = plot_trig_sample(x_train, y_train)

      # Linear regression prediction
      ax.plot(xvalues, yhat_grid_ols, c='purple', alpha=0.7, label='Linear regression')

      # Lasso prediction
      ax.plot(xvalues, yhat_grid_lasso, c='darkorange', lw=2.0, label='Lasso')

      ax.set_title(rf'Lasso with $\alpha$ = {alpha:.4g} vs OLS benchmark')
      _ = ax.legend()
```



We chose $\alpha = 0.0075$ for this illustration, whereas we initially used $\alpha = 3$ for Ridge regression. These values are not directly comparable: scikit-learn divides the squared-error term by $2N$ for Lasso, and the two methods also use different penalties. There is no general conversion that gives equivalent shrinkage for Ridge and Lasso. We therefore tune α separately for each model.

Your turn

Set $\alpha=1$ in the Lasso setup and rerun the fit and prediction plot. How does the fitted curve change? Explain the intuition behind this result.

3.4.2 Lasso path

As for Ridge regression, we can visualize how the estimated Lasso coefficients change as we vary the penalty parameter. This plot is called the *Lasso path*.

To compute the path, we fit the model for different values of α on the interval $[10^{-4}, 1]$, where we again space α uniformly on a logarithmic scale. This range includes smaller penalties that can retain more polynomial features.

We reuse the Lasso pipeline defined above. As in the Ridge path, we update `lasso__alpha` with `set_params()`, refit the complete pipeline on `X_train`, and recover the coefficients from `named_steps['lasso']`. The preprocessing steps are refitted for each value of α , using the same training sample.

```
[26]: # Create grid of alphas spaced uniformly in logs on [1e-4, 1]
alphas = np.logspace(start=np.log10(1.0e-4), stop=np.log10(1.0), num=200)

# Allow more iterations and reuse intermediate calculations within each fit
pipe_lasso.set_params(lasso__max_iter=20_000_000, lasso__precompute=True)

# Array to store one coefficient for each polynomial feature and alpha
coefs = np.empty((len(alphas), DEGREE))

# Reuse the Lasso pipeline, updating alpha before each fit
for i, alpha in enumerate(alphas):
    # Update the penalty in the Lasso step
```

3 Regression models with scikit-learn

```
pipe_lasso.set_params(lasso__alpha=alpha)

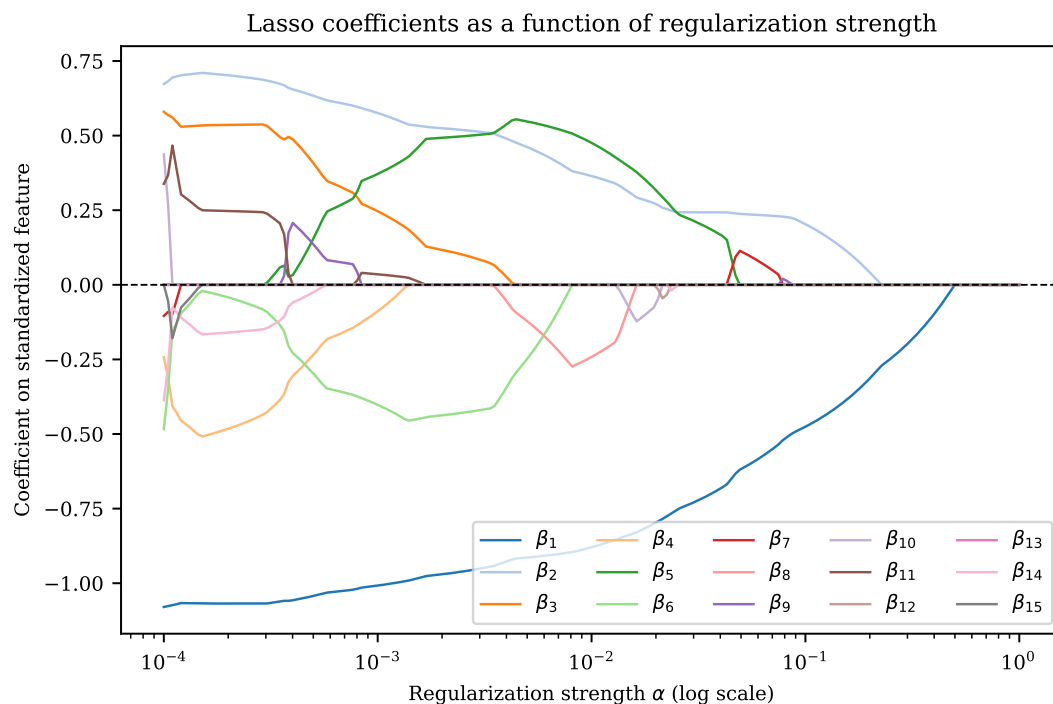
# Refit the pipeline on the original input
pipe_lasso.fit(X_train, y_train)

# Store the coefficients for this alpha
coefs[i] = pipe_lasso.named_steps['lasso'].coef_
```

We reuse `plot_coefficient_path()` to plot the fitted Lasso coefficients against α on a log scale:

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

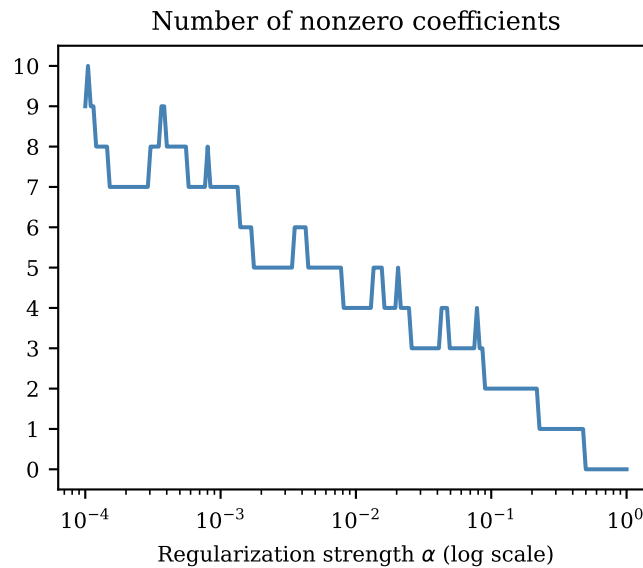
plot_coefficient_path(alphas, coefs, 'Lasso')
```



The coefficient paths differ from those obtained with Ridge. For sufficiently large α , many Lasso coefficients are exactly zero. We highlight this result in the graph below, which plots the number of nonzero coefficients against α .

```
[28]: # Treat very small coefficient values as zero
nonzero = np.sum(np.abs(coefs) > 1.0e-6, axis=1)

[29]: # Plot number of nonzero coefficients against alpha
plt.plot(alphas, nonzero, lw=1.5, c='steelblue')
plt.xscale('log', base=10)
plt.yticks(np.arange(0, np.amax(nonzero) + 1))
plt.xlabel(r'Regularization strength  $\alpha$  (log scale)')
_ = plt.title('Number of nonzero coefficients')
```



At small values of α , Lasso retains more of the 15 polynomial features. As the penalty increases, we generally obtain models with fewer nonzero coefficients, eventually leaving only the intercept.

It's important to stress that we fit Lasso on standardized polynomial features. Our pipeline first centers and scales x to obtain z , then creates $z, z^2, \dots, z^{15}$. The second scaler standardizes each of these polynomial columns before fitting Lasso, putting them on comparable scales for the penalty. A zero coefficient means that Lasso leaves out one of these *transformed* features. It does not necessarily mean that Lasso leaves out the corresponding power of the original input x .

We can see why centering matters with a simple example. We temporarily ignore scaling and write $z = x - 5$. Suppose our fitted model retains only the z^2 feature. Rewriting this feature gives

$$z^2 = (x - 5)^2 = x^2 - 10x + 25.$$

The same fitted curve therefore contains both an x^2 term and an x term when written in terms of the original input, with the constant absorbed into the intercept. Standardizing the z^2 column changes its scale and shifts its constant term, but it still involves both powers of x .

The practical implication is that the number of nonzero coefficients describes how many features we retain from our chosen polynomial expansion. We cannot interpret a zero coefficient on the standardized z feature as evidence that the fitted curve has no x term.

Changing the center before fitting can also change the fitted curve: it changes the polynomial features and the coefficients that Lasso penalizes. Standardizing the columns makes their scales comparable, but does not make these different expansions equivalent under the Lasso penalty. Our pipeline is appropriate for this prediction example; we use cross-validation to select its penalty strength. If we also want to choose between different centering or feature-construction methods, we should compare those choices using cross-validation as well.

We can also compute the Lasso path using the convenience function `lasso_path()`. This function takes a feature matrix directly, so we prepare the standardized polynomial features ourselves. It does not fit an intercept, so we also subtract the mean of y . Here we are illustrating fitted coefficients on one sample, rather than estimating validation performance.

```
[30]: from sklearn.linear_model import lasso_path

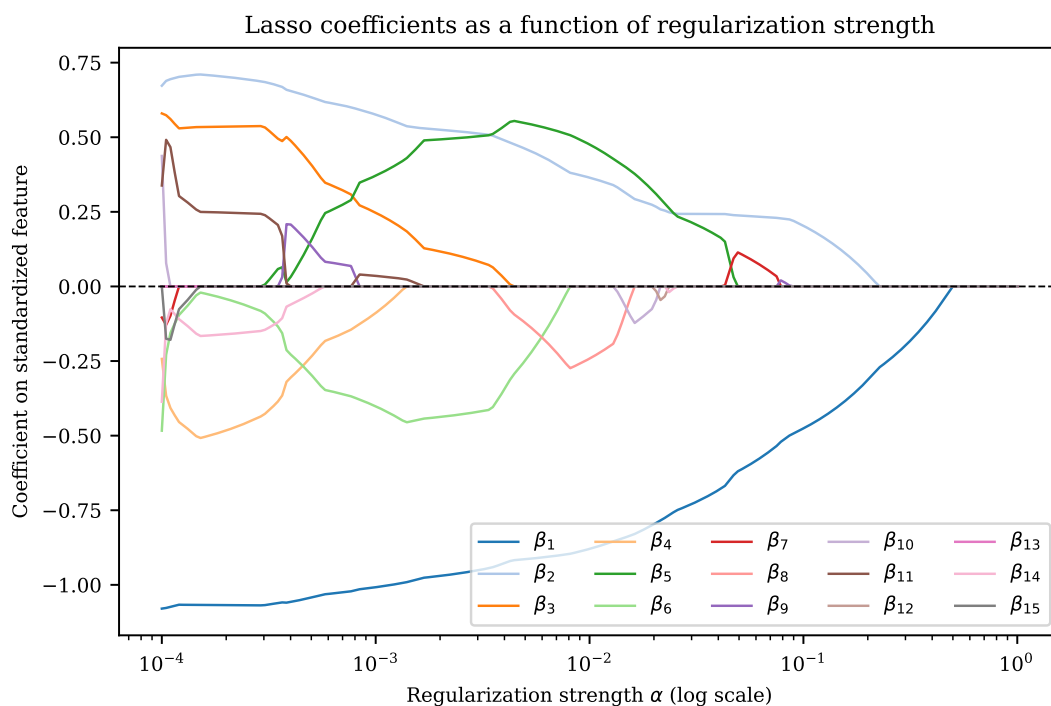
# Re-create pipeline w/o Lasso estimator to be used with lasso_path()
transform = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(degree=DEGREE, include_bias=False),
    StandardScaler(),
)
```

```
# Create polynomial features
X_train_trans = transform.fit_transform(X_train)

# Center the training target because lasso_path does not fit an intercept
y_train_centered = y_train - np.mean(y_train)

# Compute Lasso path automatically
alphas, coefs, _ = lasso_path(
    X_train_trans, y_train_centered, alphas=alphas, max_iter=20_000_000, tol=1e-6
)
```

```
[31]: # Transpose so each row contains the coefficients for one alpha
plot_coefficient_path(alphas, coefs.T, 'Lasso')
```



The two paths can differ because the numerical algorithm stops once it reaches a specified tolerance. Our loop fits each model from scratch, while `lasso_path()` starts each fit from the solution for a nearby α . Reusing a previous solution is called a *warm start* and often saves computation. With highly correlated polynomial features, these choices can produce visible differences in coefficients. More iterations and a smaller `tol` can improve agreement, at the cost of more computation.

3.4.3 Tuning the regularization parameter via cross-validation

We now select α for Lasso using `GridSearchCV()`. As in the Ridge example, the search wraps the complete pipeline so that both scalers are learned separately within each training fold. Scikit-learn also provides a dedicated `LassoCV()` class that selects α by cross-validation. However, it does not refit scalers within its internal folds, and thus may suffer from the same data-leakage issue as `RidgeCV`. We therefore use `GridSearchCV` around the complete pipeline here and discuss the `LassoCV` alternative below.

We search from 10^{-4} to 10^{-1} using the same five folds and negative MSE scoring rule as before. Including smaller penalties lets us compare sparse models with models that retain more of the expansion.

We allow more iterations for the highly correlated polynomial features and keep `tol=1e-6`. Setting `precompute=True` computes a matrix of feature cross-products once per fit and reuses it during the solver's iterations. Each fit performed by `GridSearchCV` computes its own matrix.

```
[32]: # Candidate regularization strengths for Lasso
lasso_alphas = np.logspace(-4, -1, num=200)

# Keep all preprocessing inside the pipeline
pipe_lasso = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(degree=DEGREE, include_bias=False),
    StandardScaler(),
    Lasso(max_iter=20_000_000, tol=1e-6, precompute=True),
)

# Define the grid search for Lasso regression with cross-validation
lasso_cv = GridSearchCV(
    pipe_lasso,
    param_grid={'lasso__alpha': lasso_alphas}, # parameters to search over
    scoring=scoring, # scoring metric for cross-validation
    cv=cv, # CV splitting strategy
    refit=True, # refit the best model on the entire training set
    n_jobs=4, # parallelize over 4 CPU cores
)

# Run the cross-validation
lasso_cv.fit(X_train, y_train)
```

We recover the selected α from `best_params_` and the validation scores from `cv_results_`, just as we did for Ridge.

```
[33]: # Optimal alpha of the best model
alpha_best = lasso_cv.best_params_['lasso__alpha']

# Average validation MSE of the best model
mse_lasso_cv = -lasso_cv.best_score_

print(f'Best alpha: {alpha_best:.4g}')
print(f'CV MSE: {mse_lasso_cv:.4g}')
```

```
Best alpha: 0.005053
CV MSE: 0.2487
```

The selected pipeline is stored in `best_estimator_`. We access its fitted Lasso step through `named_steps['lasso']` and count the nonzero coefficients. As in the path plot, we treat absolute values at or below 10^{-6} as zero.

```
[34]: # Reference to best Lasso estimator
lasso_best = lasso_cv.best_estimator_.named_steps['lasso']

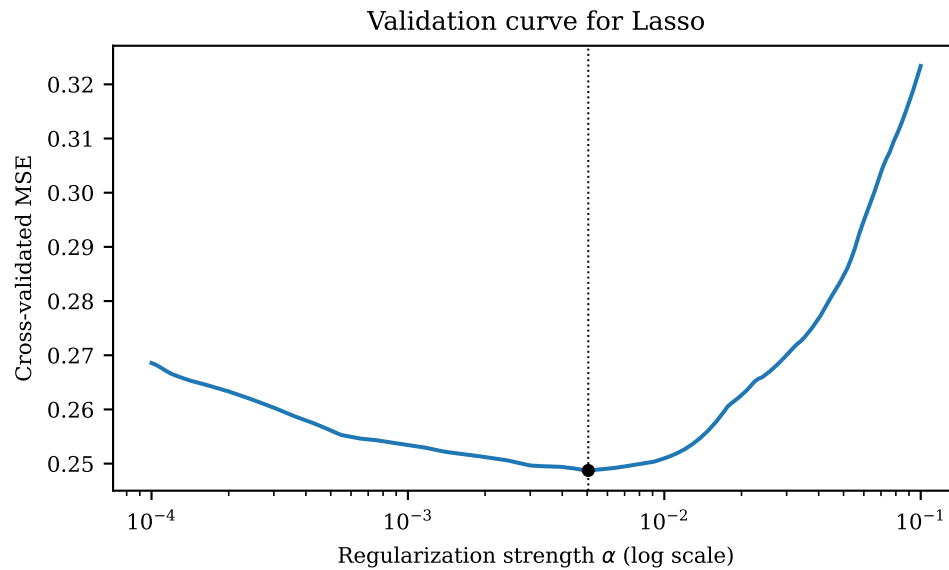
# Count the number of nonzero coefficients in the best Lasso model
n_features = np.sum(np.abs(lasso_best.coef_) > 1.0e-6)
print(f'Number of nonzero coefficients: {n_features}')
```

```
Number of nonzero coefficients: 5
```

For our demo sample, cross-validation selects $\alpha \approx 0.00505$ with a CV MSE of about 0.249. The refitted model retains 5 of the 15 polynomial features. Thus, Lasso reduces the number of features, although its sparsity depends on the input transformation and the penalty chosen by cross-validation.

We reuse `plot_validation_curve()` with the fitted Lasso search to show average validation MSE across the candidate values of α . If a selected value is at an endpoint of the grid, we would expand the grid to check whether values beyond it perform better.

```
[35]: plot_validation_curve(lasso_cv, 'Lasso')
```



The search has already refitted the selected pipeline on the full training sample. We use this best model to predict `yhat_grid_lasso` on the prediction grid.

```
[36]: # Recover the pipeline already fitted with the selected alpha
pipe_lasso = lasso_cv.best_estimator_

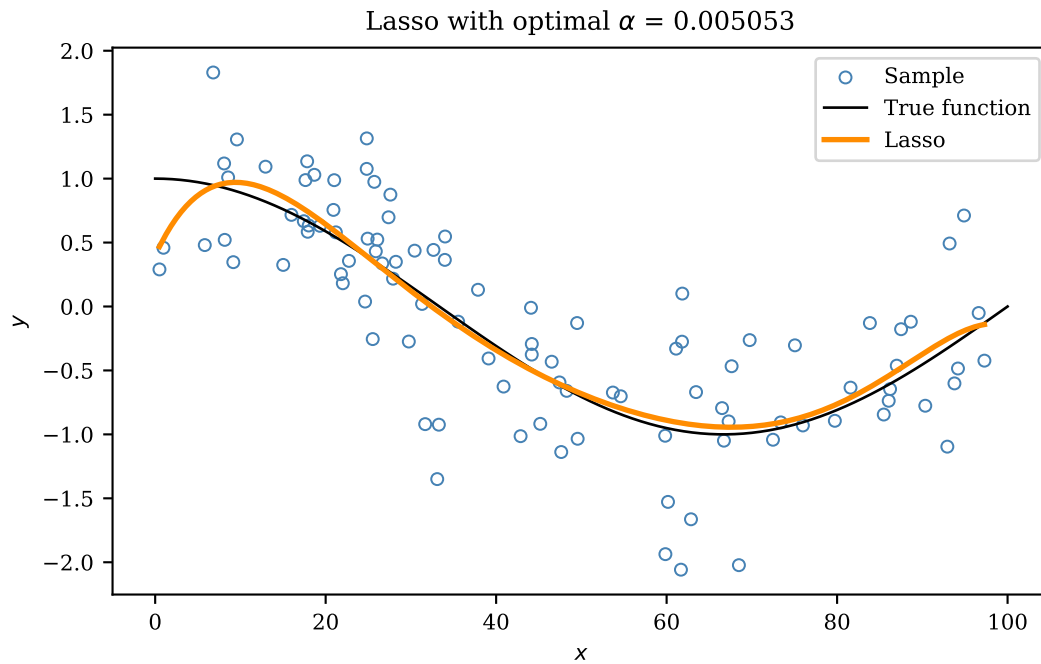
# Compute grid predictions using the selected Lasso model
yhat_grid_lasso = pipe_lasso.predict(X_grid)
```

Finally, we visually compare the true model to the Lasso prediction using the optimal value of α .

```
[37]: # Plot sample and true relationship
ax = plot_trig_sample(x_train, y_train)

# Plot prediction from optimal Lasso model
ax.plot(xvalues, yhat_grid_lasso, c='darkorange', lw=2.0, label=r'Lasso')

ax.set_title(rf'Lasso with optimal  $\alpha = \{\alpha\_best:.4g\}$ ')
_ = ax.legend()
```



Your turn

Perform the following tasks using the fitted Lasso model `pipe_lasso`:

- Compute the training MSE using the predicted values `pipe_lasso.predict(X_train)` and `y_train`.
Hint: You can use `mean_squared_error()` from `sklearn.metrics` to compute the MSE.
- Compare the training MSE with the CV MSE stored in `mse_lasso_cv`. Which is lower for this sample? What could explain the difference?

3.4.4 Dedicated cross-validation with **LassoCV**

The dedicated `LassoCV()` class is useful when no preprocessing needs to be learned within the folds. It can be faster than a general grid search because it reuses the coefficients from a nearby α as a starting point for the next fit. We encountered this idea of warm starts in the Lasso path example.

As in the `RidgeCV` illustration, we center and scale x using its known mean of 50 and upper bound of 100, then construct polynomial features from $z = (x - 50)/100$. We do not fit a scaler on the full sample. This illustration uses centered powers of z without either learned scaler. Its polynomial basis and feature scales differ from our main example; we use this calculation only to illustrate the dedicated class.

We supply the candidate grid and folds through `alphas` and `cv`. The class selects α using validation MSE and stores it in `alpha_`.

```
[38]: from sklearn.linear_model import LassoCV

# Fixed feature construction; no scales are estimated from the sample
poly = PolynomialFeatures(degree=DEGREE, include_bias=False)
X_train_fixed = poly.fit_transform((X_train - 50) / 100)

# Candidate strengths for these fixed feature scales
lasso_alphas_dedicated = np.logspace(-5, -2, num=200)

# Create a LassoCV object with candidate alphas
```

```
lasso_cv_dedicated = LassoCV(alphas=lasso_alphas_dedicated, cv=cv, max_iter=1_000_000)

# Run the cross-validation
lasso_cv_dedicated.fit(X_train_fixed, y_train)

print(f'LassoCV selected alpha: {lasso_cv_dedicated.alpha_:.4g}')
```

LassoCV selected alpha: 7.233e-05

We obtain a much smaller $\alpha \approx 7.2 \times 10^{-5}$ than in the standardized pipeline ($\alpha \approx 0.0051$), but these values are not directly comparable because the polynomial basis and feature scales differ.

3.5 Elastic Net

Elastic Net is a regularized linear regression model that combines the L1 and L2 penalties of Lasso and Ridge. Its objective, using the scikit-learn formulation, is:

$$L(\mu, \beta) = \underbrace{\frac{1}{2N} \sum_{i=1}^N (y_i - \mu - \mathbf{x}_i' \beta)^2}_{\text{Half the mean squared error}} + \underbrace{\alpha \rho \sum_{k=1}^K |\beta_k|}_{\text{L1 penalty}} + \underbrace{\frac{\alpha(1-\rho)}{2} \sum_{k=1}^K \beta_k^2}_{\text{L2 penalty}}$$

As before, the intercept is not penalized.

By incorporating both penalties, Elastic Net can produce a sparse model with many zero coefficients while also using Ridge's squared-coefficient penalty. The hyperparameter $\rho \in [0, 1]$, called `l1_ratio` in scikit-learn, controls the mix of the two penalties:

- When $\rho = 1$, the model is equivalent to Lasso with the same α .
- When $\rho = 0$, only the L2 penalty remains. This gives a Ridge model, but the different scaling of the objective means that the equivalent Ridge parameter is $\alpha_{\text{Ridge}} = N \cdot \alpha_{\text{Elastic Net}}$.

3.5.1 Estimating the Elastic Net and linear regression models

We use the same training sample, `X_train` and `y_train`, from the previous sections to fit the Elastic Net model.

We first illustrate the estimation for a fixed set of hyperparameters: `alpha = 0.01` and `l1_ratio = 0.5` (an equal mix of L1 and L2 penalties). For comparison, we reuse the OLS pipeline `pipe_ols` from the earlier `LinearRegression()` fit. It already uses the same training sample and polynomial features.

As with Lasso earlier, it may be necessary to increase `max_iter` to ensure convergence.

```
[39]: from sklearn.linear_model import ElasticNet
      from sklearn.pipeline import make_pipeline
      from sklearn.preprocessing import PolynomialFeatures, StandardScaler

      # Build pipeline for Elastic Net
      pipe_enet = make_pipeline(
          StandardScaler(),
          PolynomialFeatures(degree=DEGREE, include_bias=False),
          StandardScaler(),
          ElasticNet(
              alpha=0.01, l1_ratio=0.5, fit_intercept=True, max_iter=100_000, tol=1e-6
          ),
      )

      # Fit Elastic Net
      pipe_enet.fit(X_train, y_train)
```

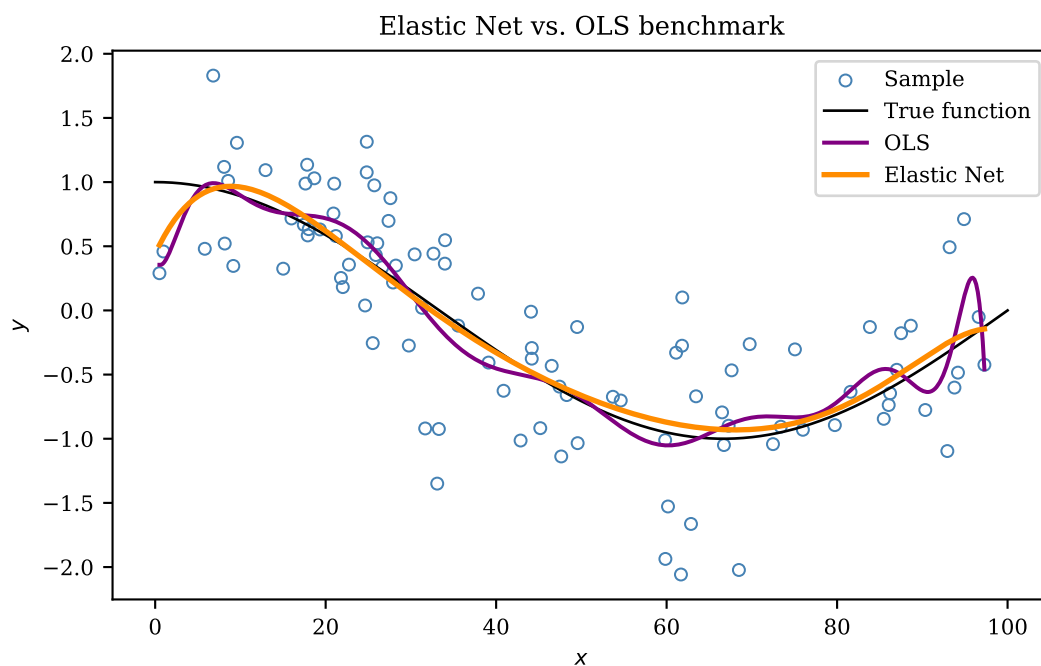
We now compare the predicted values from both models against the true underlying function.

```
[40]: # Grid predictions from the illustrative Elastic Net model
yhat_grid_enet = pipe_enet.predict(X_grid)

# Plot sample and true relationship
ax = plot_trig_sample(x_train, y_train)

# Plot predictions
ax.plot(xvalues, yhat_grid_ols, c='purple', lw=1.5, ls='-', label='OLS')
ax.plot(xvalues, yhat_grid_enet, c='darkorange', lw=2.0, label='Elastic Net')

ax.set_title(r'Elastic Net vs. OLS benchmark')
_ = ax.legend()
```



3.5.2 Tuning the regularization parameters via cross-validation

Unlike Ridge and Lasso, Elastic Net has two hyperparameters to tune: α and the L1 ratio ρ . We use `GridSearchCV()` to try every combination of 200 candidate values of α and six L1 ratios. We search from 10^{-4} to 10^{-1} for α and include small L1 ratios to explore solutions close to Ridge.

The search again wraps the entire pipeline so that both scalers are fitted within each training fold. The dedicated `ElasticNetCV()` class has the same data leakage issue if we preprocess the full sample before its internal splits. We keep the same five folds and scoring rule as in the other examples.

```
[41]: # Candidate values for both hyperparameters
l1_ratios = [0.001, 0.01, 0.1, 0.5, 0.9, 0.95]
enet_alphas = np.logspace(-4, -1, num=200)

# Keep all preprocessing inside the pipeline
pipe_enet = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(degree=DEGREE, include_bias=False),
    StandardScaler(),
    ElasticNet(max_iter=20_000_000, tol=1e-6, precompute=True),
)
```

```

# Define the parameter grid for the Elastic Net search
param_grid = {
    'elasticnet__alpha': enet_alphas,
    'elasticnet__l1_ratio': l1_ratios,
}

# Define the grid search for Elastic Net regression with cross-validation
enet_cv = GridSearchCV(
    pipe_enet,
    param_grid=param_grid, # parameters to search over
    scoring=scoring, # scoring metric for cross-validation
    cv=cv, # CV splitting strategy
    refit=True, # refit the best model on the entire training set
    n_jobs=4, # parallelize over 4 CPU cores
)

# Run the cross-validation
enet_cv.fit(X_train, y_train)

```

We allow more iterations than the default of 1,000 and set `tol=1e-6` because these polynomial features are highly correlated. Setting `precompute=True` also speeds up fitting by reusing intermediate calculations within each fit. If fitting produces a `ConvergenceWarning`, we need to check the affected fits before relying on the search results. A larger `max_iter` allows more work but does not guarantee convergence.

Once the cross-validation is completed, we can access the selected hyperparameters through `best_params_` and the average validation score through `best_score_`:

```

[42]: # Optimal alpha of the best model
alpha_best = enet_cv.best_params_['elasticnet__alpha']

# Optimal L1 ratio of the best model
l1_ratio_best = enet_cv.best_params_['elasticnet__l1_ratio']

# Average validation MSE of the best model
mse_enet_cv = -enet_cv.best_score_

print(f'Best alpha: {alpha_best:.4g}')
print(f'Best L1 ratio: {l1_ratio_best:.4g}')
print(f'CV MSE: {mse_enet_cv:.4g}')

```

```

Best alpha: 0.01534
Best L1 ratio: 0.1
CV MSE: 0.2483

```

For our demo sample, the search selects $\alpha \approx 0.0153$ and $\rho = 0.1$, with a CV MSE of about 0.248.

A small selected L1 ratio puts most weight on the L2 part of the penalty. Such an Elastic Net model can retain many or all of the features, much like Ridge. A ratio close to one favors Lasso's L1 penalty and can produce a sparser model. We therefore cannot assume that Elastic Net will select only a few features in every sample.

The search has already refitted the selected pipeline on the full training sample. We use this best model to predict `yhat_grid_enet` on the prediction grid.

```

[43]: # Recover the pipeline already fitted with the selected hyperparameters
pipe_enet = enet_cv.best_estimator_

# Compute grid predictions using the selected Elastic Net model
yhat_grid_enet = pipe_enet.predict(X_grid)

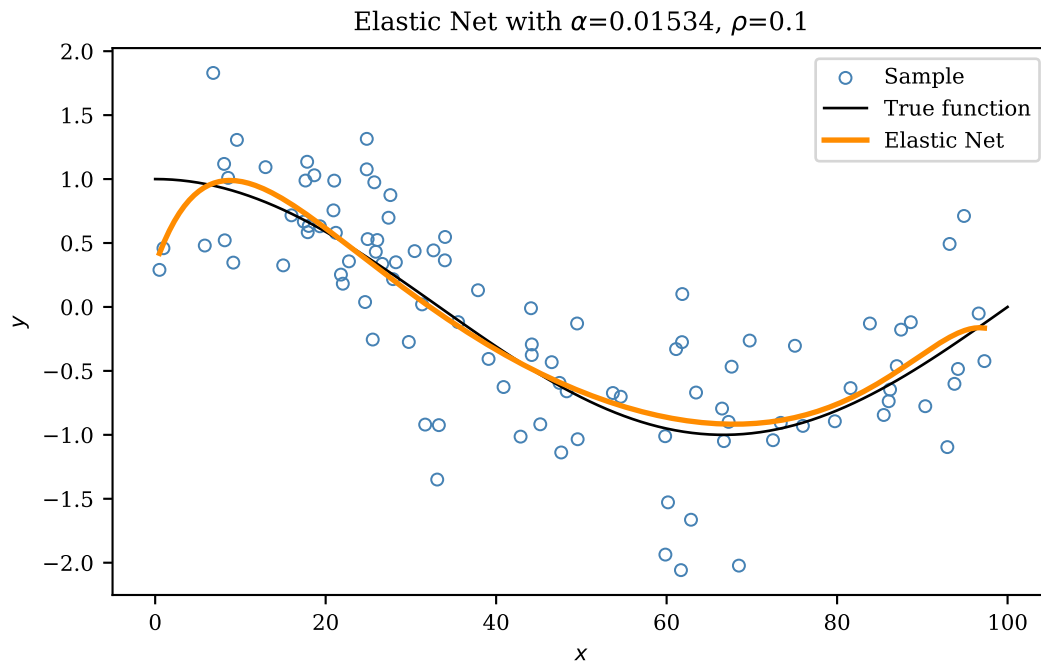
```

Finally, we visually compare the true model to the Elastic Net prediction using the optimal values of α and ρ .

```
[44]: # Plot sample and true relationship
ax = plot_trig_sample(x_train, y_train)

# Plot prediction from optimal Elastic Net model
ax.plot(xvalues, yhat_grid_enet, c='darkorange', lw=2.0, label='Elastic Net')

title = rf'Elastic Net with  $\alpha$ ={alpha_best:.4g},  $\rho$ ={l1_ratio_best:.4g}'
ax.set_title(title)
_ = ax.legend()
```



Your turn

Using the code from above, perform the following tasks:

- Count the nonzero coefficients in the selected Elastic Net model, using the same 10^{-6} threshold as for Lasso. How many additional features does Elastic Net retain?
- Compare the CV MSEs of the two models. Is the improvement from Elastic Net substantial?

Hint: Use `enet_cv.best_estimator_.named_steps['elasticnet'].coef_` for the Elastic Net coefficients. For Lasso, reuse `n_features`. The CV MSEs are `mse_enet_cv` and `mse_lasso_cv`.

3.5.3 Dedicated cross-validation with ElasticNetCV

The dedicated `ElasticNetCV()` class searches over both α and the L1 ratio. Like `LassoCV`, it reuses solutions along the regularization path, which can make tuning faster than fitting each candidate independently. It is useful when the input features require no preprocessing learned within each fold. It does not solve the leakage problem for a scaler placed before it in a pipeline.

As in the `LassoCV` illustration, we use fixed, centered powers of $z = (x - 50)/100$ and no learned scalers. These features differ from those in our main pipeline. We recover the selected parameters from `alpha_` and `l1_ratio_`.

```
[45]: from sklearn.linear_model import ElasticNetCV
```

```

# Fixed feature construction; no scales are estimated from the sample
poly = PolynomialFeatures(degree=DEGREE, include_bias=False)
X_train_fixed = poly.fit_transform((X_train - 50) / 100)

# Candidate strengths for these fixed feature scales
enet_alphas_dedicated = np.logspace(-6, -2, num=200)

# Candidate L1 ratios for Elastic Net
l1_ratios_dedicated = [1.0e-6, 0.001, 0.01, 0.1, 0.5, 0.9, 0.95]

# Create an ElasticNetCV object with candidate alphas and L1 ratios
enet_cv_dedicated = ElasticNetCV(
    alphas=enet_alphas_dedicated,
    l1_ratio=l1_ratios_dedicated,
    cv=cv,
    max_iter=5_000_000,
    tol=1e-6,
    precompute=True,
    n_jobs=4,
)

# Run the cross-validation
enet_cv_dedicated.fit(X_train_fixed, y_train)

print(f'ElasticNetCV selected alpha: {enet_cv_dedicated.alpha_:.4g}')
print(f'ElasticNetCV selected L1 ratio: {enet_cv_dedicated.l1_ratio_:.4g}')

```

```

ElasticNetCV selected alpha: 8.407e-06
ElasticNetCV selected L1 ratio: 1e-06

```

We are fitting substantially different regularized models because the polynomial basis and feature scales change the meaning of the penalty, so the selected α and L1 ratio are not directly comparable to those from our standardized pipeline.

The selected L1 ratio is close to zero, suggesting that cross-validation favors an almost pure Ridge penalty for these features; repeatedly lowering the grid boundary adds little practical information.

3.6 Comparing predictions on new observations

The minimum CV MSE helped us choose hyperparameters. To assess the resulting models, we now generate a large test sample of 100,000 observations that played no part in fitting or tuning. A new random generator gives us independent observations from the same model. The large sample makes the comparison less sensitive to chance variation in the test observations. We use the fitted pipelines without refitting or tuning them on this test sample.

We compute prediction errors with `mean_squared_error()` and compare predictions with the noisy test outcomes `y_test`. Even the true cosine function would have an expected MSE of $0.5^2 = 0.25$ because of the measurement error. This gives us a useful benchmark when interpreting the results.

```

[46]: from sklearn.metrics import mean_squared_error

# Generate a large independent test sample
x_test, y_test = create_trig_sample(N=100_000, rng=default_rng(5678))
X_test = x_test[:, None]

# Predictions from OLS and the selected regularized models
yhat_test_ols = pipe_ols.predict(X_test)
yhat_test_ridge = pipe_ridge.predict(X_test)
yhat_test_lasso = pipe_lasso.predict(X_test)
yhat_test_enet = pipe_enet.predict(X_test)

```

```
# Compute the OLS benchmark for the relative comparison
mse_ols_test = mean_squared_error(y_test, yhat_test_ols)
```

The following code reports the results. The table includes the CV MSE from each regularized model's pipeline search; we show a dash for OLS because we did not cross-validate it. For each model, we also report the percentage reduction in test MSE relative to OLS. A positive reduction means better prediction than OLS. This comparison evaluates the models fitted on our particular training sample; a different training sample could change the results.

```
[47]: predictions = [
    ('OLS', yhat_test_ols, None),
    ('Ridge', yhat_test_ridge, mse_ridge_cv),
    ('Lasso', yhat_test_lasso, mse_lasso_cv),
    ('Elastic Net', yhat_test_enet, mse_enet_cv),
]

print(f'{"Model":<12} {"CV MSE":>8} {"Test MSE":>10} {"Reduction vs OLS":>17}')
for name, yhat_test, mse_cv in predictions:
    mse_test = mean_squared_error(y_test, yhat_test)
    reduction = (mse_ols_test - mse_test) / mse_ols_test

    # OLS was not cross-validated in this example
    cv_text = '-' if mse_cv is None else f'{mse_cv:.5f}'

    print(f'{name:<12} {cv_text:>8} {mse_test:10.5f} {reduction:17.2%}')
```

Model	CV MSE	Test MSE	Reduction vs OLS
OLS	-	1.32244	0.00%
Ridge	0.24892	0.26217	80.17%
Lasso	0.24874	0.25870	80.44%
Elastic Net	0.24826	0.26058	80.30%

The test MSE is about 1.322 for OLS, 0.262 for Ridge, 0.259 for Lasso, and 0.261 for Elastic Net. All three regularized models reduce test MSE by about 80% relative to OLS, with errors close to the noise benchmark of 0.25. The large OLS error is mostly driven by predictions near the upper end of $[0, 100]$, beyond the largest training value of about 97.3. Our prediction plots stop at the largest training value, so they do not show these extreme predictions.

These results illustrate the benefits of regularization for our fitted degree-15 polynomial; they do not establish that one method always predicts better than the others. Elastic Net achieves the lowest CV MSE but a slightly higher test MSE than Lasso, while all three regularized models perform similarly on this sample.

3.7 List of functions used in this lecture

The following list contains the central functions covered in this lecture. Each function name is linked to the official API documentation which you can consult for more details regarding function arguments and return values. The `[[go to section]]` links take us to the relevant sections in this notebook.

3.7.1 NumPy array functions

- `logspace()`: Function for creating numbers spaced evenly on a log scale. [\[go to section\]](#)

3.7.2 scikit-learn model selection

- `GridSearchCV()`: Cross-validation over a parameter grid for a complete pipeline. [\[go to section\]](#)
- `KFold()`: Split observations into training and validation folds. [\[go to section\]](#)

3.7.3 scikit-learn metrics

- `mean_squared_error()`: Average squared prediction error. [\[go to section\]](#)

3.7.4 scikit-learn linear models

- `Ridge()`: Linear least squares with L2 regularization. [\[go to section\]](#)
- `RidgeCV()`: Ridge regression with built-in cross-validation. [\[go to section\]](#)
- `Lasso()`: Linear model with L1 regularization (shrinkage). [\[go to section\]](#)
- `lasso_path()`: Function for computing the Lasso path with coordinate descent. [\[go to section\]](#)
- `LassoCV()`: Lasso linear model with iterative fitting along a regularization path. [\[go to section\]](#)
- `ElasticNet()`: Linear regression with combined L1 and L2 priors as regularizer. [\[go to section\]](#)
- `ElasticNetCV()`: Elastic Net model with iterative fitting along a regularization path. [\[go to section\]](#)

4 Classification models with Scikit-learn

4.1 Motivation

In the previous lecture, we studied how to predict continuous variables using linear models with shrinkage. We now turn to the other common machine learning task, classification. We study a simplified setting where we have only two classes (labeled 0 and 1) so that the outcome is $y_i \in \{0, 1\}$, and we want to train a model to predict the class membership of observations based on some set of features $\mathbf{x}_i$.

We begin with logistic regression, then study decision trees and random forests. After fitting and tuning these models, we compare their predictions on an independent test sample. We conclude the lecture with an optional section on support vector machines (SVM).

The composable API of scikit-learn makes it easy to try different classifiers in a processing and estimation pipeline. Nevertheless, each classifier has its own peculiarities, and we need to account for these differences in feature construction, scaling, and hyperparameters:

- Logistic regression uses an index that is linear in its input features. We can add polynomial interactions to capture nonlinear relationships. Decision trees and random forests, on the other hand, can capture nonlinear relationships without this expansion. With SVM, we can select a nonlinear kernel instead of adding polynomial features.
- We standardize features for regularized logistic regression and SVM so that their scales do not dominate the penalty or distance calculations. Decision trees and random forests do not require this scaling.
- Each classifier has its own hyperparameters. We use cross-validation to select their values.

4.2 Example data for classification

To compare classification methods throughout this lecture, we create an artificial sample with two features (x_{1i}, x_{2i}) that are mapped into a binary outcome $y_i \in \{0, 1\}$. We restrict our attention to only two features since this makes it easy to plot decision regions.

4.2.1 Creating a demo dataset

Assume that the data-generating process can be characterized as follows:

$$y_i = \begin{cases} 1 & \text{if } f(x_{1i}, x_{2i}) + \epsilon_i \geq 0 \\ 0 & \text{else} \end{cases}$$

where $f(\bullet)$ is some deterministic function of (x_{1i}, x_{2i}) and ϵ_i is an error term. For illustration, we'll assume that $f(\bullet)$ is a combination of trigonometric functions. The data-generating process used throughout this lecture is

$$f(x_{1i}, x_{2i}) = \sin(2\pi x_{1i}) \cos(\pi x_{2i})$$
$$\epsilon_i \stackrel{\text{iid}}{\sim} \mathcal{N}(0, \sigma_\epsilon^2)$$

The code below defines $f(x_{1i}, x_{2i})$:

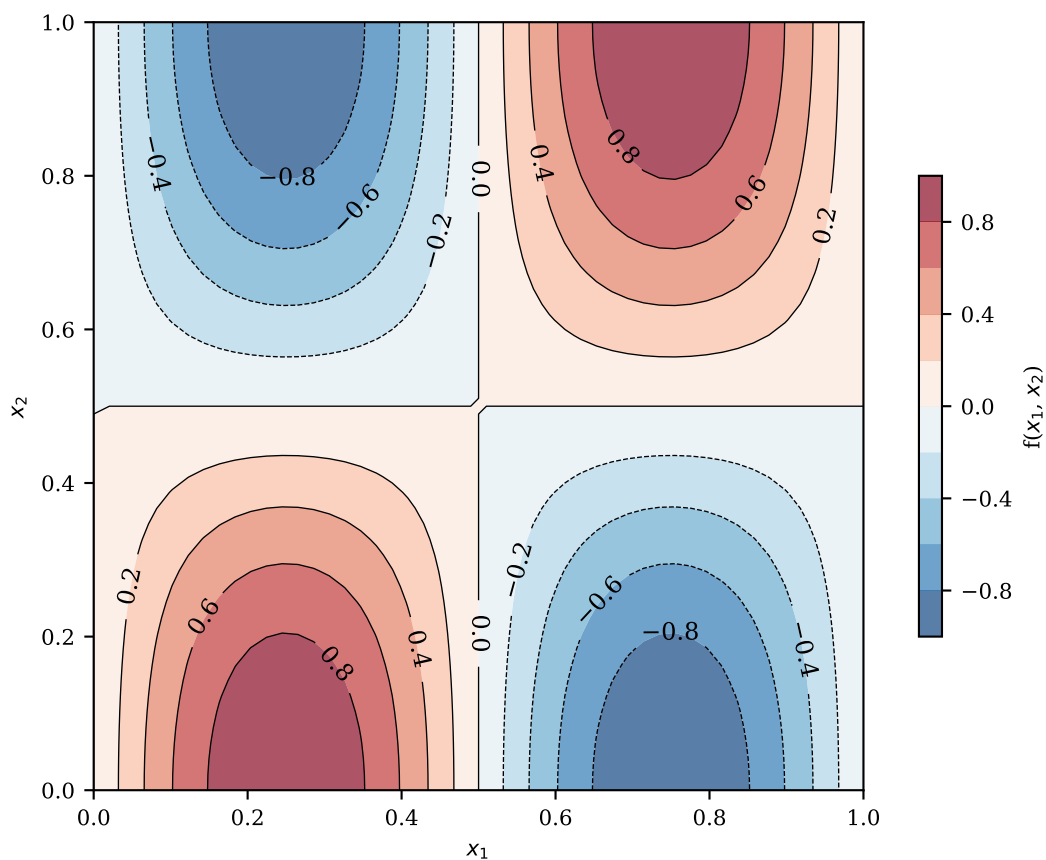
```
[1]: import numpy as np

def f(x1, x2):
    """
    True function for classification examples
    """
    return np.sin(2 * np.pi * x1) * np.cos(np.pi * x2)
```

To keep the notebook clean and focused on the main ideas, in this lecture we'll use a helper module `lecture4_plots.py` which contains most plotting functions. To create a contour plot of the true relationship, we import the function `plot_true_relationship()` from this module and pass `f` as its argument.

```
[2]: from lecture4_plots import plot_true_relationship

_ = plot_true_relationship(f)
```



As you see, this function takes on positive values in two quadrants (where y_i would be 1 without noise) and negative values in the other two (where y_i would be 0 without noise). Without some noise ϵ_i , class membership would be trivial to predict, so we next create a sample that incorporates noise.

To do this, we define the function `create_class_data()` which we'll use throughout the lecture to create a demo dataset for classification of the desired size:

```
[3]: def create_class_data(N=100, sigma=0.2, rng=None):
    """
    Create synthetic data for binary classification examples.

    Parameters
    -----
```

```

N : int
    Number of observations to generate.
sigma : float
    Standard deviation of the error term.
rng : numpy.random.Generator, optional
    Random number generator to use.

Returns
-----
X : array-like
    Feature matrix.
y : array-like
    Binary response variable.

"""

if rng is None:
    rng = np.random.default_rng(seed=1234)

# Draw features from uniform distribution
x1 = rng.uniform(0, 1, size=N)
x2 = rng.uniform(0, 1, size=N)

z = f(x1, x2)

# Add noise to latent variable if noise variance is positive
if sigma > 0:
    # Draw errors from normal distribution
    epsilon = rng.normal(0, sigma, size=N)
    z += epsilon

# Convert latent variable to binary response variable with values 0 and 1
y = (z >= 0).astype(int)

# Stack features into matrix
X = np.column_stack((x1, x2))

return X, y

```

We can now use this function to create the demo dataset:

```

[4]: # Sample size
N = 100

# Standard deviation of noise
sigma_eps = 0.2

# Create demo dataset for classification
X, y = create_class_data(N=N, sigma=sigma_eps)

```

We can verify that y_i takes on only two values with class labels 0 and 1:

```

[5]: import pandas as pd

# Tabulate unique values of y and their counts
pd.Series(y).value_counts()

```

```

[5]: 0    54
     1    46
     Name: count, dtype: int64

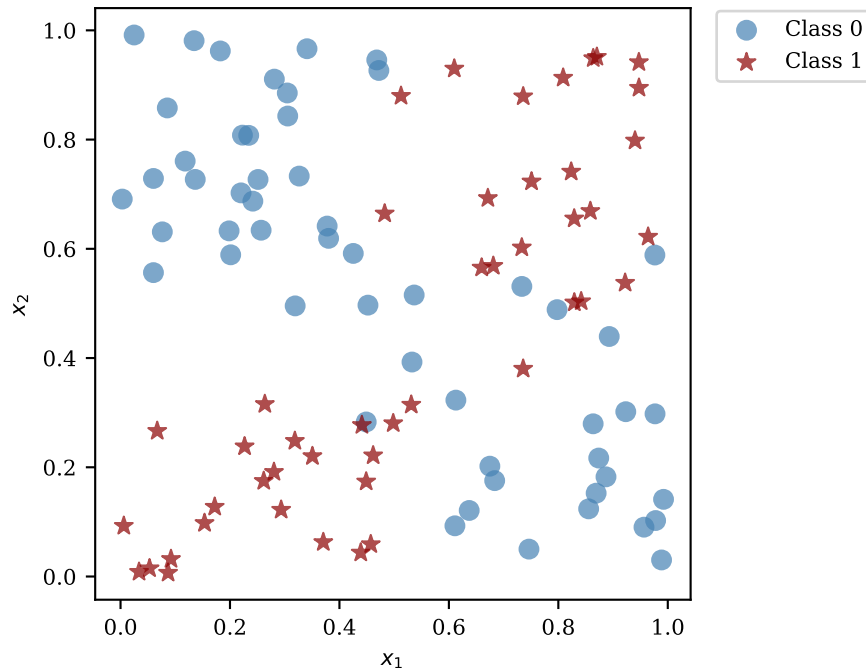
```

We import the function `plot_classes()` defined in `lecture4_plots.py` to create a scatter plot of the sample. This plot uses different marker symbols based on the class labels.

The figure below visualizes the synthetic sample we have just created. As you can see, unlike in the previous graph, the class membership is no longer clearly split between quadrants due to the additional noise ϵ_i .

```
[6]: from lecture4_plots import plot_classes

# Plot demo dataset for classification
_ = plot_classes(X, y)
```



4.2.2 Training and test samples

With real data, we would usually reserve a test sample before fitting or tuning our models. For classification, we can use a *stratified* split so that the training and test samples have approximately the same proportions of each class. We can do this with `train_test_split()` by passing the class labels through `stratify=y`.

The following cells demonstrate this split. We use names ending in `_demo` because these subsets are only an illustration. First, without stratification, the class proportions in the two subsets are quite different:

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

# Fraction of the dataset to be used for testing
test_size = 0.3

# Create a non-stratified split (the default)
X_train_demo, X_test_demo, y_train_demo, y_test_demo = train_test_split(
    X, y, test_size=test_size, random_state=123
)

print('Distribution of classes in demo training sample:')
print(pd.Series(y_train_demo).value_counts() / len(y_train_demo))

print('\nDistribution of classes in demo test sample:')
print(pd.Series(y_test_demo).value_counts() / len(y_test_demo))
```

Distribution of classes in demo training sample:

```
1    0.542857
0    0.457143
Name: count, dtype: float64
```

Distribution of classes in demo test sample:

```
0    0.733333
1    0.266667
Name: count, dtype: float64
```

This is due to the small sample size and this particular seed. Stratifying the split ensures that the class proportions are similar in both subsets even for this small sample:

```
[8]: # Demonstrate a stratified split
X_train_demo, X_test_demo, y_train_demo, y_test_demo = train_test_split(
    X, y, stratify=y, test_size=test_size, random_state=123
)

print('Distribution of classes in demo training sample:')
print(pd.Series(y_train_demo).value_counts() / len(y_train_demo))

print('\nDistribution of classes in demo test sample:')
print(pd.Series(y_test_demo).value_counts() / len(y_test_demo))
```

Distribution of classes in demo training sample:

```
0    0.542857
1    0.457143
Name: count, dtype: float64
```

Distribution of classes in demo test sample:

```
0    0.533333
1    0.466667
Name: count, dtype: float64
```

For our synthetic example, we can generate independent observations whenever we need them. We therefore use all 100 observations for training. After fitting and tuning the logistic regression, decision tree, and random forest models, we will generate a fresh test sample of 10,000 observations from the same process to compare their predictions.

Until that final comparison, we use training observations for the plots and metric examples. Training scores describe the fit to observations already seen by the model; they do not measure performance on new observations.

```
[9]: # Use the complete synthetic sample for training
X_train = X
y_train = y
```

4.3 Logistic regression

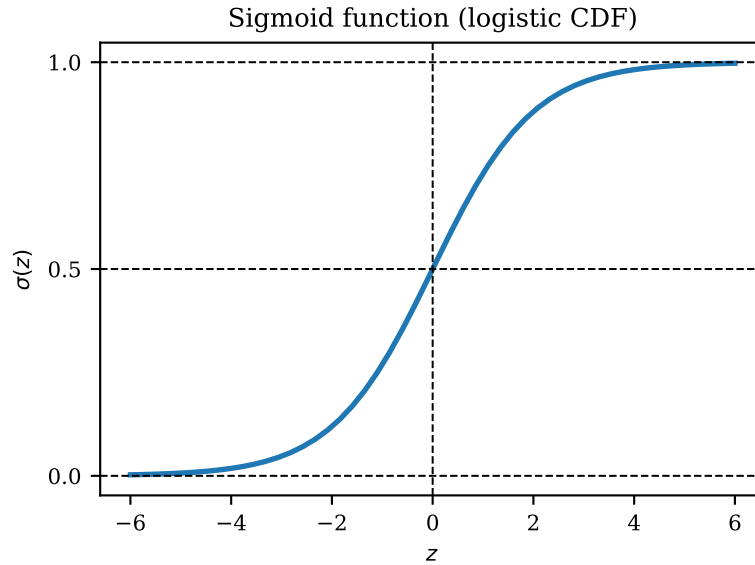
Logistic regression is a well-established approach to performing classification tasks in econometrics. In the binary setting, we model the probability of the outcome being 1 as

$$p(\mathbf{x}_i) \equiv \text{Prob}(y_i = 1 \mid \mathbf{x}_i) = \frac{1}{1 + \exp(-(\mu + \mathbf{x}_i' \boldsymbol{\beta}))}$$

where the function on the right $\sigma(z) = \frac{1}{1+e^{-z}}$ is called the **sigmoid function** in machine learning or the **logistic cumulative distribution function (CDF)** in statistics and econometrics. Note that this function maps any (positive or negative) real value z into a value in the interval $(0, 1)$, and hence is well suited to modeling the probability that $y_i = 1$.

We can easily verify this by plotting the logistic CDF from `scipy.stats`, using the helper function `plot_sigmoid()` from `lecture4_plots.py`:

```
[10]: from lecture4_plots import plot_sigmoid
      _ = plot_sigmoid()
```



As you can see, the value of $\sigma(z)$ approaches 0 as z becomes increasingly negative, and it approaches 1 for large values of z .

In machine learning, logistic regression is categorized as a linear model (see the scikit-learn user guide on [linear models](#)) even though $\sigma(z)$ is clearly not linear. This is because the index z_i that enters the sigmoid function is assumed to be linear in the features $\mathbf{x}_i$,

$$z_i = \mu + \mathbf{x}_i' \boldsymbol{\beta}$$

Consequently, one can show that the log odds are a linear function of features:

$$\log \left(\frac{p(\mathbf{x}_i)}{1 - p(\mathbf{x}_i)} \right) = \mu + \mathbf{x}_i' \boldsymbol{\beta}$$

Before we can fit the model using logistic regression, we need to define an appropriate loss function. This loss function is derived from the log-likelihood function of a binary logit model, defined as

$$\mathcal{L}(\mu, \boldsymbol{\beta}) = \sum_{i=1}^N \left[y_i \log(p(\mathbf{x}_i)) + (1 - y_i) \log(1 - p(\mathbf{x}_i)) \right]$$

The details underlying the derivation of this log-likelihood function are beyond the scope of this course. For our purposes, it's sufficient to know that in maximum likelihood estimation (MLE), the parameters μ and $\boldsymbol{\beta}$ are chosen so that the log-likelihood $\mathcal{L}$ of observing a given sample of $(y_i, \mathbf{x}_i)$ is *maximized*.

In machine learning, we typically minimize a loss function instead of maximizing a likelihood function. To derive the loss function L for the logistic regression, we simply take the *negative* log-likelihood function and find the parameters $(\mu, \boldsymbol{\beta})$ which *minimize* the loss L :

$$L(\mu, \boldsymbol{\beta}) = - \underbrace{\frac{1}{N} \mathcal{L}(\mu, \boldsymbol{\beta})}_{\text{scaled log-likelihood}} + \underbrace{\frac{r(\boldsymbol{\beta})}{NC}}_{\text{regularization}}$$

Moreover, we have already incorporated the scaled regularization term $\frac{r(\boldsymbol{\beta})}{NC}$, where $r(\bullet)$ is a function of $\boldsymbol{\beta}$, which is used analogously to Ridge regression and Lasso to shrink the parameters $\boldsymbol{\beta}$ and prevent

overfitting. We divide both the negative log-likelihood and the penalty by N to match scikit-learn's convention for C . The fitted logistic regression parameters are thus

$$(\hat{\mu}, \hat{\beta}) = \arg \min_{\mu, \beta} L(\mu, \beta)$$

4.3.1 Fitting a simple model

We first fit logistic regression to the synthetic training sample without imposing any regularization, thus setting $C = \infty$. We'll cross-validate the regularization strength later.

Specifying the regularization strength

We proceed to fit the model using the `LogisticRegression` estimator from scikit-learn. `LogisticRegression` implements four different types of regularization which correspond to different functional forms for $r(\bullet)$ in the penalty term:

1. the *L1 penalty* we encountered with Lasso;
2. the *L2 penalty* from the Ridge regression;
3. the convex combination of these used for the *elastic net*; or
4. *no penalty* (this then corresponds to the classical binary logit used in econometrics to model discrete choice).

The following table lists the available regularization terms and the corresponding arguments for `LogisticRegression`. The argument C is the *inverse* penalty strength: *higher* values of C *decrease* regularization.

Penalty type	$r(\beta)$	<code>l1_ratio</code>	C
L1	$r(\beta) = \ \beta\ _1 = \sum_{k=1}^K \ \beta_k\ $	1	desired inverse penalty
L2	$r(\beta) = \frac{1}{2} \ \beta\ _2^2 = \frac{1}{2} \sum_{k=1}^K \beta_k^2$	0	desired inverse penalty
L1 and L2	$r(\beta) = \rho \ \beta\ _1 + \frac{1-\rho}{2} \ \beta\ _2^2$	$0 < \rho < 1$	desired inverse penalty
None	0	0	<code>np.inf</code>

The default solver, 'lbfgs', supports L2 regularization and unpenalized models. For L1 or elastic net, we can use `solver='saga'`.

Fitting the logistic regression model

We first experiment with the simplest possible linear model of two explanatory variables where

$$z_i = \mu + \beta_1 x_{1i} + \beta_2 x_{2i}$$

so that the probability of observing class 1 is

$$p(\mathbf{x}_i) \equiv \text{Prob}(y_i = 1 \mid \mathbf{x}_i) = \frac{1}{1 + \exp(-(\mu + \beta_1 x_{1i} + \beta_2 x_{2i}))}$$

For now, we omit regularization by specifying `C=np.inf`.

```
[11]: from sklearn.linear_model import LogisticRegression

# Create logistic regression model without regularization penalty
lr = LogisticRegression(C=np.inf, fit_intercept=True, max_iter=1000)
```

Note that we also skipped input scaling here since we have only two features without polynomial interactions, and these only take values on the interval $[0, 1]$. Hence there is no need for scaling to address any potential numerical stability issues.

The following code fits the logistic regression model to the training sample:

```
[12]: lr.fit(X_train, y_train)

# Retain the fitted linear model
lr_linear = lr
```

With $C=np.inf$, we estimate the coefficients using only the logistic loss. We introduce a finite value of C when we add regularization below.

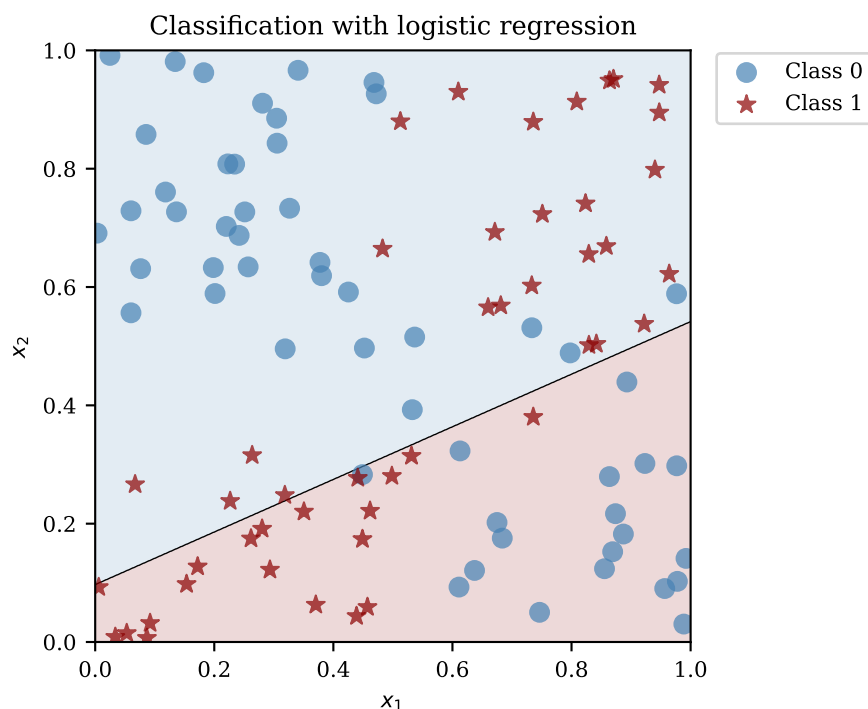
4.3.2 Assessing model accuracy

We first plot the decision boundaries implied by the fitted model. To this end, we use the helper function `plot_decision_boundary()` implemented in `lecture4_plots.py`. The function takes as argument an existing Matplotlib Axes object and adds the decision boundaries to an existing figure.

```
[13]: from lecture4_plots import plot_decision_boundary

# Create x-values to evaluate decisions (used for either dimension)
xvalues = np.linspace(0, 1, 1000)

ax = plot_classes(X_train, y_train)
plot_decision_boundary(ax, xvalues, lr)
_ = ax.set_title('Classification with logistic regression')
```



The simple linear model misclassifies many training observations. For any combination of (x_{1i}, x_{2i}) in the blue area, it predicts $y_i = 0$; in the red area, it predicts $y_i = 1$.

We can compute accuracy, the share of correctly predicted class labels, using the estimator's `score()` method or the `accuracy_score()` function. These return the same result. We use training predictions here to illustrate the calculation and reserve the independent test comparison for later.

```
[14]: # Predict class labels for the training observations
y_train_pred = lr.predict(X_train)
```

```
[15]: from sklearn.metrics import accuracy_score

# Compute training accuracy
acc_train = accuracy_score(y_train, y_train_pred)
print(f'Training accuracy: {acc_train:.3f}')
```

Training accuracy: 0.550

Alternatively, we can use the `score()` method directly to compute the same training accuracy:

```
[16]: acc_train = lr.score(X_train, y_train)
print(f'Training accuracy: {acc_train:.3f}')
```

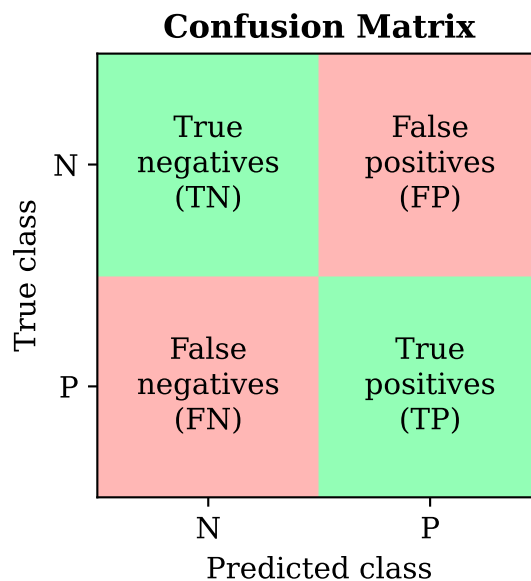
Training accuracy: 0.550

Accuracy tells us how often the model predicts correctly, but it does not distinguish between different types of mistakes. The confusion matrix makes this distinction explicit and provides the counts used to calculate other metrics.

First, consider the confusion matrix for a generic classification problem (with labels P and N for Positive and Negative), plotted below:

```
[17]: from lecture4_plots import plot_generic_confusion_matrix

plot_generic_confusion_matrix()
```



As you see, a classifier would ideally make predictions that fall in the green diagonal cells. Using the labels of this matrix, the accuracy score we have computed above is given by

$$ACC = \frac{TP + TN}{FP + FN + TP + TN}$$

Scikit-learn implements many other commonly used scoring metrics in the module `sklearn.metrics`:

- *Precision* is the share of predicted positives that are actually positive. We compute it using `precision_score()` as

$$PRE = \frac{TP}{TP + FP}$$

- *Recall* is the share of actual positives that the model correctly identifies. We compute it using `recall_score()` as

$$REC = \frac{TP}{FN + TP}$$

- The *F1 score* combines precision and recall into a single measure. We compute it using `f1_score()` as

$$F1 = 2 \frac{PRE \cdot REC}{PRE + REC}$$

The score is high only when both precision and recall are high.

Precision, recall, and F1 depend on which class we treat as positive. The functions `precision_score()`, `recall_score()`, and `f1_score()` each accept a `pos_label` argument that selects this class, with a default of 1. Accuracy is unaffected by this choice.

We define `tabulate_training_metrics()` to compute and print these four metrics for our training predictions:

```
[18]: from sklearn.metrics import f1_score, precision_score, recall_score

def tabulate_training_metrics(y_train, y_train_pred):
    """
    Print accuracy, precision, recall, and F1 score for training predictions.

    Parameters
    -----
    y_train : array
        Observed class labels in the training sample.
    y_train_pred : array
        Predicted class labels for the training observations.
    """
    acc_train = accuracy_score(y_train, y_train_pred)
    pre = precision_score(y_train, y_train_pred)
    rec = recall_score(y_train, y_train_pred)
    f1 = f1_score(y_train, y_train_pred)

    print(f'Training accuracy: {acc_train:.3f}')
    print(f'Training precision: {pre:.3f}')
    print(f'Training recall: {rec:.3f}')
    print(f'Training F1 score: {f1:.3f}')
```

```
[19]: tabulate_training_metrics(y_train, y_train_pred)
```

```
Training accuracy:  0.550
Training precision: 0.513
Training recall:    0.435
Training F1 score:  0.471
```

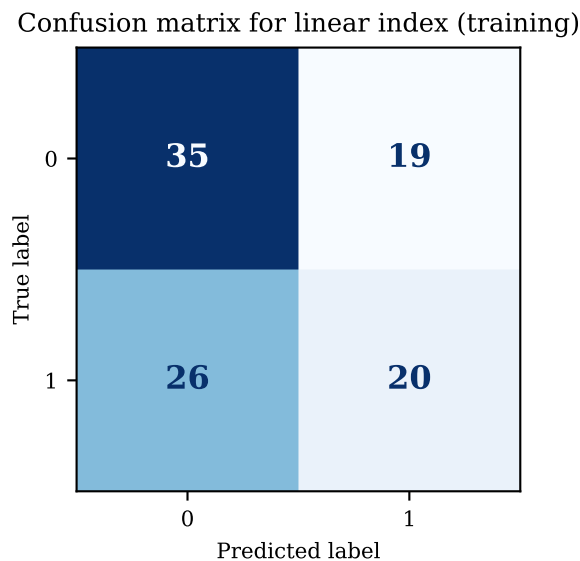
These metrics describe different aspects of the model's fit to the training observations. We will assess performance on new observations after fitting all main classifiers.

We can plot the confusion matrix using `ConfusionMatrixDisplay`. This plot can be created from predicted values or directly from an estimator. Here, class 1 is positive, and scikit-learn orders the labels as 0, then 1, so true positives appear in the lower-right cell. Below, we use the training predictions:

```
[20]: from sklearn.metrics import ConfusionMatrixDisplay

# Plot confusion matrix for the training predictions
_ = ConfusionMatrixDisplay.from_predictions(
    y_train,
    y_train_pred,
    colorbar=False,
```

```
cmap='Blues',
text_kw={'fontsize': 12, 'fontweight': 'bold'},
).ax_.set_title('Confusion matrix for linear index (training)')
```



The confusion matrix shows that there is a sizable number of misclassified observations (the off-diagonal elements).

Class imbalance

In real applications, one class may be much rarer than the other. Suppose only 100 of 10,000 observations are positive, for example fraudulent transactions. A classifier that always predicts the negative class achieves *99% accuracy*, but detects none of the positive cases: its recall is *zero*. We should therefore compare accuracy with the simple baseline of always predicting the most common class.

When missing positive cases is costly, we pay particular attention to recall. When false alarms are costly, precision also matters. F1 summarizes their tradeoff in a single number, but the appropriate metric still depends on the costs of different mistakes.

We can also use `balanced_accuracy_score()`, which averages recall computed separately for each class. In our example, the classifier that always predicts the negative class has balanced accuracy of 50%, since it identifies all negative cases and none of the positive cases.

4.3.3 Digression: linear decision boundary

You might wonder why the decision boundary plotted earlier is a straight line even though the sigmoid function is clearly nonlinear. This arises because `LogisticRegression` assigns predictions based on the most likely class as determined by the predicted probabilities of belonging to each class for each observation. We can predict these underlying probabilities using the `predict_proba()` method, for example for the first five training observations:

```
[21]: # Predict probabilities for the first five training observations
lr.predict_proba(X_train)[:5]
```

```
[21]: array([[0.44213578, 0.55786422],
            [0.58697448, 0.41302552],
            [0.4489442 , 0.5510558 ],
            [0.49038364, 0.50961636],
            [0.56350387, 0.43649613]])
```

Based on these predicted probabilities, LogisticRegression assigns the predicted class as the most likely outcome. With only two categories, $\hat{y}_i = 1$ is therefore predicted whenever the value in the second column is larger than 0.5. Using the fitted coefficients in the sigmoid function, we have

$$\begin{aligned}
 \hat{y}_i = 1 &\iff \sigma(\hat{\mu} + \hat{\beta}_1 x_{1i} + \hat{\beta}_2 x_{2i}) > 0.5 \\
 &\iff \frac{1}{1 + \exp(-\hat{\mu} - \hat{\beta}_1 x_{1i} - \hat{\beta}_2 x_{2i})} > 0.5 \\
 &\iff \exp(-\hat{\mu} - \hat{\beta}_1 x_{1i} - \hat{\beta}_2 x_{2i}) < 1 \\
 &\iff \hat{\mu} + \hat{\beta}_1 x_{1i} + \hat{\beta}_2 x_{2i} > 0 \\
 &\iff x_{2i} < -\frac{\hat{\mu}}{\hat{\beta}_2} - \frac{\hat{\beta}_1}{\hat{\beta}_2} x_{1i}
 \end{aligned}$$

The last step above follows because $\hat{\beta}_2 < 0$ in this example, as you can verify by checking the attribute `lr.coef_`. The takeaway from these derivations is that any combination of features x_{1i}, x_{2i} is mapped to $\hat{y}_i = 1$ if x_{2i} lies underneath the line defined by $-\frac{\hat{\mu}}{\hat{\beta}_2} - \frac{\hat{\beta}_1}{\hat{\beta}_2} x_{1i}$, and therefore the decision boundary is a *straight line*.

4.3.4 Fitting a model with polynomials

We now extend the simple model and approximate the nonlinear relationship between y_i and $\mathbf{x}_i$ using polynomials in x_{1i}, x_{2i} where we allow for interaction terms up to some upper bound:

$$z_i = \mu + \beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{1i} x_{2i} + \beta_4 x_{1i}^2 + \beta_5 x_{2i}^2 + \dots$$

```
[22]: from sklearn.pipeline import make_pipeline
      from sklearn.preprocessing import PolynomialFeatures, StandardScaler

      # Maximum polynomial degree
      degree = 5

      # Create logistic regression model without regularization penalty
      lr = LogisticRegression(fit_intercept=True, C=np.inf, max_iter=1_000)

      # Create pipeline with polynomial features and logistic regression
      pipe_lr = make_pipeline(
          StandardScaler(),
          PolynomialFeatures(degree=degree, include_bias=False),
          StandardScaler(),
          lr,
      )
```

At this point we don't strictly need to scale the polynomial features using the `StandardScaler`, but this will be necessary below when we introduce a regularization term.

We can now fit the model on the training data:

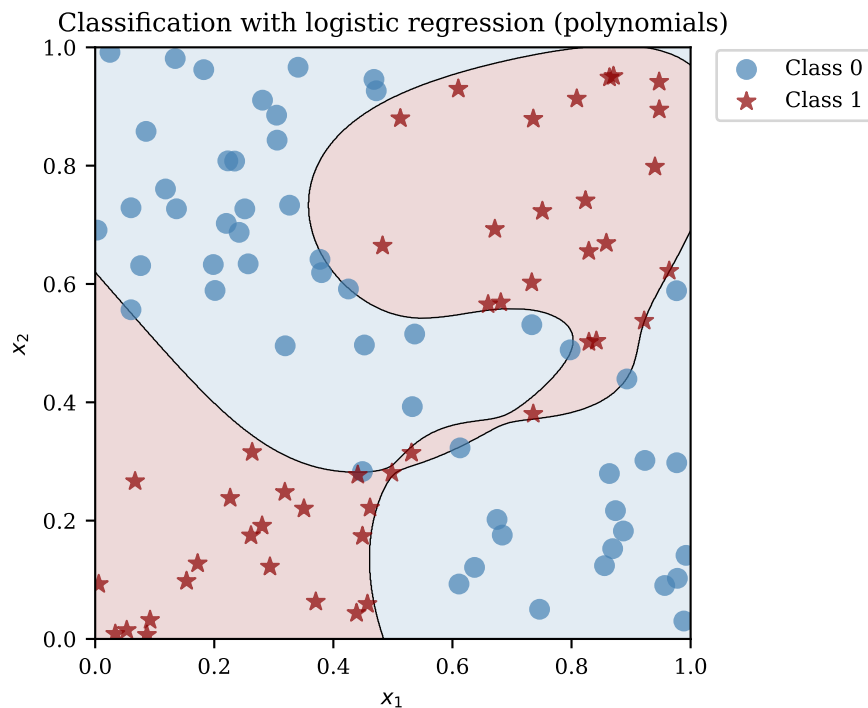
```
[23]: pipe_lr.fit(X_train, y_train)

      # Retain the unpenalized polynomial pipeline
      lr_pipe_unpenalized = pipe_lr
```

Using the functions defined above, we can plot the decision boundaries of this more complex model:

```
[24]: # Create x-values used to evaluate decisions
      xvalues = np.linspace(0, 1, 1000)
      ax = plot_classes(X_train, y_train)
```

```
plot_decision_boundary(ax, xvalues, pipe_lr)
_ = ax.set_title('Classification with logistic regression (polynomials)')
```



The polynomial features allow a more flexible decision boundary. We can compute the same metrics to see how well this model fits the training observations:

```
[25]: # Predict class labels for the training observations
y_train_pred = pipe_lr.predict(X_train)

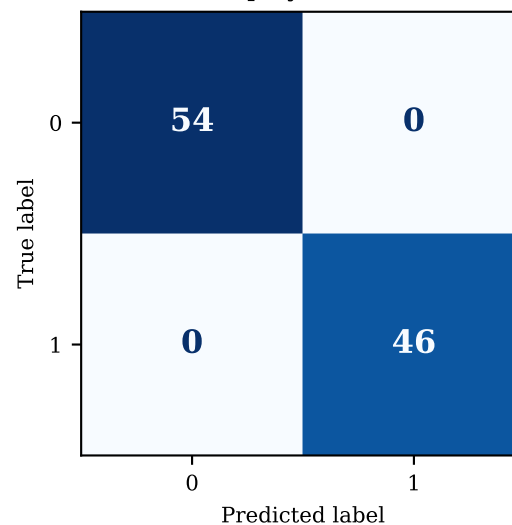
tabulate_training_metrics(y_train, y_train_pred)
```

```
Training accuracy:  1.000
Training precision: 1.000
Training recall:    1.000
Training F1 score:  1.000
```

The confusion matrix summarizes these training predictions. A good fit to the training observations can still come with overfitting, so we will compare the models on independent test observations later.

```
[26]: # Plot confusion matrix for the training predictions
_ = ConfusionMatrixDisplay.from_predictions(
    y_train,
    y_train_pred,
    colorbar=False,
    cmap='Blues',
    text_kw={'fontsize': 12, 'fontweight': 'bold'},
).ax_.set_title('Confusion matrix for polynomial features (training)')
```

Confusion matrix for polynomial features (training)



As you see from the above metrics and the confusion matrix, the model with a degree 5 polynomial predicts *every single* observation in the training sample correctly. However, the diagnostic plot above suggests overfitting, as the decision boundaries are highly irregular and follow the training observations too closely.

4.3.5 Cross-validating the penalty term

So far, we haven't introduced a regularization term when fitting the logistic regression. Analogous to Ridge regression and Lasso, regularization can help prevent overfitting. Since we don't know the best regularization parameter C , we use `GridSearchCV()` around the complete pipeline.

We first define a cross-validation splitter and scoring metric that we will use for all models in this lecture:

```
[27]: from sklearn.model_selection import StratifiedKFold

# Reproducible shuffled folds that preserve the class proportions
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1234)

# Define the scoring metric for model evaluation
scoring = 'accuracy'
```

We define five folds with `StratifiedKFold()` to keep the class proportions approximately equal across folds. We set `shuffle=True` and `random_state=1234` to shuffle observations within each class and make the splits reproducible. Shuffling is not needed for this example because our training observations were already drawn independently in random order. We include it to make the split strategy explicit.

Stratified splitting preserves class proportions across subsets; it does not remove the imbalance or make accuracy informative about rare positive cases. With severe imbalance, we also need enough minority-class observations for the chosen number of CV folds.

We set `scoring='accuracy'` because our classes are approximately balanced and we count mistakes in both classes equally.

If we choose balanced accuracy or F1 for an application, we can pass `scoring='balanced_accuracy'` or `scoring='f1'` to `GridSearchCV`. We choose the scoring rule before tuning and report the relevant metrics on independent test observations.

We construct the estimation pipeline as usual: The first scaler standardizes the original features, and the second standardizes the polynomial features. In each fold, we fit each `StandardScaler` using only the training observations and apply those fitted means and standard deviations to the validation

observations. Fitting either scaler on the entire training sample before cross-validation would let the validation observations influence the transformation used for fitting. This is *data leakage*, even though the scalers do not use the labels. Our synthetic sample is no exception: the polynomial columns' sample means and standard deviations still depend on the observations. The polynomial expansion itself uses fixed powers and learns no means or scales, but we keep all preprocessing inside the pipeline so the complete procedure is cross-validated.

For this example, we use the L2 penalty, selected with `l1_ratio=0.0`. The complete pipeline is defined as follows:

```
[28]: # Pipeline with polynomial features, scaling, and L2 logistic regression
pipe_lr = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(include_bias=False),
    StandardScaler(),
    LogisticRegression(l1_ratio=0.0, max_iter=1_000),
)
```

We try 100 values of C , spaced evenly on a log scale between 10^{-2} and 10^2 . The pipeline parameter `logisticregression__C` uses the step name and parameter name separated by a double underscore. Additionally, we cross-validate the polynomial degree, which is specified by the pipeline parameter `polynomialfeatures__degree`. We try degrees 1 through 5.

```
[29]: from sklearn.model_selection import GridSearchCV

# Candidate inverse penalty strengths, spaced uniformly in logs on [1e-2, 1e2]
c_values = np.logspace(-2, 2, 100)

# Candidate polynomial degrees
degrees = [1, 2, 3, 4, 5]

# Define the parameter grid for GridSearchCV
param_grid = {'polynomialfeatures__degree': degrees, 'logisticregression__C': c_values}

# Define the GridSearchCV object for hyperparameter tuning
lr_cv = GridSearchCV(
    estimator=pipe_lr,
    param_grid=param_grid,
    scoring=scoring,
    cv=cv,
    refit=True,
    n_jobs=4,
)
```

We run the search on the original training features. `GridSearchCV` fits both scalers and the classifier separately within each fold:

```
[30]: # Run cross-validation on the original, unscaled features
lr_cv.fit(X_train, y_train)
```

Once cross-validation is complete, we can retrieve the selected C and polynomial degree from `best_params_` and the corresponding mean validation accuracy from `best_score_`:

```
[31]: print(f'Best C: {lr_cv.best_params_["logisticregression__C"]:.4f}')
print(f'Best polynomial degree: {lr_cv.best_params_["polynomialfeatures__degree"]}')
print(f'Best validation accuracy: {lr_cv.best_score_:.3f}')
```

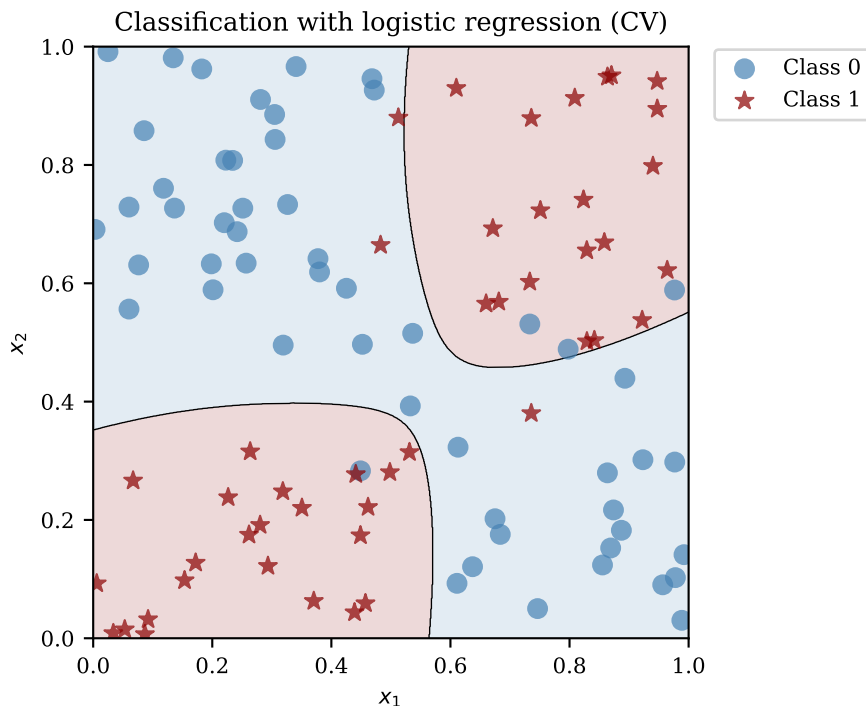
```
Best C: 0.6579
Best polynomial degree: 4
Best validation accuracy: 0.910
```

Note that unlike the penalty parameters used for Ridge and Lasso, C in this case is the *inverse* penalty strength, i.e., larger values of C introduce a *smaller* penalty.

With `refit=True`, `GridSearchCV` refits the selected pipeline on the entire training sample. We can use `lr_cv` directly for predictions and plots, which apply the complete fitted pipeline, including the scalers, polynomial expansion, and classifier.

We reuse the plotting code from above to plot the decision boundaries for the optimal model:

```
[32]: # Create x-values used to evaluate decisions
xvalues = np.linspace(0, 1, 1000)
ax = plot_classes(X_train, y_train)
plot_decision_boundary(ax, xvalues, lr_cv)
_ = ax.set_title('Classification with logistic regression (CV)')
```



We can compute the selected pipeline's training accuracy, precision, recall, and F1 score using the same functions as before:

```
[33]: # Compute training metrics for the cross-validated model
y_train_pred = lr_cv.predict(X_train)

tabulate_training_metrics(y_train, y_train_pred)
```

```
Training accuracy: 0.930
Training precision: 0.915
Training recall: 0.935
Training F1 score: 0.925
```

All tuning uses only the training sample. We leave the independent test comparison until after the random forest section, when all main models have been fitted.

4.3.6 Dedicated cross-validation with `LogisticRegressionCV`

The dedicated `LogisticRegressionCV()` class can reuse coefficient estimates along the regularization path, making it faster than fitting each candidate independently. It is useful when no preprocessing needs to be learned within the folds.

For this illustration, we construct polynomial features directly from the original inputs, without fitting a scaler. We keep degree 4, which the preceding grid search selected. `LogisticRegressionCV` tunes

the regularization strength for a fixed feature matrix, so it does not also cross-validate the polynomial degree.

The earlier grid search included scaling, which changes how regularization acts on the polynomial coefficients. Degree 4 is therefore not guaranteed to be optimal for this specification, and the selected C values are not directly comparable.

We specify accuracy scoring explicitly.

With `use_legacy_attributes=False`, C_* is a scalar, which we can format directly.

```
[34]: from sklearn.linear_model import LogisticRegressionCV

# Candidate inverse penalty strengths, spaced uniformly in logs on [1e-2, 1e2]
c_values_dedicated = np.logspace(-2, 2, 100)

# Polynomial features with degree 4 selected by the preceding grid search
poly_fixed = PolynomialFeatures(degree=4, include_bias=False)
X_train_fixed = poly_fixed.fit_transform(X_train)

# Dedicated cross-validation on these fixed polynomial features
lr_cv_dedicated = LogisticRegressionCV(
    Cs=c_values_dedicated,
    cv=cv,
    scoring=scoring,
    l1_ratios=(0.0,),
    max_iter=1_000,
    n_jobs=4,
    use_legacy_attributes=False,
).fit(X_train_fixed, y_train)

print(f'LogisticRegressionCV selected C: {lr_cv_dedicated.C_:.4f}')
```

LogisticRegressionCV selected C: 1.2619

4.3.7 Validation curves

A validation curve plots a model's validation score against a hyperparameter, here C . We use `validation_curve()` to fit the complete pipeline within each fold and compute training and validation accuracy for a range of values of C . We define a separate pipeline with the polynomial degree fixed at 4, as selected by the preceding grid search, since we want to focus on the effect of C on the model's performance.

The argument `param_name` identifies the parameter to vary. For our pipeline, `logisticregression__C` combines the step name and parameter name with a double underscore. The argument `param_range` supplies the candidate values. We use the same stratified folds and accuracy scoring rule as in our grid search:

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

# Pipeline for the validation curve with polynomial degree fixed at 4
pipe_lr_validation = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(degree=4, include_bias=False),
    StandardScaler(),
    LogisticRegression(l1_ratio=0.0, max_iter=1_000),
)

# Define range of values for C, spaced uniformly in logs
param_range = np.logspace(-4, 4, 100)

# Parameter name
```

```

param_name = 'logisticregression__C'

# Compute validation curve
train_scores, test_scores = validation_curve(
    estimator=pipe_lr_validation, # Estimator pipeline
    X=X_train, # Training data
    y=y_train, # Training labels
    param_name=param_name, # Parameter name for C in pipeline
    param_range=param_range, # Range of C values to evaluate
    cv=cv, # Same stratified folds as in the grid search
    scoring=scoring, # Same scoring rule as in the grid search
    n_jobs=4, # Number of parallel jobs
)

```

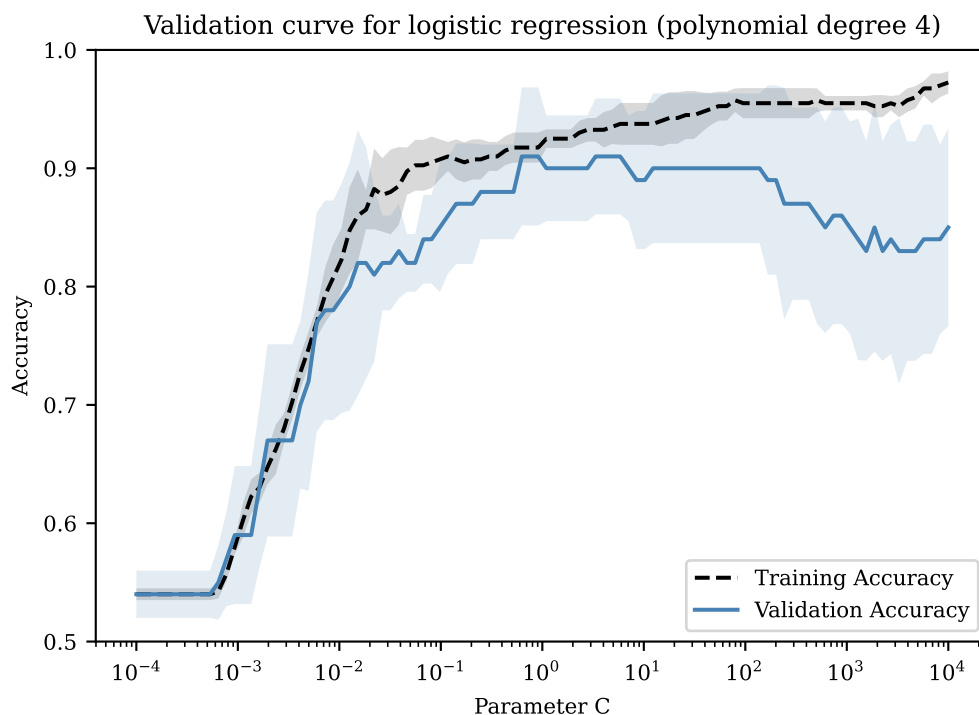
The module `lecture4_plots` implements the function `plot_accuracy_validation_curve()` which we can use to plot the validation curve for a given range of parameter values. The lines show mean accuracy across folds. The shaded bands extend one standard deviation above and below the means. We pass the metric name through `score` so that the axis and legend labels match the scoring rule:

```

[36]: from lecture4_plots import plot_accuracy_validation_curve

plot_accuracy_validation_curve(
    param_range,
    train_scores,
    test_scores,
    score='Accuracy',
    title='Validation curve for logistic regression (polynomial degree 4)',
)

```



As the plot shows, small values of C (large penalty strengths) can lead to underfitting, with low accuracy on both the training and validation folds. At large values of C , higher training accuracy can come with lower validation accuracy, suggesting overfitting.

4.4 Decision trees

Decision trees are nonlinear classifiers that are constructed by splitting the sample in two at each branch according to some criterion (e.g., if feature 1 is larger than a value a). Fitting a decision tree is the process of growing a tree and finding the criteria that split the sample to achieve the largest reduction in impurity. In principle, a tree could be grown until each leaf node is pure, i.e., it contains only observations from a single class. In practice, this can lead to very deep trees which tend to overfit the data, so it's advisable to limit the tree's growth by specifying a maximum depth.

It should be stressed that decision trees are fully nonlinear models, so there is usually no need to expand the feature space using polynomial interactions. Moreover, decision trees are not sensitive to scaling, so we can skip the entire data processing pipeline we used for other classifiers.

Decision trees are implemented in the `DecisionTreeClassifier` class in scikit-learn. We use the same 100 training observations as for logistic regression, so the final test comparison will assess models fitted on the same sample. We grow the tree by fitting it to the training data. For the reasons explained above, we limit the maximum tree depth to 3 by specifying the argument `max_depth=3`. We use `criterion='gini'` to favor splits that produce groups with less mixed class labels, and `random_state=1234` to make the result reproducible when several splits are equally good.

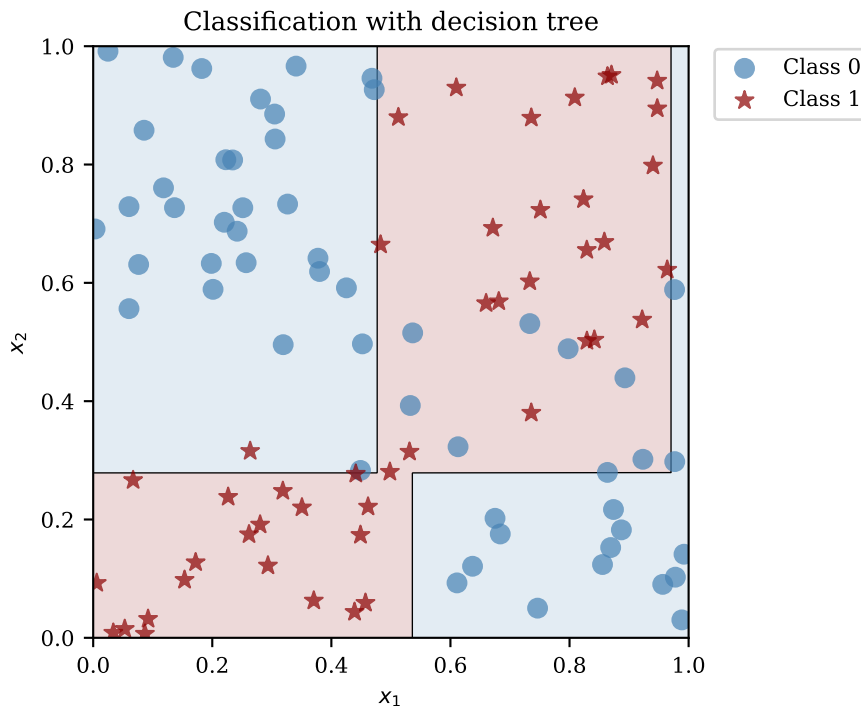
```
[37]: from sklearn.tree import DecisionTreeClassifier

# Create decision tree classifier
tree = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=1234)

# Grow tree on training data (no transformation needed)
tree.fit(X_train, y_train)
```

We now plot the decision boundaries of the fitted decision tree. As you can see, the boundaries are all parallel to the axes, which is by construction.

```
[38]: ax = plot_classes(X_train, y_train)
plot_decision_boundary(ax, xvalues, tree)
_ = ax.set_title('Classification with decision tree')
```



We can compute the tree's training accuracy, precision, recall, and F1 score. As before, these describe the fit to observations already used to grow the tree:

```
[39]: y_train_pred_tree = tree.predict(X_train)

      tabulate_training_metrics(y_train, y_train_pred_tree)
```

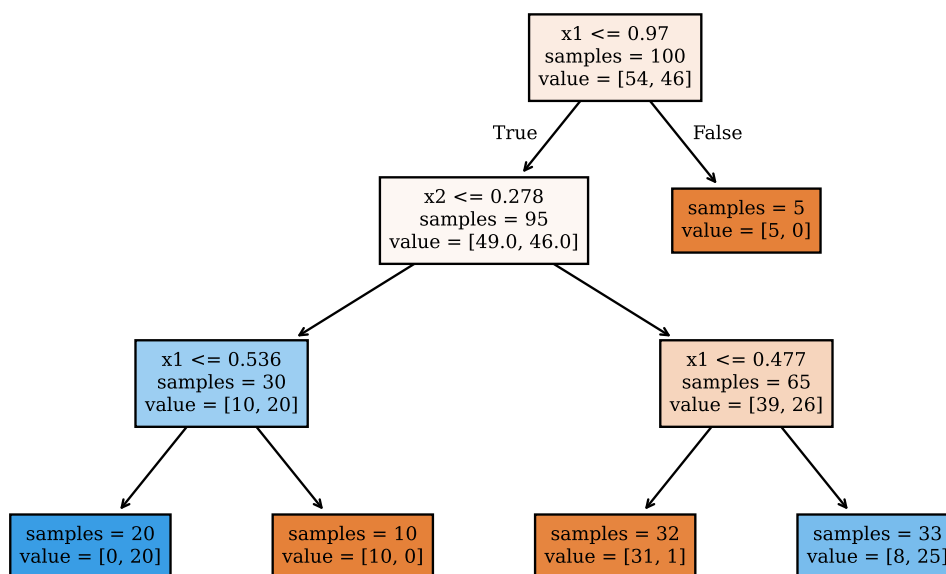
```
Training accuracy: 0.910
Training precision: 0.849
Training recall:    0.978
Training F1 score: 0.909
```

Scikit-learn offers a convenience function that visualizes the fitted tree. At each non-leaf node, the first line states the splitting criterion, the second reports the number of observations at that node, and the third gives the class counts in the order [class 0, class 1]. Leaf nodes show only the sample and class counts.

We can follow an observation down the tree by comparing its feature values with the displayed thresholds. A leaf containing observations from only one class is called *pure*.

```
[40]: import matplotlib.pyplot as plt
      from sklearn.tree import plot_tree

      # Visualize nodes of fitted decision tree
      fig, ax = plt.subplots(figsize=(8, 5))
      _ = plot_tree(
          tree, filled=True, feature_names=['x1', 'x2'], fontsize=8, impurity=False, ax=ax
      )
```



We fitted the tree above choosing a maximum tree depth ourselves. The tree depth is a hyperparameter, so we should as usual use cross-validation to select its value. To do this, we again use the `GridSearchCV` cross-validator to find the optimal tree depth for this sample. For now, we vary only `max_depth`. The random forest section introduces additional hyperparameters. We reuse the stratified folds and accuracy scoring rule from logistic regression.

```
[41]: # Create decision tree classifier used for CV
      tree = DecisionTreeClassifier(criterion='gini', random_state=1234)
```

```
# Define range of values for max_depth
max_depth = np.arange(1, 11)

# Parameter grid for GridSearchCV
param_grid = {'max_depth': max_depth}

tree_cv = GridSearchCV(
    estimator=tree, param_grid=param_grid, scoring=scoring, cv=cv, refit=True, n_jobs=4
).fit(X_train, y_train)

print(f'Best validation accuracy: {tree_cv.best_score_:.3f}')
print(f'Best max depth: {tree_cv.best_params_["max_depth"]}')

```

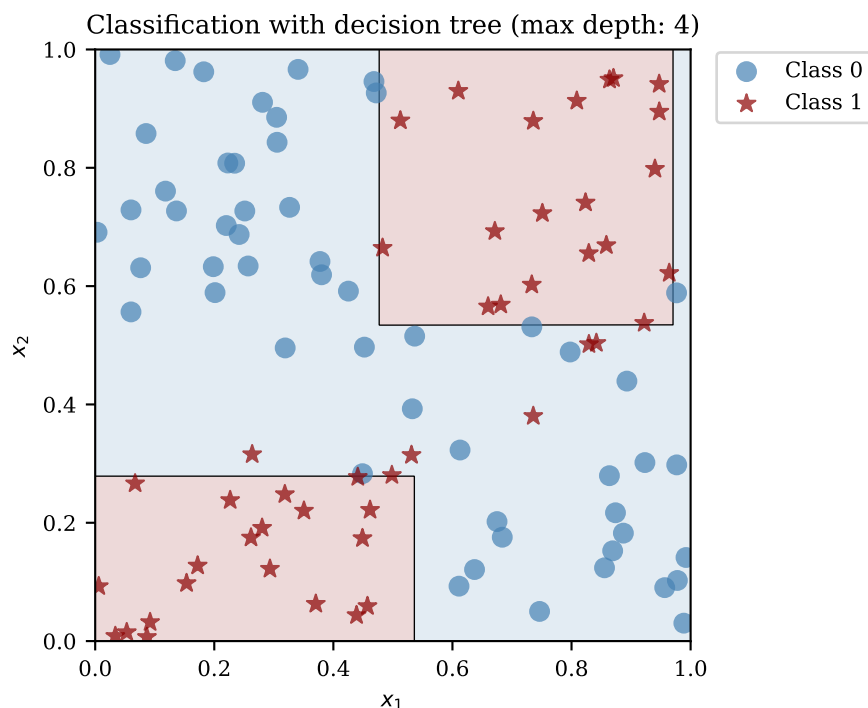
Best validation accuracy: 0.800

Best max depth: 4

The output reports the depth selected by cross-validation. We can plot the decision boundary of the tree refitted on all 100 training observations:

```
[42]: ax = plot_classes(X_train, y_train)
plot_decision_boundary(ax, xvalues, tree_cv)
_ = ax.set_title(
    f'Classification with decision tree (max depth: {tree_cv.best_params_["max_depth"]})'
)

```



The training metrics below show how closely the selected tree fits our sample. We will compare its predictions on new observations with those of the other classifiers in the final test comparison.

```
[43]: # Predicting using the best decision tree model
y_train_pred_tree = tree_cv.predict(X_train)

tabulate_training_metrics(y_train, y_train_pred_tree)

```

Training accuracy: 0.940

Training precision: 1.000

Training recall: 0.870

Training F1 score: 0.930

4.5 Random forest

A random forest classifier is an *ensemble method*, which means that we combine predictions from multiple models into a single prediction. Here, we fit multiple decision trees on bootstrap samples of the data, drawn with replacement, and average their predicted class probabilities. This model averaging is intended to lead to less overfitting and higher predictive accuracy. The random forest classifier is implemented in `RandomForestClassifier` in scikit-learn.

Just like decision trees, a random forest can capture nonlinear relationships without polynomial features and does not require feature standardization. We can therefore fit it directly on the original features.

The following hyperparameters control how the algorithm grows and combines the trees:

- `n_estimators`: the number of trees. Averaging more trees makes predictions more stable; beyond some point, additional trees change predictions little.
- `max_features`: the number of features randomly selected as candidates at each split. An integer specifies a count, so `1` considers one feature. A float specifies a fraction of all features, so `1.0` considers all features and `0.5` considers half (rounded down, with a minimum of one). The default `'sqrt'` uses the square root of the total number of features, rounded down; with our two features, it considers one.
- `max_depth`: the maximum tree depth. Smaller values limit complexity; `None` leaves the depth unrestricted.
- `min_samples_leaf`: the minimum number of observations in each leaf. Larger values prevent the tree from creating very small terminal groups.
- `min_samples_split`: the minimum number of observations a node needs before we consider splitting it. This does not set a minimum size for each child.

`max_depth`, `min_samples_leaf`, and `min_samples_split` also apply to a single decision tree. For example, `min_samples_split=10` can allow a node with ten observations to split into groups of one and nine. Setting `min_samples_leaf=5` rules out that split because both resulting groups must contain at least five observations.

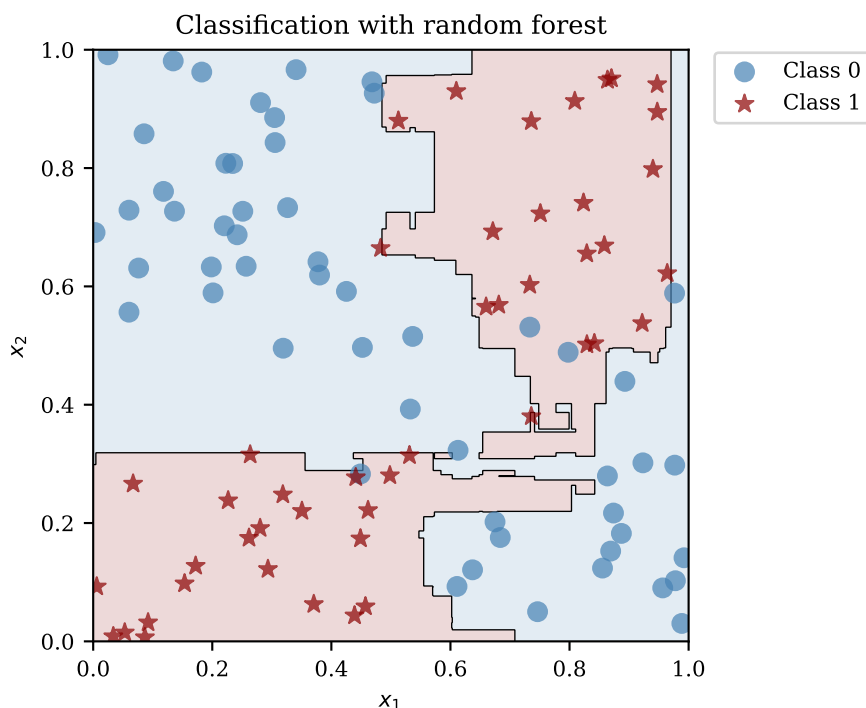
```
[44]: from sklearn.ensemble import RandomForestClassifier

# Create random forest classifier, fit on training data
forest = RandomForestClassifier(
    n_estimators=100, # Number of trees in the forest
    max_depth=3, # Maximum depth of each tree
    random_state=1234, # Random seed for bootstrapping and feature selection
    n_jobs=4, # Number of parallel jobs to run for both fit and predict
)

# Fit the random forest model on the training data
forest.fit(X_train, y_train)
```

Compared to a single decision tree, the random forest leads to more granular decision boundaries which are due to the averaging of many decision trees.

```
[45]: ax = plot_classes(X_train, y_train)
plot_decision_boundary(ax, xvalues, forest)
_ = ax.set_title('Classification with random forest')
```



We can compute the random forest's training accuracy, precision, recall, and F1 score in the same way:

```
[46]: y_train_pred_forest = forest.predict(X_train)

      tabulate_training_metrics(y_train, y_train_pred_forest)
```

```
Training accuracy:  0.960
Training precision: 0.938
Training recall:    0.978
Training F1 score:  0.957
```

We use `GridSearchCV()` to select the maximum tree depth and number of trees. We try depths from 5 to 15 and tree counts from 100 to 190 in steps of 10, giving $11 \times 10 = 110$ combinations. Each candidate uses the same five stratified folds and accuracy scoring rule as before.

We leave the other forest parameters at their defaults in this example. The hyperparameters listed above give us further ways to control the model in other applications.

Note that we set `n_jobs=1` when constructing the random forest. This is because the random forest itself can be parallelized, but we also parallelize the cross-validation. Setting `n_jobs=-1` for both would oversubscribe the CPU and slow down the computation.

```
[47]: # Fit each forest with one job; parallelize the grid search below
      forest_tune = RandomForestClassifier(random_state=1234, n_jobs=1)

      # Candidate ranges for depth and number of trees
      max_depth_grid = np.arange(5, 16)
      n_estimators_grid = np.arange(100, 200, 10)
      param_grid = {'max_depth': max_depth_grid, 'n_estimators': n_estimators_grid}

      # Create GridSearchCV object for hyperparameter tuning of the random forest
      forest_cv = GridSearchCV(
          estimator=forest_tune,
          param_grid=param_grid,
          cv=cv,
          scoring=scoring,
          refit=True,
          n_jobs=4,
```

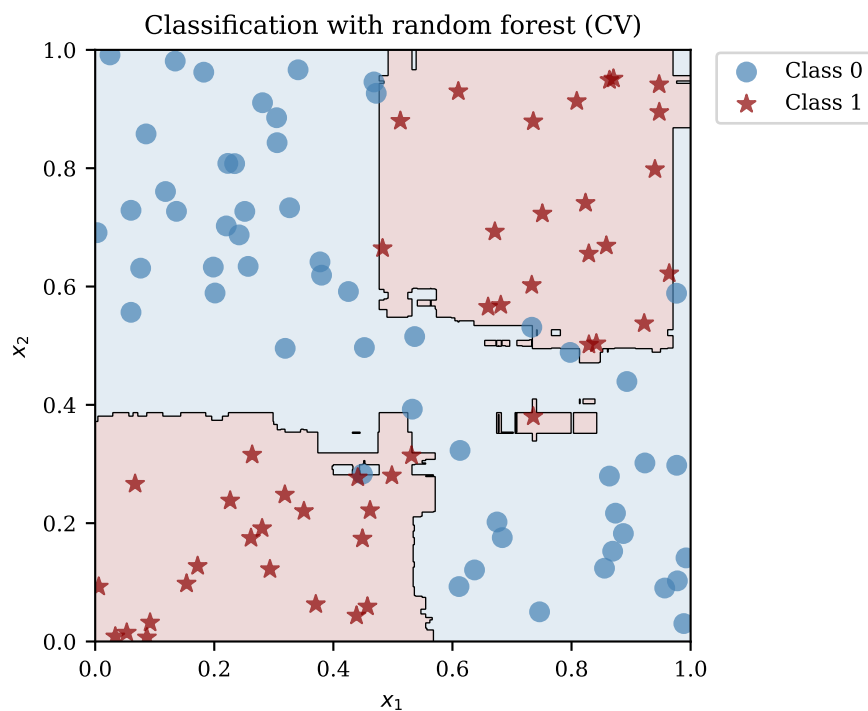
```
)

# Run cross-validation on the original, unscaled features
forest_cv.fit(X_train, y_train)
```

```
[48]: print(f'Best validation accuracy: {forest_cv.best_score_:.3f}')
      for param, value in forest_cv.best_params_.items():
          print(f'Best {param}: {value}')
```

```
Best validation accuracy: 0.880
Best max_depth: 10
Best n_estimators: 180
```

```
[49]: ax = plot_classes(X_train, y_train)
      plot_decision_boundary(ax, xvalues, forest_cv)
      _ = ax.set_title('Classification with random forest (CV)')
```



```
[50]: y_train_pred_forest_cv = forest_cv.predict(X_train)

      tabulate_training_metrics(y_train, y_train_pred_forest_cv)
```

```
Training accuracy: 1.000
Training precision: 1.000
Training recall: 1.000
Training F1 score: 1.000
```

Here, all four training metrics equal 1 because the refitted forest classifies every training observation correctly. Deep trees can isolate individual observations, including noisy labels, and averaging their predictions can preserve this fit.

In cross-validation, each forest predicts observations excluded from its training fold, which explains why validation accuracy is lower. A perfect training fit alone does not tell us whether the selected forest is too complex: we judge its predictive performance using cross-validation and the independent test sample below.

4.6 Comparing predictions on new observations

We have now fitted and tuned the main classifiers using the same 100 training observations. We generate 10,000 independent test observations from the same process, with the same noise level, using a separate random generator. This larger test sample makes the comparison less sensitive to chance variation in the test observations.

We use the fitted models directly to predict these new observations. The test sample has played no part in fitting the models or selecting hyperparameters.

```
[51]: # Generate a large independent test sample after fitting all main models
rng_test = np.random.default_rng(seed=5678)
X_test, y_test = create_class_data(N=10_000, sigma=sigma_eps, rng=rng_test)
```

The tables below compare the classifiers selected by cross-validation using accuracy, precision, recall, and F1 score for the training sample and the same independent test sample. The training table also includes the mean validation accuracy used to select hyperparameters in the CV ACC column. The CV scores describe models trained within the folds; the training and test metrics evaluate the final models refitted on all 100 training observations.

Here, the classes are approximately balanced and we count mistakes in both classes equally, so we use accuracy to select hyperparameters. We report all four metrics to show different aspects of model performance. We use these tables to assess the fitted models without changing their hyperparameters after seeing the test scores. A different training sample could change the ranking, even with a large test sample.

```
[52]: # Compare models selected by cross-validation
models = {
    'Logistic, CV': lr_cv,
    'Decision tree, CV': tree_cv,
    'Random forest, CV': forest_cv,
}

# Report the same four metrics for the training and test samples
samples = {'Training': (X_train, y_train), 'Test': (X_test, y_test)}

for sample_name, (X_sample, y_sample) in samples.items():
    print(f'\n{sample_name} sample:')
    header = (
        f'{"Model":<25} {"Accuracy":>10} {"Precision":>10} {"Recall":>10} '
        f'{"F1 score":>10}'
    )
    if sample_name == 'Training':
        header += f' {"CV ACC":>10}'
    print(header)
    for name, model in models.items():
        y_sample_pred = model.predict(X_sample)
        acc = accuracy_score(y_sample, y_sample_pred)
        pre = precision_score(y_sample, y_sample_pred)
        rec = recall_score(y_sample, y_sample_pred)
        f1 = f1_score(y_sample, y_sample_pred)

        row = f'{name:<25} {acc:10.3f} {pre:10.3f} {rec:10.3f} {f1:10.3f}'
        if sample_name == 'Training':
            cv_text = f'{model.best_score_:.3f}'
            row += f' {cv_text:>10}'
        print(row)
```

Training sample:

Model	Accuracy	Precision	Recall	F1 score	CV ACC
Logistic, CV	0.930	0.915	0.935	0.925	0.910

Decision tree, CV	0.940	1.000	0.870	0.930	0.800
Random forest, CV	1.000	1.000	1.000	1.000	0.880

Test sample:

Model	Accuracy	Precision	Recall	F1 score
Logistic, CV	0.842	0.874	0.796	0.833
Decision tree, CV	0.803	0.896	0.684	0.776
Random forest, CV	0.831	0.869	0.777	0.820

4.7 Support vector machines (SVM)

We now turn to support vector machines, which classify observations by finding a decision boundary with a wide *margin*, or buffer zone, between the classes. We balance the width of this margin against a penalty for observations that lie inside it or on the wrong side of the boundary. The observations that determine the fitted boundary are called *support vectors*. `scikit-learn` offers several implementations of SVMs that differ in the underlying algorithm and capabilities. We focus on the `SVC` (support vector classifier) class which supports both linear and nonlinear kernels.

Just like logistic regression, `SVC` uses the regularization parameter C . However, SVMs offer additional flexibility in that they support various kernels. A kernel measures similarity between observations in a way that lets the SVM work implicitly in a different feature space. For example, we might have a classification problem like the one we studied above, where the classes 0 and 1 clearly cannot be separated by a single linear boundary. Instead of adding polynomial interactions, with SVMs we can specify a nonlinear kernel in the hope that once the features have been represented in this implicit feature space, the individual classes become much easier to separate.

4.7.1 Linear kernel

We use exactly the same 100 training observations and 10,000 independent test observations as for logistic regression, decision trees, and random forests. We keep the same noise level and reuse the same stratified CV folds, so the results are directly comparable.

We first fit the SVM specifications and examine their training predictions. After tuning, we extend the earlier test comparison with the tuned RBF SVM. We start with the simplest linear kernel which is selected using the `kernel='linear'` argument.

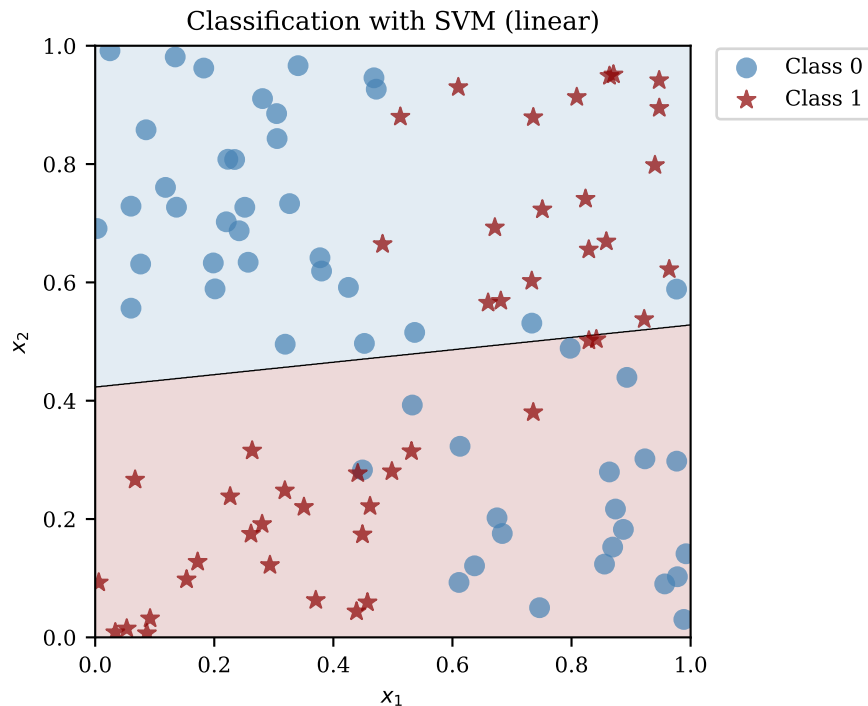
```
[53]: from sklearn.svm import SVC

# Create pipeline with linear SVM
pipe_svm = make_pipeline(StandardScaler(), SVC(kernel='linear', C=1.0))

# Fit the linear SVM model on the training data
pipe_svm.fit(X_train, y_train)
```

Plotting the decision boundaries, we see that the linear kernel leads to a single linear decision boundary:

```
[54]: xvalues = np.linspace(0, 1, 1000)
ax = plot_classes(X_train, y_train)
plot_decision_boundary(ax, xvalues, pipe_svm)
_ = ax.set_title('Classification with SVM (linear)')
```



We can compute training accuracy, precision, recall, and F1 score for this linear decision boundary. As with linear logistic regression, the model has limited flexibility for our nonlinear classification problem:

```
[55]: y_train_pred_svm = pipe_svm.predict(X_train)

tabulate_training_metrics(y_train, y_train_pred_svm)
```

```
Training accuracy: 0.590
Training precision: 0.553
Training recall: 0.565
Training F1 score: 0.559
```

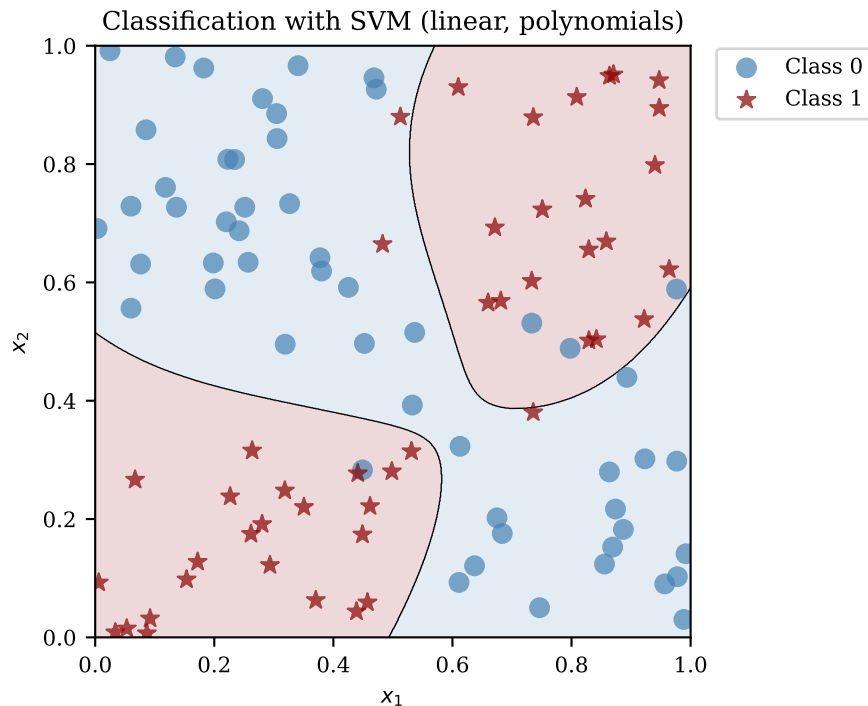
As with logistic regression, we can easily augment this model by adding polynomial interactions of features:

```
[56]: # Maximum polynomial degree
degree = 5

# Create pipeline with polynomial features and SVM
pipe_svm_poly = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(degree=degree, include_bias=False),
    StandardScaler(),
    SVC(kernel='linear', C=1.0),
)

# Fit the polynomial SVM model on the training data
pipe_svm_poly.fit(X_train, y_train)
```

```
[57]: xvalues = np.linspace(0, 1, 1000)
ax = plot_classes(X_train, y_train)
plot_decision_boundary(ax, xvalues, pipe_svm_poly)
_ = ax.set_title('Classification with SVM (linear, polynomials)')
```



We compute training accuracy, precision, recall, and F1 score to assess the fit of this more flexible model:

```
[58]: y_train_pred_svm = pipe_svm_poly.predict(X_train)
      tabulate_training_metrics(y_train, y_train_pred_svm)
```

```
Training accuracy:  0.930
Training precision: 0.915
Training recall:    0.935
Training F1 score:  0.925
```

As expected, the polynomial features allow a more flexible decision boundary and lead to a better fit to the training observations.

4.7.2 Gaussian kernel

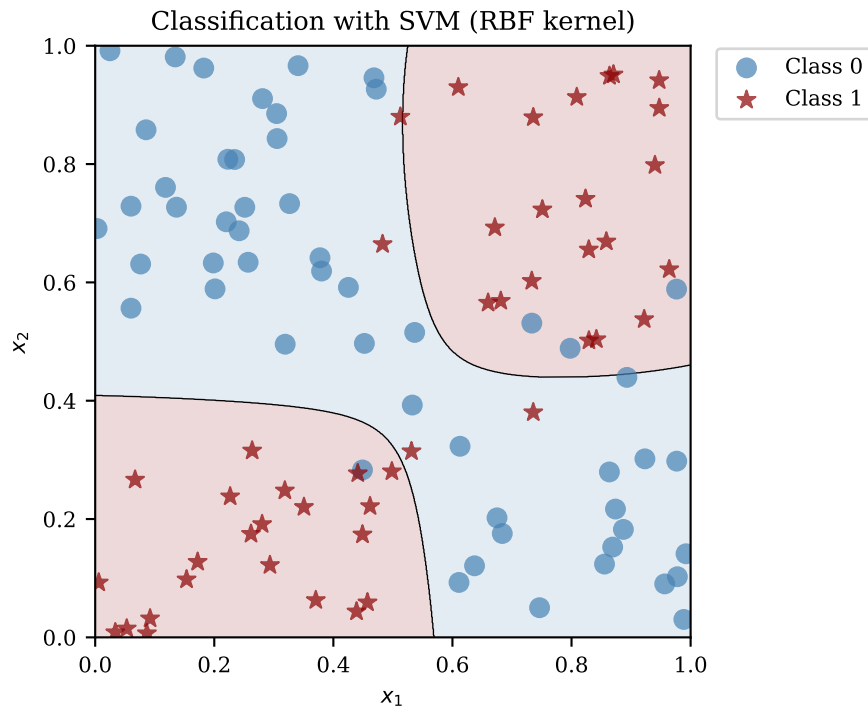
However, it is not necessary to introduce polynomial features with SVM. Alternatively, we can operate with just the two original features, but choose a different kernel. By default, the kernel used by SVC is `kernel='rbf'` which stands for the *radial basis function*. This kernel is also called the Gaussian kernel.

We rerun the above code, omitting the polynomial features but instead selecting the Gaussian kernel:

```
[59]: # Create SVM with RBF kernel
      pipe_svm_rbf = make_pipeline(StandardScaler(), SVC(kernel='rbf', C=1.0, gamma='scale'))

      # Fit the SVM model with RBF kernel on the training data
      pipe_svm_rbf.fit(X_train, y_train)
```

```
[60]: ax = plot_classes(X_train, y_train)
      plot_decision_boundary(ax, xvalues, pipe_svm_rbf)
      _ = ax.set_title('Classification with SVM (RBF kernel)')
```



We can compute the same four training metrics for the Gaussian kernel. After tuning, the final test comparison will show how the selected RBF model performs on new observations:

```
[61]: y_train_pred_svm = pipe_svm_rbf.predict(X_train)

      tabulate_training_metrics(y_train, y_train_pred_svm)
```

```
Training accuracy: 0.920
Training precision: 0.913
Training recall: 0.913
Training F1 score: 0.913
```

So far, we have used fixed SVM hyperparameters. For the RBF kernel, we now tune both:

1. C , the inverse regularization penalty strength; and
2. γ , which scales the squared Euclidean distance between feature vectors.

We use `GridSearchCV()` to evaluate all combinations of 50 values of C and 50 values of γ . As with the other classifiers, we use the same five shuffled, stratified folds and accuracy scoring rule.

```
[62]: pipe_svm_rbf = make_pipeline(StandardScaler(), SVC(kernel='rbf'))

      # Define range of values for C and gamma, spaced uniformly in logs
      param_range = np.logspace(-1, 1, 50)

      # Define param_grid argument for GridSearchCV
      param_grid = {'svc__C': param_range, 'svc__gamma': param_range}

      # Create grid search cross-validation object, run cross-validation
      svm_rbf_cv = GridSearchCV(
          estimator=pipe_svm_rbf, # Model to tune
          param_grid=param_grid, # Dictionary of hyperparameter grids
          cv=cv, # Same stratified folds as for the other classifiers
          scoring=scoring, # Same scoring rule as for the other classifiers
          refit=True, # Refit best model
          n_jobs=4, # Number of parallel jobs
      ).fit(X_train, y_train)
```

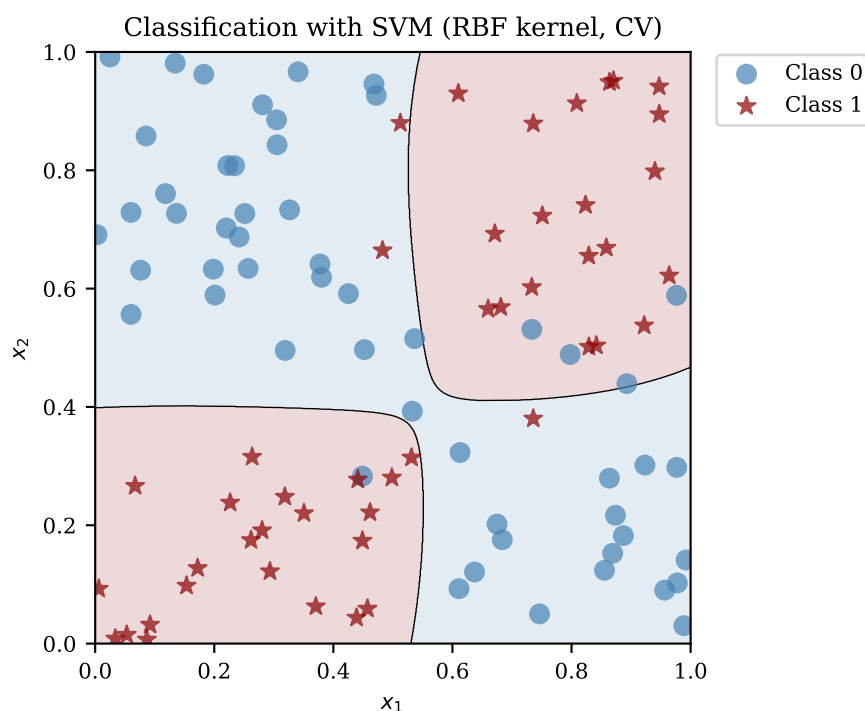
The best accuracy score and the optimal parameter values can be retrieved from the attributes `best_score_` and `best_params_`:

```
[63]: print(f'Best validation accuracy: {svm_rbf_cv.best_score_:.3f}')
      print(f'Best C: {svm_rbf_cv.best_params_["svc__C"]:.3f}')
      print(f'Best gamma: {svm_rbf_cv.best_params_["svc__gamma"]:.3f}')
```

```
Best validation accuracy: 0.930
Best C: 6.866
Best gamma: 0.373
```

Finally, we can plot the decision boundaries for the cross-validated SVM model:

```
[64]: ax = plot_classes(X_train, y_train)
      plot_decision_boundary(ax, xvalues, svm_rbf_cv)
      _ = ax.set_title('Classification with SVM (RBF kernel, CV)')
```



We now extend the earlier comparison with the tuned RBF SVM. The ranking describes this particular training sample and may change with a different draw.

```
[65]: # Copy the main comparison and add the SVM selected by cross-validation
      models_with_svm = models.copy()
      models_with_svm['SVM, RBF CV'] = svm_rbf_cv

      # Report the same four metrics for the training and test samples
      samples = {'Training': (X_train, y_train), 'Test': (X_test, y_test)}
      for sample_name, (X_sample, y_sample) in samples.items():
          print(f'\n{sample_name} sample:')
          header = (
              f'{"Model":<25} {"Accuracy":>10} {"Precision":>10} {"Recall":>10} '
              f'{"F1 score":>10}'
          )
          if sample_name == 'Training':
              header += f' {"CV ACC":>10}'
          print(header)
          for name, model in models_with_svm.items():
              y_sample_pred = model.predict(X_sample)
```

```

acc = accuracy_score(y_sample, y_sample_pred)
pre = precision_score(y_sample, y_sample_pred)
rec = recall_score(y_sample, y_sample_pred)
f1 = f1_score(y_sample, y_sample_pred)

row = f'{name:<25} {acc:10.3f} {pre:10.3f} {rec:10.3f} {f1:10.3f}'
if sample_name == 'Training':
    cv_text = f'{model.best_score_:.3f}'
    row += f' {cv_text:>10}'
print(row)

```

Training sample:

Model	Accuracy	Precision	Recall	F1 score	CV ACC
Logistic, CV	0.930	0.915	0.935	0.925	0.910
Decision tree, CV	0.940	1.000	0.870	0.930	0.800
Random forest, CV	1.000	1.000	1.000	1.000	0.880
SVM, RBF CV	0.920	0.896	0.935	0.915	0.930

Test sample:

Model	Accuracy	Precision	Recall	F1 score
Logistic, CV	0.842	0.874	0.796	0.833
Decision tree, CV	0.803	0.896	0.684	0.776
Random forest, CV	0.831	0.869	0.777	0.820
SVM, RBF CV	0.842	0.854	0.824	0.838

The tuned SVM achieves similar test accuracy to logistic regression, with higher recall and a slightly higher F1 score in this example.

4.8 List of functions and classes used in this lecture

The following list contains the central functions and classes covered in this lecture. Each name is linked to the official API documentation, which you can consult for more details regarding arguments and return values. The “go to section” links lead to the relevant section in this notebook.

4.8.1 Scikit-learn classification models

- [LogisticRegression\(\)](#): Class for fitting logistic regression models. [\[go to section\]](#)
- [LogisticRegressionCV\(\)](#): Class for fitting logistic regression models with built-in cross-validation. [\[go to section\]](#)
- [DecisionTreeClassifier\(\)](#): Class for fitting decision tree classifiers. [\[go to section\]](#)
- [RandomForestClassifier\(\)](#): Class for fitting random forest classifiers. [\[go to section\]](#)
- [SVC\(\)](#): Class for fitting support vector machine classifiers. [\[go to section\]](#)

4.8.2 Scikit-learn metrics and visualization

- [accuracy_score\(\)](#): Function for computing classification accuracy. [\[go to section\]](#)
- [precision_score\(\)](#): Function for computing precision. [\[go to section\]](#)
- [recall_score\(\)](#): Function for computing recall. [\[go to section\]](#)
- [f1_score\(\)](#): Function for computing the F1 score. [\[go to section\]](#)
- [ConfusionMatrixDisplay](#): Class for visualizing confusion matrices. [\[go to section\]](#)
- [plot_tree\(\)](#): Function for visualizing decision trees. [\[go to section\]](#)

4.8.3 Scikit-learn model selection and evaluation

- `train_test_split()`: Function for demonstrating a stratified training and test split. [\[go to section\]](#)
- `GridSearchCV()`: Class for performing grid search with cross-validation. [\[go to section\]](#)
- `StratifiedKFold()`: Class for defining folds with approximately equal class proportions. [\[go to section\]](#)
- `validation_curve()`: Function for computing validation curves. [\[go to section\]](#)