Lecture 12: Models for regression and classification

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn

NHH Norwegian School of Economics

April 1, 2025

See GitHub repository for notebooks and data:

https://github.com/richardfoltyn/FIE463-V25

Contents

1 Linear regression models

1.1 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,

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

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

$$(\widehat{\mu}, \widehat{\boldsymbol{\beta}}) = \operatorname*{arg\,min}_{\mu, \boldsymbol{\beta}} L(\mu, \boldsymbol{\beta})$$

While OLS is the best linear unbiased estimator (BLUE) under the usual assumption about the errors ϵ_i, the estimated $(\widehat{\mu}, \widehat{\boldsymbol{\beta}})$ 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 which prevent overfitting and improve out-of-sample prediction, thus making these models more suitable for machine learning applications.

1.2 Ridge regression

The first model we study is the Ridge regression, which estimates a linear model but adds a penalty term to the loss function:

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

Compared to ordinary least squares (OLS) we discussed initially, the penalty term increases the loss, in particular if the estimated coefficients $\boldsymbol{\beta}$ are large. This term is called an L2 penalty because it corresponds to the (squared) L2 vector norm. In many textbooks, you will see the penalty term written as $\alpha\|\boldsymbol{\beta}\|_2^2$ which is equivalent to the formula used above.

For any $\alpha > 0$, the loss function is increasing in the (absolute) coefficient values, thus the minimum L might be one where the elements of $\boldsymbol{\beta}$ are smaller than what they would be with OLS. We therefore say that the Ridge regression applies *regularization* or *shrinkage*. Note, however, that the intercept μ is not included in the penalty term, and thus no regularization is applied to it.

Clearly, the regularization strength depends on the value of α. For tiny (or zero) α, the estimated $\widehat{\boldsymbol{\beta}}$ will be close (or identical) to OLS, while for large α the estimated coefficients will be compressed towards zero. In this setting, α is a hyperparameter and we can accordingly use cross-validation to find an "optimal" value.

Why would we ever want to use Ridge regression given that OLS is the best linear unbiased estimator? It turns out that regularization can help in scenarios where we have a large number of (potentially multicollinear) regressors in which case OLS is prone to overfitting.

1.2.1 Example: Polynomial approximation

We illustrate such problems and the benefits of Ridge regression using a highly stylized example. Imagine that we want to approximate a (non-linear) trigonometric function using a high-order polynomial, a setting which is notoriously susceptible to overfitting (also see the previous lecture and workshop for illustrations). Assume that our model is given by

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

where ϵ_i as an additive, normally-distributed measurement error term with mean 0 and variance $\frac{1}{4}$.

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

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

def compute_true_y(x):
    """
    True function (w/o errors)
    """
    return np.cos(1.5 * np.pi * x)
```

The following code creates a demo sample with $N = 200$ observations.

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

def create_trig_sample(N=200, sigma=0.5, rng=None):
    """
    Create sample data for Ridge and Lasso.

    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=1234)

    # Randomly draw explanatory variable x uniformly distributed on [0, 1]
    x = rng.uniform(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
```

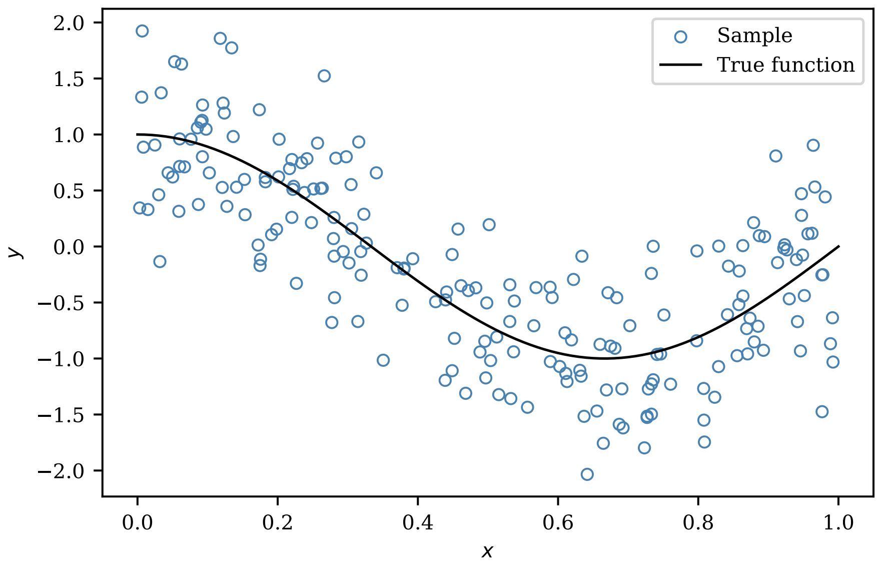
[3]:
```python
x, y = create_trig_sample()
```

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]:
```python
import matplotlib.pyplot as plt


def plot_trig_sample(x, y):
    """
    Plot a sample for Ridge and Lasso
    """
    # Sample scatter plot
    fig, ax = plt.subplots(1, 1, figsize=(5.5, 3.5))
    ax.scatter(x, y, s=20, c='none', edgecolor='steelblue', lw=0.75, label='Sample')
    # Plot true relationship
    xvalues = np.linspace(0.0, 1.0, 101)
    y_true = compute_true_y(xvalues)
    ax.plot(xvalues, 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, y)
```

[5]:
```
<Axes: xlabel='$x$', ylabel='$y$'>
```



1.2.2 Estimating the Ridge and linear regression models

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 $K = 15$ since we anticipate that this leads to problems with OLS.

A few more steps are required before we can run the Ridge regression:

- We need to create the polynomial in $x$ which we do with the `PolynomialFeatures` transformation we already encountered.
- Moreover, regularization methods can be susceptible to scaling issues, so we need to demean and normalize all input variables, i.e., we make sure that each feature has mean 0 and a variance of 1. We can automate this step using the `StandardScaler` transformation.
- Finally, the estimation step is performed by the `Ridge` class.
- We build a `Pipeline` that combines these transformation together with the estimation step. We use the function `make_pipeline()` to simplify this step.

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

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

# Polynomial degree
degree = 15

# 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(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler(),                          # standardize features
    Ridge(alpha=3.0, fit_intercept=True)       # Fit Ridge regression
)

# Make sure X is a matrix
X = x[:, None]

# Fit model
pipe_ridge.fit(X, y)
```

[6]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(degree=15, include_bias=False)),
                ('standardscaler', StandardScaler()),
                ('ridge', Ridge(alpha=3.0))])
```

It is instructive to estimate the same model using linear regression and compare the results:

[7]:
```
from sklearn.linear_model import LinearRegression

# Create pipeline for linear model (linear regression does not require
# standardization!)
pipe_lr = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    LinearRegression(fit_intercept=True)
)

# Make sure X is a matrix
X = x[:, None]

# Fit model
pipe_lr.fit(X, y)
```

[7]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(degree=15, include_bias=False)),
                ('linearregression', LinearRegression())])
```

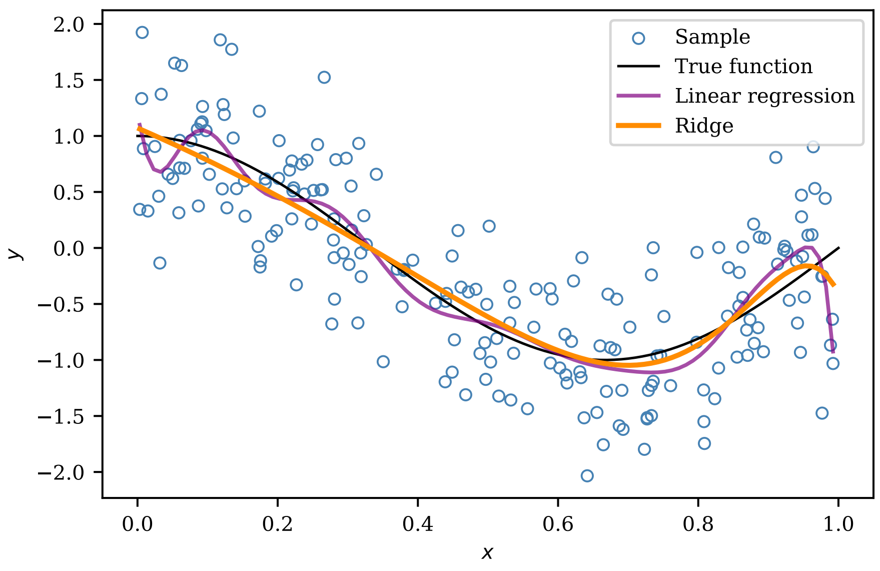
To illustrate the difference for this artificial example, we add the model predictions from the Ridge and linear regressions to the sample scatter plot we created earlier.

[8]:
```
# Grid on which to evaluate true model and predictions
xvalues = np.linspace(np.amin(x), np.amax(x), 100)

# Predicted values from linear regression
y_pred_lr = pipe_lr.predict(xvalues[:, None])
```

```
# Predicted values from Ridge regression
y_pred_ridge = pipe_ridge.predict(xvalues[:, None])
```

[9]:
```
# Plot sample and true relationship
ax = plot_trig_sample(x, y)

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

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

ax.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x7f0f3ad77020>
```



As the graph shows, the OLS model vastly overfits the data which is what we would expect in this settings. Conversely, the prediction of the Ridge regression is reasonably close to the true function and much better behaved despite the high polynomial degree used here.

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

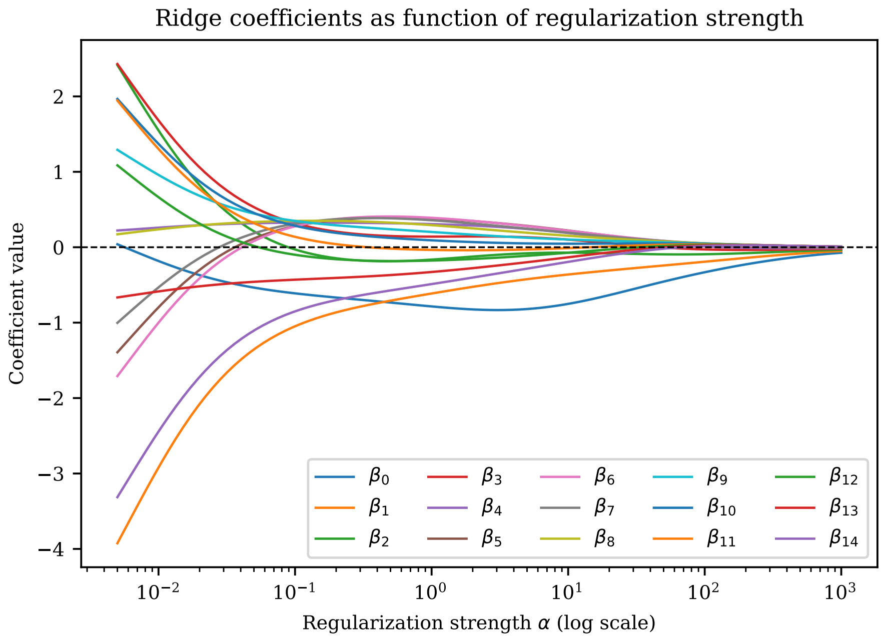
[10]:
```
# Create grid of alphas spaced uniformly in logs on [5e-3, 1000]
alphas = np.logspace(start=np.log10(5.0e-3), stop=3, num=100)

# Re-create pipeline w/o Ridge estimator, estimation step differs for each alpha
transform = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler()
)

# Create polynomial features
X_trans = transform.fit_transform(x[:, None])

# Array to store coefficients for all alphas
```

```
coefs = np.empty((len(alphas), X_trans.shape[1]))

# Estimate Ridge for each alpha, store fitted coefficients
for i, alpha in enumerate(alphas):
    ridge = Ridge(alpha=alpha, fit_intercept=True)

    # Fit model for given alpha
    ridge.fit(X_trans, y)

    coefs[i] = ridge.coef_
```

The following plot shows each coefficient (one color corresponds to one coefficients) for different values of $\alpha$ on the x-axis. Note that the x-axis is plotted on a $\log_{10}$ scale which allows us to zoom in on smaller values of $\alpha$.

[11]:
```
plt.figure(figsize=(6,4))

# Plot coefficient arrays against penalty strength
plt.plot(alphas, coefs, lw=1.0)

plt.xscale('log', base=10)
plt.axhline(0.0, ls='--', lw=0.75, c='black')
plt.xlabel(r'Regularization strength $\alpha$ (log scale)')
plt.ylabel('Coefficient value')
plt.title('Ridge coefficients as function of regularization strength')
plt.legend([rf'$\beta_{{{i}}}$' for i in range(degree)], ncols=5, loc='lower right')
```

[11]:
```
<matplotlib.legend.Legend at 0x7f0f3846d340>
```



For small $\alpha$ (on the left) the estimated coefficients are close to the (standardised) OLS coefficients, but their values shrink towards zero as $\alpha$ increases. For a very large $\alpha = 10^3$ the estimated coefficients are basically zero since the penalty dominates the sum of squared errors in the loss function.

1.2.3 Tuning the regularization parameter via cross-validation

In the previous example, we picked an arbitrary regularization strength $\alpha$ when fitting the Ridge regression. In applied work, we would want to tune $\alpha$ (which is a hyperparameter) using cross-validation instead. To this end, we could use the generic cross-validation functionality we studied in the previous lecture since that one works for all types of estimators. However, `scikit-learn` implements a cross-validation class specifically for Ridge regression called `RidgeCV` which we use in the code below to find an optimal $\alpha$.

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

# RidgeCV does not support pipelines, so we need to transform x before
# cross-validation.
transform = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler()
)

# Create standardized polynomial features
X_trans = transform.fit_transform(x[:, None])

# Candidate alphas on used for cross-validation.
# Spaced uniformly in logs to get denser grid for small alphas.
N_alphas = 100
alphas = np.logspace(start=np.log10(1.0e-5), stop=np.log10(5), num=N_alphas)

# Create and run Ridge cross-validation
rcv = RidgeCV(alphas=alphas, store_cv_results=True).fit(X_trans, y)
```

Note that we don't need to specify the number of folds to be used for cross-validation (this defaults to `cv=None` when creating `RidgeCV`). In this case, `RidgeCV` performs the more efficient leave-on-out cross-validation instead of splitting the sample into `n` folds.

By default, `RidgeCV` uses the (negative) mean squared error (MSE) to find the best $\alpha$ which we can then recover from the `alpha_` attribute.

[13]:
```
# Recover best alpha that minimizes MSE
alpha_best = rcv.alpha_

# Best MSE is stored as negative score!
MSE_best = - rcv.best_score_

print(f'Best alpha: {alpha_best:.3g} (MSE: {MSE_best:.3g})')
```

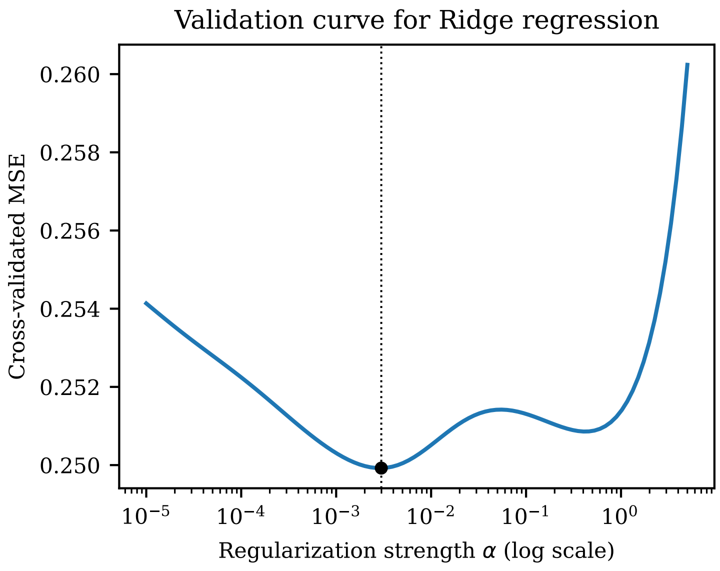
```
Best alpha: 0.00299 (MSE: 0.25)
```

Because we fitted `RidgeCV` with the argument `store_cv_results=True`, the fitted object stores the MSE for each `alpha` in `cv_results_` which we can use to plot the MSE as a function of $\alpha$. The dimension of `cv_results_` is `(Nobs, Nalphas)`. Note that this attribute is only available with leave-one-out cross-validation, but not when manually specifying the number of cross-validation folds.

[14]:
```
# Compute average MSE for each alpha value across all leave-one-out splits
mse_mean = np.mean(rcv.cv_results_, axis=0)

# Index of MSE-minimizing alpha
imin = np.argmin(mse_mean)
```

[15]:
```
import matplotlib.pyplot as plt

# Plot MSE against alphas, highlight minimum MSE
plt.plot(alphas, mse_mean)
plt.xlabel(r'Regularization strength $\alpha$ (log scale)')
plt.ylabel('Cross-validated MSE')
plt.scatter(alphas[imin], mse_mean[imin], s=15, c='black', zorder=100)
plt.axvline(alphas[imin], ls=':', lw=0.75, c='black')
plt.xscale('log')
plt.title('Validation curve for Ridge regression')
```

[15]:
```
Text(0.5, 1.0, 'Validation curve for Ridge regression')
```



Now that we have identified the optimal $\alpha$, we can re-fit the Ridge regression and plot the prediction from this model. Note that this is not strictly necessary as the return value of `RidgeCV` can be used to do prediction based on coefficients estimated for the best $\alpha$, but because `RidgeCV` does not support pipelines, we'd have to apply any transformations manually before doing so.

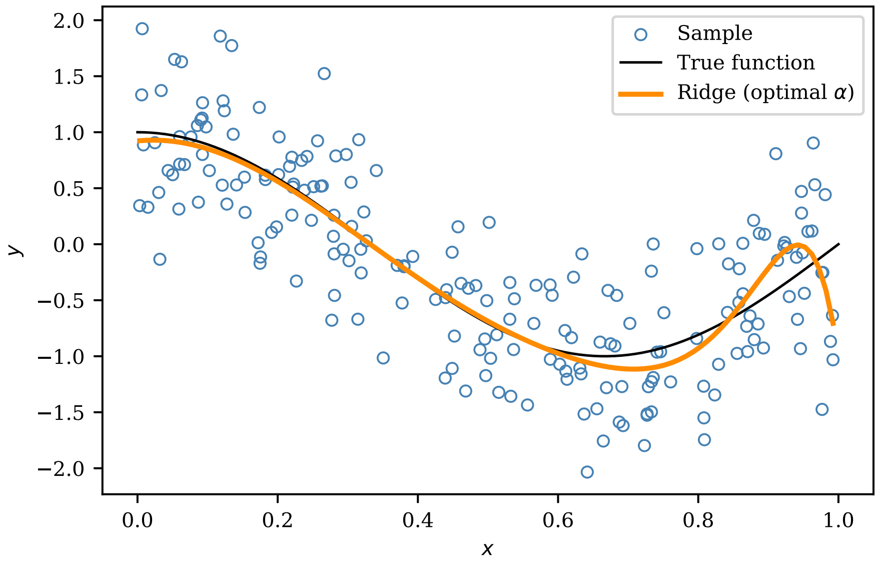
[16]:
```
# Create pipeline with Ridge estimator using optimal alpha
pipe_ridge = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler(),
    Ridge(alpha=alpha_best, fit_intercept=True)
)

# Fit Ridge regression with optimal alpha
pipe_ridge.fit(x[:, None], y)
```

[16]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(degree=15, include_bias=False)),
                ('standardscaler', StandardScaler()),
                ('ridge', Ridge(alpha=np.float64(0.0029875157525272996)))])
```

[17]:
```
# Grid on which to evaluate true model and predictions
xvalues = np.linspace(np.amin(x), np.amax(x), 100)

# Predicted values from Ridge regression
y_pred = pipe_ridge.predict(xvalues[:, None])
```

[18]:
```
# Plot sample and true relationship
ax = plot_trig_sample(x, y)

# Plot predicted values from cross-validated Ridge regression
ax.plot(xvalues, y_pred, c='darkorange', lw=2.0, label=r'Ridge (optimal $\alpha$)')
ax.legend()
```

[18]:
```
<matplotlib.legend.Legend at 0x7f0f307552e0>
```



Your turn. Rerun the whole Ridge example with a smaller sample size of N=50. What happens to the optimal cross-validated penalty parameter α?

1.3 Lasso

Another widely used estimator with shrinkage is LASSO (least absolute shrinkage and selection operator) which adds an L1 penalty term to the objective function:

$$L(\mu, \boldsymbol{\beta}) = \underbrace{\sum_{i=1}^{N} \Big(y_i - \mu - \mathbf{x}_i' \boldsymbol{\beta} \Big)^2}_{\text{Sum of squared errors}} + \alpha \underbrace{\sum_{k=1}^{K} |\beta_k|}_{\text{L1 penalty}}$$

As with the Ridge regression, this additional term penalizes large coefficient values. This term is called an L1 penalty because it corresponds to the L1 vector norm which can equivalently be written as $\alpha \|\boldsymbol{\beta}\|_1$.

While the objective looks very similar to Ridge regression, using the L1 instead of the L2 norm can produce much more parsimonious models because many coefficients end up being exactly zero and the corresponding features are thus eliminated from the model. We will see this in the example below.

1.3.1 Example: Polynomial approximation

We apply Lasso to the same random sample as in the section on Ridge which allows us to compare the two methods. The following code recreates that data, making the same functional form and distributional assumptions as in the previous section.

[19]:
```
x, y = create_trig_sample()
```

1.3.2 Estimating the Lasso and linear regression models

As with Ridge, we need to standardise the explanatory variables before fitting Lasso. The code below repeats these steps, but we now use `Lasso` to perform the model estimation. For now, we set the regularization strength to $\alpha = 0.015$ and will use cross-validation to find the optimal value later.

Note that for Lasso it might be necessary to increase the number of iterations by increasing the `max_iter` parameter (from the default of 1,000).

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

      # Polynomial degree
      degree = 15

      # 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(
          PolynomialFeatures(
              degree=degree,
              include_bias=False
          ),
          StandardScaler(),
          Lasso(alpha=0.0075, fit_intercept=True, max_iter=10000)
      )

      # Make sure X is a matrix
      X = x[:, None]

      pipe_lasso.fit(X, y)
```

```
[20]: Pipeline(steps=[('polynomialfeatures',
                       PolynomialFeatures(degree=15, include_bias=False)),
                      ('standardscaler', StandardScaler()),
                      ('lasso', Lasso(alpha=0.0075, max_iter=10000))])
```

For completeness, let's also recreate the linear regression estimation and plot the model predictions alongside the Lasso.

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

      # Create pipeline for linear model (linear regression does not require
      # standardization!)
      pipe_lr = make_pipeline(
          PolynomialFeatures(
              degree=degree,
              include_bias=False
          ),
          LinearRegression(fit_intercept=True)
      )

      # Make sure X is a matrix
      X = x[:, None]

      pipe_lr.fit(X, y)
```

```
[21]: Pipeline(steps=[('polynomialfeatures',
                       PolynomialFeatures(degree=15, include_bias=False)),
                      ('linearregression', LinearRegression())])
```

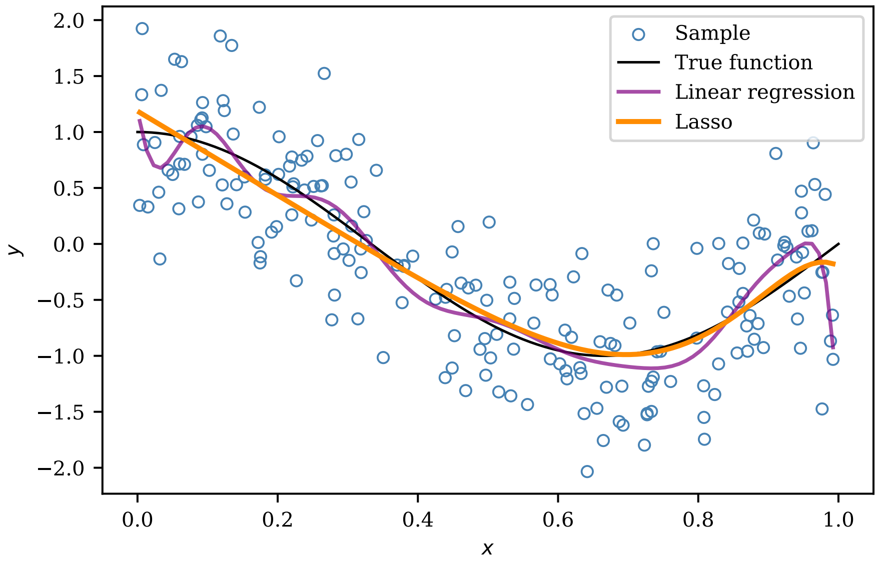
The following code plots the sample data, the true functional relationship, and the predictions from the linear regression and Lasso models.

```
[22]: # Grid on which to evaluate true model and predictions
      xvalues = np.linspace(np.amin(x), np.amax(x), 100)

      # Predicted values from linear regression
```

```
y_pred_lr = pipe_lr.predict(xvalues[:, None])

# Predicted values from Lasso regression
y_pred_lasso = pipe_lasso.predict(xvalues[:, None])
```

[23]:
```
ax = plot_trig_sample(x, y)

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

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

ax.legend()
```

[23]:
```
<matplotlib.legend.Legend at 0x7f0f3066c350>
```



You might be wondering why we chose $\alpha = 0.0075$ whereas we initially used $\alpha = 3$ for the Ridge regression. The reason is that the `scikit-learn` implementation of Lasso uses a slightly different loss function than the one given above, namely

$$L(\mu, \boldsymbol{\beta}) = \frac{1}{2N} \sum_{i=1}^{N} \Big(y_i - \mu - \mathbf{x}_i' \boldsymbol{\beta} \Big)^2 + \alpha \|\boldsymbol{\beta}\|_1$$

which scales the sum of squared errors term by a factor of of $(2N)^{-1}$. This makes no difference for the optimization, but changes the interpretation of the regularization strength α compared to Ridge regression. For our sample size of $N = 200$, an α_{Ridge} used for the `Ridge` estimator approximately corresponds to $\alpha_{Lasso} = \frac{\alpha_{Ridge}}{400}$ when plugged into `Lasso`.

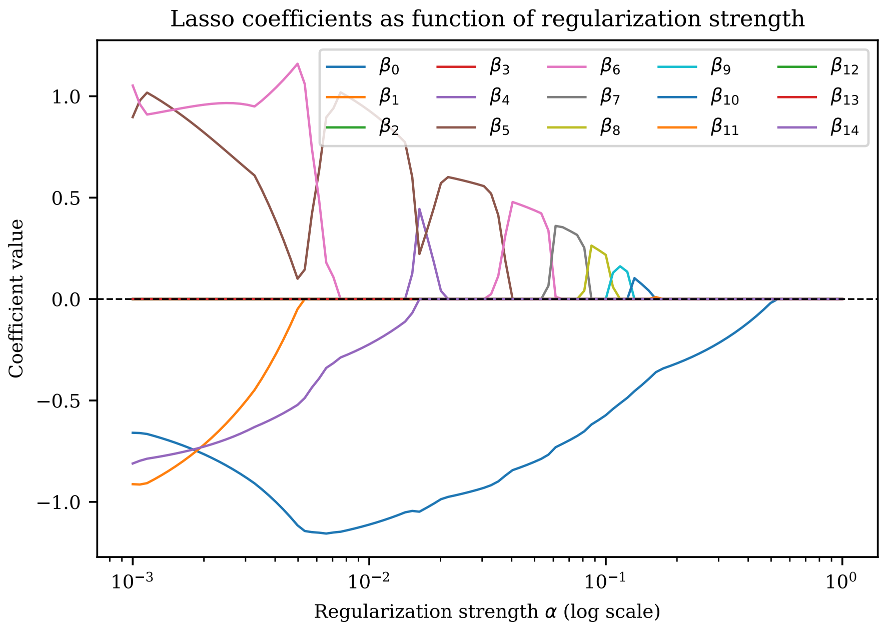
We next fit the model for different values of α on the interval $[10^{-3}, 1]$ where we again space the α uniformly in logs.

[24]:
```
# Create grid of alphas spaced uniformly in logs
alphas = np.logspace(start=np.log10(1.0e-3), stop=np.log10(1.0), num=100)

# Re-create pipeline w/o Lasso estimator, estimation step differs for each alpha
transform = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler()
)

# Create polynomial features
X_trans = transform.fit_transform(x[:, None])

# Array to store coefficients for all alphas
coefs = np.empty((len(alphas), X_trans.shape[1]))

# Estimate Lasso for each alpha, store fitted coefficients
for i, alpha in enumerate(alphas):
    lasso = Lasso(alpha=alpha,
        fit_intercept=True,
        max_iter=100_000
    )

    # Fit model for given alpha
    lasso.fit(X_trans, y)

    coefs[i] = lasso.coef_
```

Plotting the fitted coefficients against $\alpha$ on a log scale looks as follows:

[25]:
```
plt.figure(figsize=(6, 4))
plt.plot(alphas, coefs, lw=1.0)
plt.xscale('log', base=10)
plt.axhline(0.0, ls='--', lw=0.75, c='black')
plt.xlabel(r'Regularization strength $\alpha$ (log scale)')
plt.ylabel('Coefficient value')
plt.title('Lasso coefficients as function of regularization strength')
plt.legend([rf'$\beta_{{{i}}}$' for i in range(degree)], ncols=5)
```

[25]:
```
<matplotlib.legend.Legend at 0x7f0f306f8440>
```

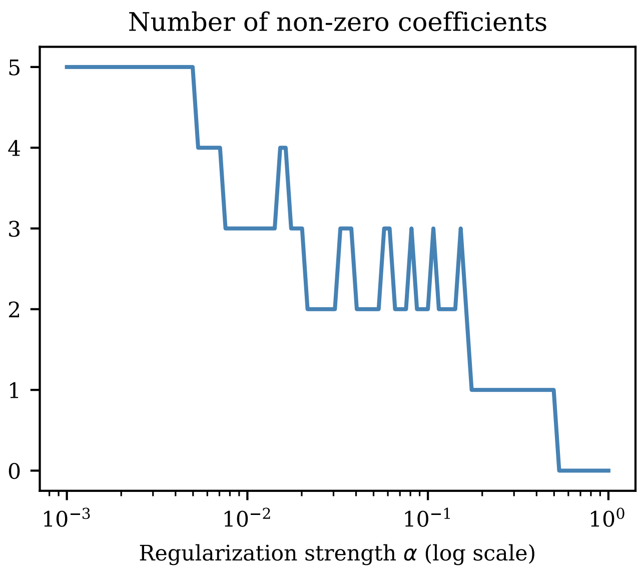


The graph shows that the coefficient estimates are quite different than what we obtained from the Ridge estimator. In fact, most of them are exactly zero for most values of $\alpha$. We highlight this result in the graph below which plots the number of non-zero coefficients against $\alpha$.

[26]:
```
# Number of non-zero coefficients for each alpha.
nonzero = np.sum(np.abs(coefs) > 1.0e-6, axis=1).astype(int)
```

[27]:
```python
# Plot number of non-zero 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 non-zero coefficients')
```

[27]:
```
Text(0.5, 1.0, 'Number of non-zero coefficients')
```



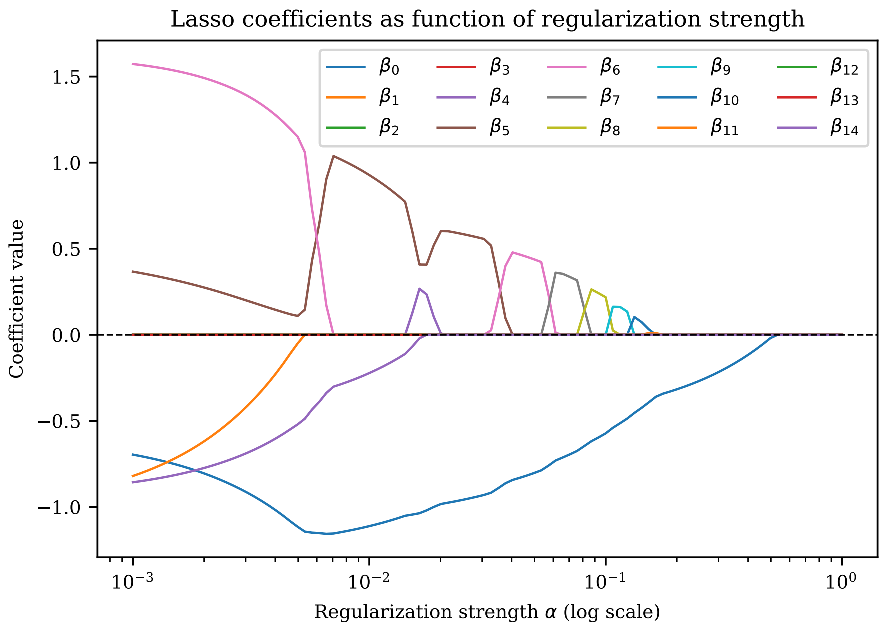
Clearly, in this case the model estimated by Lasso is substantially less complex than the linear regression or even Ridge regression. For most values of $\alpha$, only 2–3 features out of the original 15 are retained in the model!

Note that it is possible to compute the path of Lasso coefficients along the grid of $\alpha$ using the convenience function `lasso_path()`. This function does not support pipelines, so we have to apply any transformations and standardizations to `X` ourselves:

[28]:
```python
from sklearn.linear_model import lasso_path

# Compute Lasso path automatically
alphas, coefs, _ = lasso_path(
    X_trans, y,
    alphas=alphas,
    max_iter=100_000
)
```

[29]:
```python
plt.figure(figsize=(6, 4))
plt.plot(alphas, coefs.T, lw=1.0)
plt.xscale('log', base=10)
plt.axhline(0.0, ls='--', lw=0.75, c='black')
plt.xlabel(r'Regularization strength $\alpha$ (log scale)')
plt.ylabel('Coefficient value')
plt.title('Lasso coefficients as function of regularization strength')
plt.legend([rf'$\beta_{{{i}}}$' for i in range(degree)], ncols=5)
```

[29]:
```
<matplotlib.legend.Legend at 0x7f0f38086750>
```



As you can see, the estimated coefficients along the path are not exactly the same as when doing it manually, which might be because the two approaches call the underlying coordinate descent optimizer in different ways.

1.3.3 Tuning the regularization parameter via cross-validation

In the previous example, we picked an arbitrary regularization strength α when fitting the Lasso. In this section, we again find an optimal α using cross-validation. Just like in the case of Ridge regression, `scikit-learn` implements a cross-validation class specifically for Lasso called `LassoCV` which we use in the code below to find an optimal α.

`LassoCV` optionally accepts a grid of candidate α just like `RidgeCV`, but the default way to run cross-validation is to specify the fraction $\epsilon = \frac{\alpha_{min}}{\alpha_{max}}$ (default: 10^{-3}) and the grid size (default: 100). `LassoCV` then automatically determines an appropriate grid which is stored in the `alphas_` attribute once fitting is complete. Moreover, we use the `cv` argument to set the desired number of CV folds (default: 5).

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

# LassoCV does not support pipelines, so we need to transform x before
# cross-validation.
transform = make_pipeline(
    PolynomialFeatures(
        degree=degree,
        include_bias=False
    ),
    StandardScaler()
)

# Create standardized polynomial features
X_trans = transform.fit_transform(x[:, None])

# Create and run Lasso cross-validation, use defaults for eps and n_alphas
lcv = LassoCV(max_iter=100_000, cv=10).fit(X_trans, y)
```

After fitting, we can recover the best alpha from the `alpha_` attribute and the MSE for each α on the grid and each CV fold from the `mse_path_` attribute.

[31]:
```
# Recover best alpha that minimizes MSE
alpha_best = lcv.alpha_

# MSE for each alpha, averaged over folds
mse_mean = np.mean(lcv.mse_path_, axis=1)

# Index of min. MSE
imin = np.argmin(mse_mean)

mse_best = mse_mean[imin]

print(f'Best alpha: {alpha_best:.4g} (MSE: {mse_best:.3g})')
```

```
Best alpha: 0.0007855 (MSE: 0.251)
```

After fitting, The `LassoCV` instance stores the estimated parameters at the optimal $\alpha$ in the attribute `coef_`. We can use this vector to compute the number of features (explanatory variables) with non-zero coefficients:

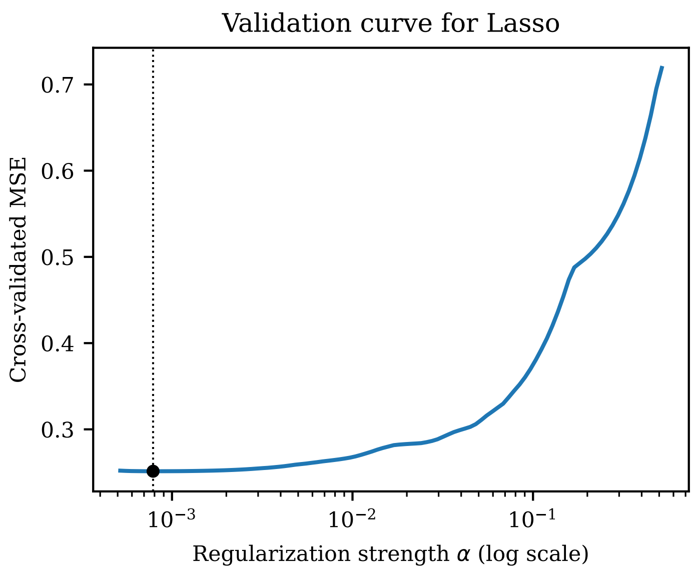
[32]:
```
n_features = np.sum(np.abs(lcv.coef_) > 0)
print(f'Number of non-zero coefficients: {n_features}')
```

```
Number of non-zero coefficients: 6
```

The next plot visualizes the average MSE over the entire range of candidate $\alpha$ values which are stored in the `alphas_` attribute of the `LassoCV` instance:

[33]:
```
import matplotlib.pyplot as plt

# Recover grid of alphas used for CV
alphas = lcv.alphas_

# Plot MSE against alphas, highlight minimum MSE
plt.plot(alphas, mse_mean)
plt.xlabel(r'Regularization strength $\alpha$ (log scale)')
plt.ylabel('Cross-validated MSE')
plt.scatter(alphas[imin], mse_mean[imin], s=15, c='black', zorder=100)
plt.axvline(alphas[imin], ls=':', lw=0.75, c='black')
plt.xscale('log')
plt.title('Validation curve for Lasso')
```

[33]:
```
Text(0.5, 1.0, 'Validation curve for Lasso')
```



Now that we have identified the optimal $\alpha$, we can re-fit the Lasso and plot the prediction from this model.

```
[34]: # Create pipeline with Lasso using optimal alpha
      pipe_lasso = make_pipeline(
          PolynomialFeatures(
              degree=degree,
              include_bias=False
          ),
          StandardScaler(),
          Lasso(alpha=alpha_best, fit_intercept=True, max_iter=100000)
      )

      pipe_lasso.fit(x[:, None], y)
```

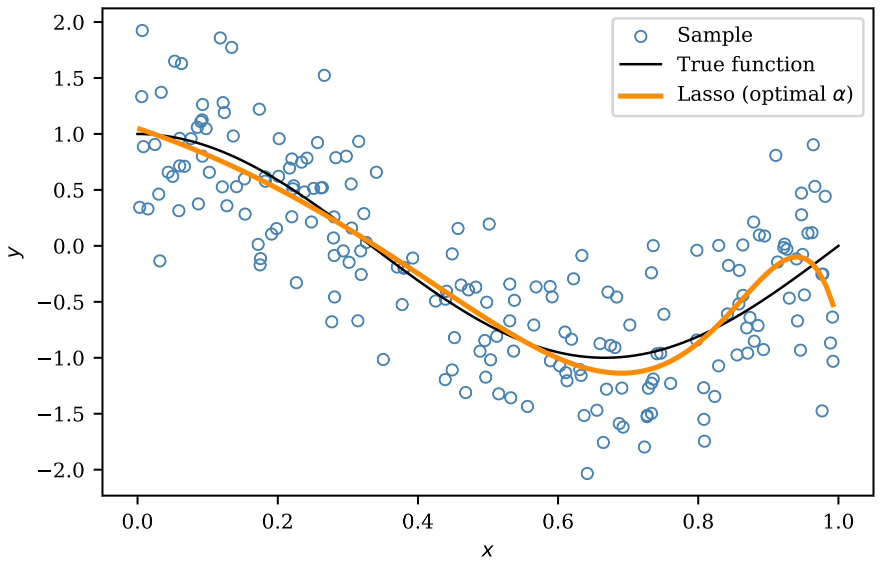
```
[34]: Pipeline(steps=[('polynomialfeatures',
                       PolynomialFeatures(degree=15, include_bias=False)),
                      ('standardscaler', StandardScaler()),
                      ('lasso',
                       Lasso(alpha=np.float64(0.0007855307824807166),
                             max_iter=100000))])
```

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

```
[35]: # Grid on which to evaluate true model and predictions
      xvalues = np.linspace(np.amin(x), np.amax(x), 100)

      # Predicted values from Lasso regression
      y_pred = pipe_lasso.predict(xvalues[:, None])
```

```
[36]: # Plot sample and true relationship
      ax = plot_trig_sample(x, y)

      # Plot prediction from optimal Lasso model
      plt.plot(xvalues, y_pred, c='darkorange', lw=2.0, label=r'Lasso (optimal $\alpha$)')

      plt.legend()
```

```
[36]: <matplotlib.legend.Legend at 0x7f0f18750860>
```



2 Models for classification

2.1 Logistic regression

In the previous sections, we studied how to predict continuous variables using linear models with shrinkage. We now turn to one of the most classical machine learning tasks, classification. To this end, we study the simplified setting where we have only two classes (labelled 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 features $\mathbf{x}_i$.

The classical approach to perform this task is to assume that the probability of the outcome 1 is given by

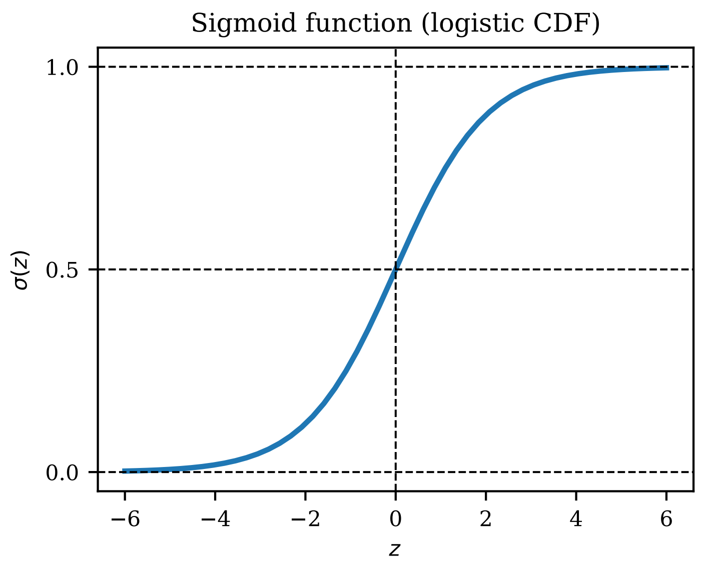
$$p(\mathbf{x}_i) \equiv \text{Prob}\left(y_i = 1 \mid \mathbf{x}_i\right) = \frac{1}{1 + \exp\left(\mu + \mathbf{x}_i'\boldsymbol{\beta}\right)}$$

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 density function (CDF) in statistics and econometrics. Note that this function maps any (positive or negative) real value $z$ into a value between $(0, 1)$, and hence is well-suited to model the probability that $y_i = 1$.

We can easily verify this by plotting the logistic CDF from `scipy.stats`:

[37]:
```
import numpy as np
from scipy.stats import logistic

zvalues = np.linspace(-6, 6, 50)

plt.plot(zvalues, logistic.cdf(zvalues), lw=2.0)
# Add horizontal and vertical lines
for y in (0.0, 0.5, 1.0):
    plt.axhline(y, ls='--', lw=0.75, c='black')
plt.axvline(0.0, ls='--', lw=0.75, c='black')
plt.xlabel('$z$')
plt.ylabel(r'$\sigma(z)$')
plt.yticks([0.0, 0.5, 1.0])
plt.title('Sigmoid function (logistic CDF)')
```

[37]:
```
Text(0.5, 1.0, 'Sigmoid function (logistic CDF)')
```



As you can see, the value of $\sigma(z)$ tends to 0 for small (negative) $z$, and it approaches 1 for large values of $z$.

In machine learning, the 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 features,

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

Consequently, one can show that the log odds-ratio is 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

$$\mathcal{L}(\mu,\boldsymbol{\beta}) = \sum_{i=1}^{N} y_i \log\bigl(p(\mathbf{x}_i)\bigr) + (1-y_i)\log\bigl(1-p(\mathbf{x}_i)\bigr)$$

The details underlying the derivation of this 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 α 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*.

To derive the loss function L for the logistic regression, we simply take the *negative* likelihood function and find the parameters $(\alpha, \boldsymbol{\beta})$ which *minimize* the loss L:

$$L(\mu,\boldsymbol{\beta}) = -\underbrace{\frac{1}{N}\mathcal{L}(\mu,\boldsymbol{\beta})}_{\text{scaled likelihood}} + \underbrace{\frac{r(\boldsymbol{\beta})}{C}}_{\text{regularization}}$$

Moreover, we have already incorporated the scaled regularization term $\frac{r(\boldsymbol{\beta})}{C}$ which is used analogously to Ridge regression and Lasso to shrink the parameters $\boldsymbol{\beta}$ and prevent overfitting. The fitted logistic regression parameters are thus

$$\left(\widehat{\mu}, \widehat{\boldsymbol{\beta}}\right) = \operatorname*{arg\,min}_{\mu,\boldsymbol{\beta}} L(\mu,\boldsymbol{\beta})$$

2.1.1 Example: Predicting binary class membership

To illustrate how to work with logistic regression, 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.

Creating a demo data set

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 again assume that $f(\bullet)$ is a combination of trigonometric functions which we'll attempt to approximate using polynomials. Specifically, the model studied in this section is

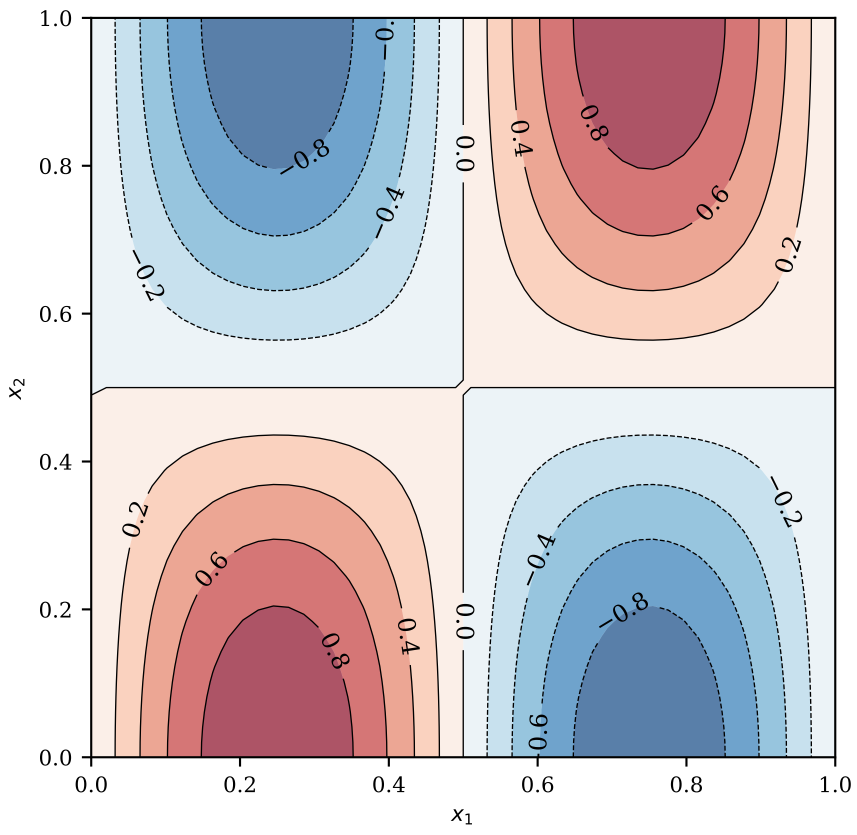
$$f(x_{1i}, x_{2i}) = \sin(2\pi x_{1i})\cos(\pi x_{2i})$$
$$\epsilon_i \overset{\text{iid}}{\sim} \mathcal{N}\left(0, \sigma_\epsilon^2\right)$$

The code below defines $f(x_{1i}, x_{2i})$ and plots the function on rectangular grid which we create with `np.meshgrid()`.

[38]:
```
import numpy as np

def f(x1, x2):
    """
    True function for classification example
    """
    return np.sin(2*np.pi*x1) * np.cos(np.pi*x2)
```

[39]:
```
import matplotlib.pyplot as plt

# x-values for grid
x = np.linspace(0, 1, 50)

# Create coordinates for x1 and x2
x1_grid, x2_grid = np.meshgrid(x, x)

# Deterministic function f(x1, x2)
z = f(x1_grid, x2_grid)

# Contour plot of deterministic function
plt.figure(figsize=(5, 5))
levels = np.linspace(-1, 1, 11)
cntrf = plt.contourf(x1_grid, x2_grid, z, levels=levels, cmap="RdBu_r", alpha=0.7)
CS = plt.contour(x1_grid, x2_grid, z, levels=levels, linewidths=0.5, colors='k')
plt.clabel(CS, CS.levels, fontsize=9)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
```

[39]:
```
Text(0, 0.5, '$x_2$')
```



As you see, this function takes on positive values in two quadrants (where y_i is thus 1) and negative values in the other two (where y_i is therefore 0). Without some noise ϵ_i, class membership would be trivial to predict, so we next create a sample that incorporates noise.

[40]:
```
# Enable automatic reloading of external modules
%load_ext autoreload
%autoreload 2
```

[41]:
```
from lecture12_classifiers import create_class_data

# Standard deviation of noise
sigma_eps = 0.2

# Create demo data set for classification
X, y = create_class_data(N=100, sigma=sigma_eps)
```

The last line in the previous code segment converts the latent continuous variable z_i to $y_i \in \{0, 1\}$. We can verify that y_i take on only two values with class labels 0 and 1:

[42]:
```
classes = np.unique(y)
classes
```

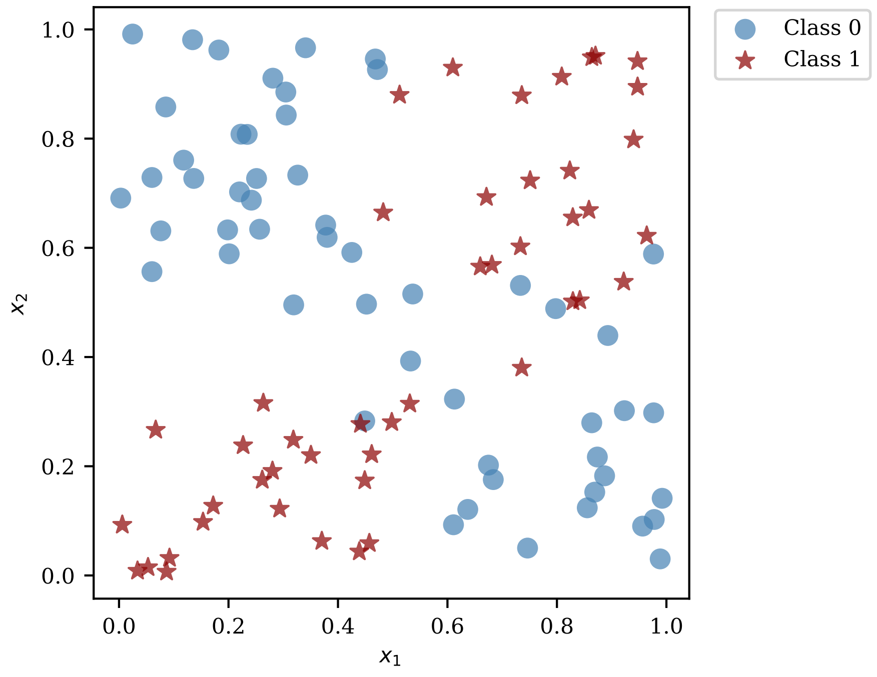
[42]:
```
array([0, 1])
```

Next, we import the function `plot_classes()` defined in `lecture12_classifiers.py` which creates a scatter plot from the sample and uses different marker symbols based on the class labels.

We will use this function repeatedly to plot the sample data and model predictions. The figure below visualizes the artificial 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.

[43]:
```
from lecture12_classifiers import plot_classes

plot_classes(X, y)
```

[43]:
```
<Axes: xlabel='$x_1$', ylabel='$x_2$'>
```



2.1.2 Fitting a simple model

As usual, before fitting the model, we first split the sample into a training and a test set. Unlike with linear regression models (Ridge, Lasso, OLS), it is advantageous to use *stratification* when splitting the sample so that both the training and test samples contain an approximately equal share of each class. We accomplish this by specifying the `stratify` keyword argument uses the values of the target variable y for stratification.

[44]:
```
from sklearn.model_selection import train_test_split

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, stratify=y, test_size=0.3, random_state=1234
)
```

Of course, given randomness and the small sample size used here, the share of classes will not be exactly identical across the training and test samples, as the following code shows:

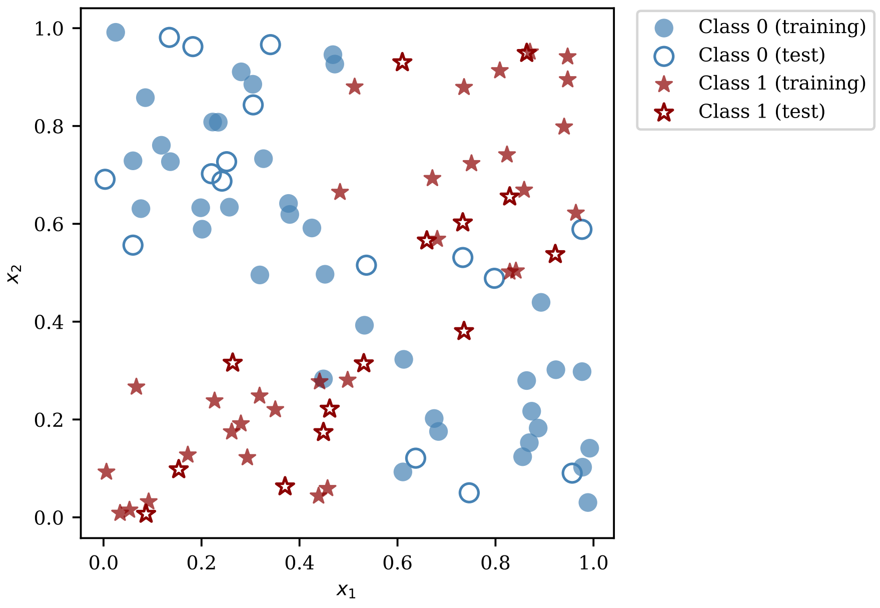
[45]:
```
import pandas as pd

print('Distribution of classes in training sample:')
print(pd.Series(y_train).value_counts() / len(y_train))

print('\nDistribution of classes in test sample:')
print(pd.Series(y_test).value_counts() / len(y_test))
```

```
Distribution of classes in training sample:
0    0.542857
1    0.457143
Name: count, dtype: float64

Distribution of classes in test sample:
0    0.533333
1    0.466667
Name: count, dtype: float64
```

We can now plot the training and test samples, using different market symbols for each:

[46]:
```
plot_classes(X_train, y_train, X_test, y_test)
```

[46]:
```
<Axes: xlabel='$x_1$', ylabel='$x_2$'>
```



We proceed to fit the model using the `LogisticRegression` estimator from scikit-learn. `LogisticRegression` implements four different types of regularization:

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 their corresponding `penalty` arguments that can be used when creating an instance of `LogisticRegression`:

Penalty	$r(\boldsymbol{\beta})$	`penalty` argument
L1	$r(\boldsymbol{\beta}) = \|\boldsymbol{\beta}\|_1 = \sum_{k=1}^K \|\beta_k\|$	`'l1'`
L2	$r(\boldsymbol{\beta}) = \frac{1}{2}\|\boldsymbol{\beta}\|_2^2 = \frac{1}{2}\sum_{k=1}^K \beta_k^2$	`'l2'`
L1 and L2	$r(\boldsymbol{\beta}) = \rho\|\boldsymbol{\beta}\|_1 + \frac{1-\rho}{2}\|\boldsymbol{\beta}\|_2^2$	`'ElasticNet'`
None		`None`

We first experiment with the simplest possible liner model with

$$z_i = \mu + \beta_1 x_{1i} + \beta_2 x_{2i}$$

so that the probability of observing class 1 is

$$p(\mathbf{x}_i) \equiv \text{Prob}\big(y_i = 1 \mid \mathbf{x}_i\big) = \frac{1}{1 + \exp\big(\,\mu + \beta_1 x_{1i} + \beta_2 x_{2i}\big)}$$

Moreover, for now we skip including a regularization term (`penalty=None`).

[47]:
```python
from sklearn.linear_model import LogisticRegression

# Create logistic regression model without regularization penalty
lr = LogisticRegression(
    penalty=None,
    fit_intercept=True,
    max_iter=1000
)
```

```
[48]: lr.fit(X_train, y_train)
```

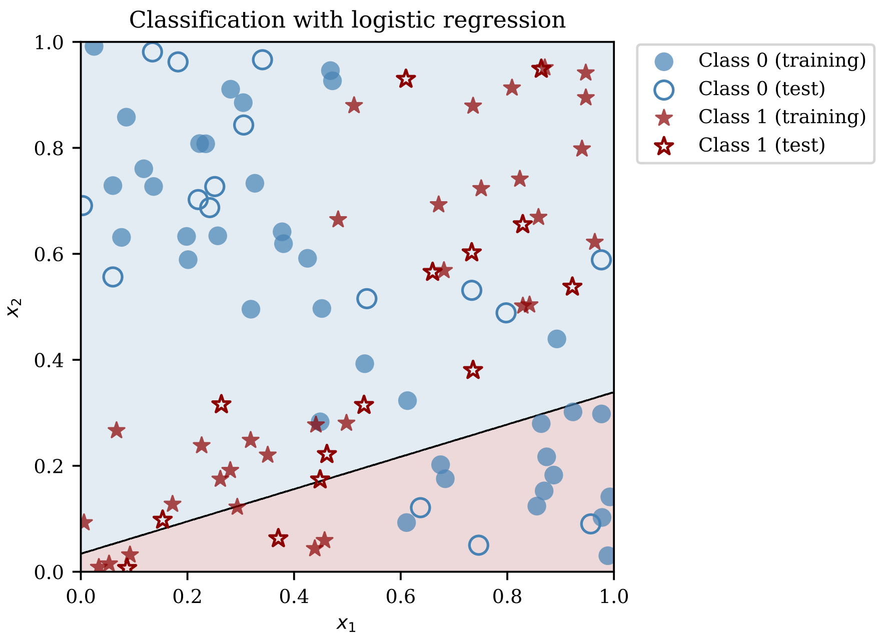
```
[48]: LogisticRegression(max_iter=1000, penalty=None)
```

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 `lecture12_classifiers.py`. The function takes as argument an existing Matplotlib `Axes` object and adds the decision boundaries to an existing figure.

```
[49]: from lecture12_classifiers import plot_decision_boundary

      # Create x-values used to evaluate decisions
      xvalues = np.linspace(0, 1, 1000)

      ax = plot_classes(X_train, y_train, X_test, y_test)
      plot_decision_boundary(ax, xvalues, lr)
      ax.set_title('Classification with logistic regression')
```

```
[49]: Text(0.5, 1.0, 'Classification with logistic regression')
```



As you see, the simple linear model does a rather poor job predicting class membership and misclassifies many observations. The graph shows that for any combination (x_{1i}, x_{2i}) falling into the blue area, the fitted model predicts $y_i = 0$, whereas in the red area it predicts $y_i = 1$.

We can explicitly compute the accuracy, i.e., the share of correctly predicted class labels, using the `score()` method of the estimator, or the `accuracy_score()` function. These return the same results.

```
[50]: y_train_pred = lr.predict(X_train)
      y_test_pred = lr.predict(X_test)
```

```
[51]: from sklearn.metrics import accuracy_score

      # Compute accuracy on training and test sets
      acc_train = accuracy_score(y_train, y_train_pred)
      acc_test = accuracy_score(y_test, y_test_pred)

      print(f'Accuracy on training sample: {acc_train:.3f}')
print(f'Accuracy on test sample: {acc_test:.3f}')
```

```
Accuracy on training sample: 0.443
Accuracy on test sample: 0.500
```

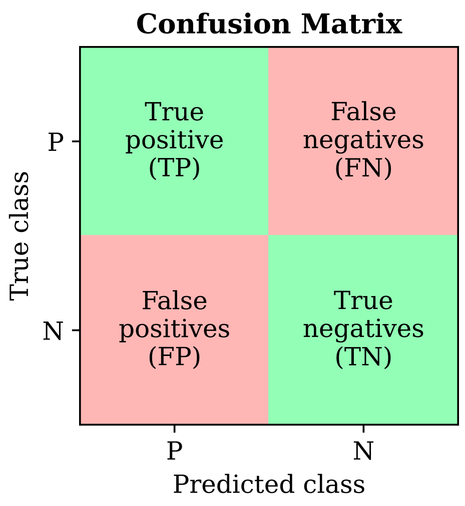
Alternatively, we can use the `score()` method directly to compute the same statistic, for example on the test sample:

[52]:
```
acc_test = lr.score(X_test, y_test)
print(f'Accuracy on test sample: {acc_test:.3f}')
```

```
Accuracy on test sample: 0.500
```

There are other metrics to assess how well the model predicts class labels. These usually combine the elements of the confusion matrix in some way. First, consider the confusion matrix for a generic classification problem (with labels `P` and `N` for `Positive` and `Negative`), plotted below:

[53]:
```
from lecture12_classifiers import plot_generic_confusion_matrix
plot_generic_confusion_matrix()
```



As you see, a classifier would ideally make predictions on some sample of known labels which are located in the green quadrants. Using the labels of this matrix, the accuracy score we have computed above is given by

$$ACC = \frac{TP + TN}{FP + FN + TP + TN}$$

However, we might be interest in different measures, for example *precision*, defined as

$$PRE = \frac{TP}{TP + FP}$$

or *recall*, defined as

$$REC = \frac{TP}{FN + TP}$$

Another commonly used scoring metric is the co-called *F1 score*, which is computed as

$$F1 = 2\frac{PRE \cdot REC}{PRE + REC}$$

Which of these metrics to use depends on the scenario. For example, we could have a sample where there are only very few positive observations, so a model that always predicts the negative label will have a high accuracy. On the other hand, it will do poorly measured by *recall*.

scikit-learn implements many commonly used scoring metrics in the module sklearn.metrics. Using functions from that module, we can compute the metrics defined above:

[54]:
```python
from sklearn.metrics import recall_score, precision_score, f1_score

# Precision
pre = precision_score(y_test, y_test_pred)
# Recall
rec = recall_score(y_test, y_test_pred)
# F1 score
f1 = f1_score(y_test, y_test_pred)

print(f'Precision: {pre:.3f}')
print(f'Recall: {rec:.3f}')
print(f'F1 score: {f1:.3f}')
```

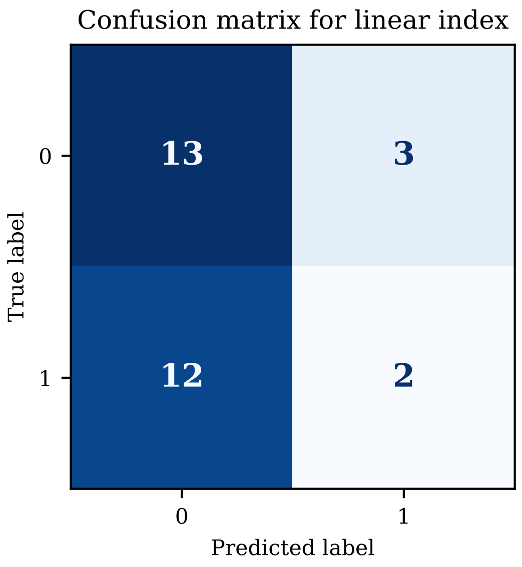
```
Precision: 0.400
Recall: 0.143
F1 score: 0.211
```

As you can see, the logistic regression with a linear index performs even poorer on some of these metrics than suggested by the accuracy score.

Finally, we can plot the confusion matrix using the class ConfusionMatrixDisplay. This plot can be created from the predicted values or directly from the estimator. We demonstrate how the confusion can be plotted from the predicted values below:

[55]:
```python
from sklearn.metrics import ConfusionMatrixDisplay

# Plot confusion matrix from predicted values
ConfusionMatrixDisplay.from_predictions(
    y_test,
    y_test_pred,
    colorbar=False,
    cmap='Blues',
    text_kw={'fontsize': 12, 'fontweight': 'bold'},
).ax_.set_title('Confusion matrix for linear index')
```

[55]:
```
Text(0.5, 1.0, 'Confusion matrix for linear index')
```



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 observations. We can predict these underlying probabilities using the `predict_proba()` method, for example for the test sample:

```
[56]: # Predict probabilities for first five test samples
lr.predict_proba(X_test)[:5]
```

```
[56]: array([[0.51830556, 0.48169444],
             [0.55159038, 0.44840962],
             [0.59726736, 0.40273264],
             [0.46472375, 0.53527625],
             [0.57511958, 0.42488042]])
```

Based on these predicted probabilities, `LogisticRegression` assigns the predicted class as the most likely outcome. With only two categories, $y_i = 1$ is therefore predicted whenever the value in the second column is larger than 0.5. Plugging this into the sigmoid function, we have

$$\begin{aligned}
y_i = 1 &\iff \sigma(\mu\beta_1 x_{1i} + \beta_2 x_{2i}) \geq 0.5 \\
&\iff \frac{1}{1 + \exp\left(-\mu - \beta_1 x_{1i} - \beta_2 x_{2i}\right)} \geq 0.5 \\
&\iff \exp\left(-\mu - \beta_1 x_{1i} - \beta_2 x_{2i}\right) \leq 1 \\
&\iff \mu + \beta_1 x_{1i} + \beta_2 x_{2i} \geq 0 \\
&\iff x_{2i} \leq \frac{\mu}{\beta_2} + \frac{\beta_1}{\beta_2} x_{1i}
\end{aligned}$$

The last step above follows because $\beta_2 < 0$ in this example as you can verify by checking the attribute `lr.coef_`. The take-away from these derivations is that any combination of features x_{1i}, x_{2i} is mapped to $y_i = 1$ if x_{2i} lies underneath the line defined by $\frac{\mu}{\beta_2} + \frac{\beta_1}{\beta_2} x_{1i}$, and therefore the decision boundary is a *straight line*.

2.1.3 Fitting a model with polynomials

We now extend the simply model and approximate the the true trigonometric 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.

```
[57]: from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import make_pipeline

# Maximum polynomial degree
degree = 5

# Create pipeline with polynomial features and logistic regression
pipe_lr = make_pipeline(
    PolynomialFeatures(degree=degree, include_bias=False),
    StandardScaler(),
    LogisticRegression(
        fit_intercept=True, penalty=None, max_iter=1000, random_state=1234
    ),
)
```

At this point we strictly speaking don't 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:

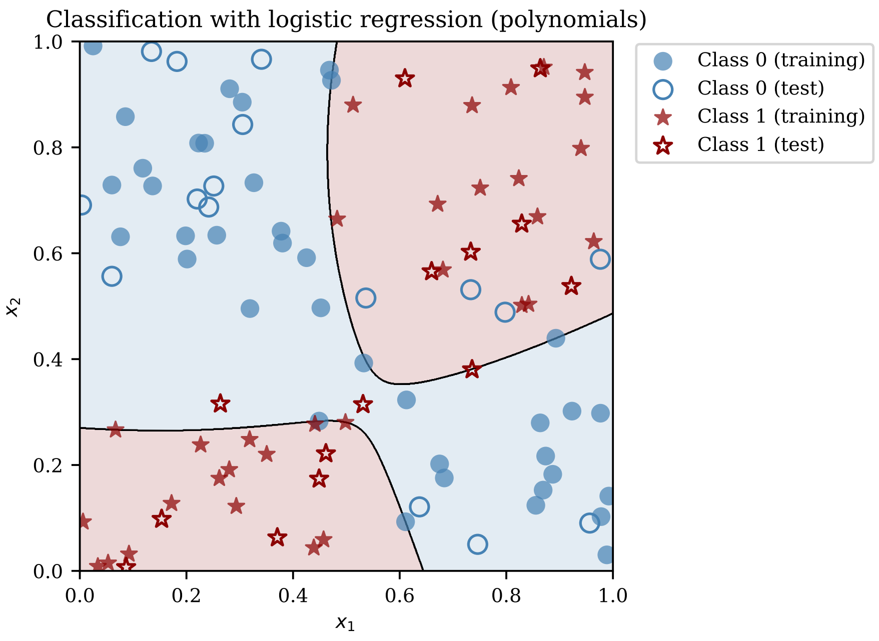
```
[58]: pipe_lr.fit(X_train, y_train)
```

[58]:
```
Pipeline(steps=[('polynomialfeatures',
                 PolynomialFeatures(degree=5, include_bias=False)),
                ('standardscaler', StandardScaler()),
                ('logisticregression',
                 LogisticRegression(max_iter=1000, penalty=None,
                                    random_state=1234))])
```

Using the functions defined above, we can plot the decision boundaries of this more complex model:

[59]:
```
# Create x-values used to evaluate decisions
xvalues = np.linspace(0, 1, 1000)
ax = plot_classes(X_train, y_train, X_test, y_test)
plot_decision_boundary(ax, xvalues, pipe_lr)
ax.set_title('Classification with logistic regression (polynomials)')
```

[59]:
```
Text(0.5, 1.0, 'Classification with logistic regression (polynomials)')
```



The substantial increase in model fit is immediately apparent. Using the accuracy score, we can confirm that the model is indeed much better predicting the class labels on the test sample:

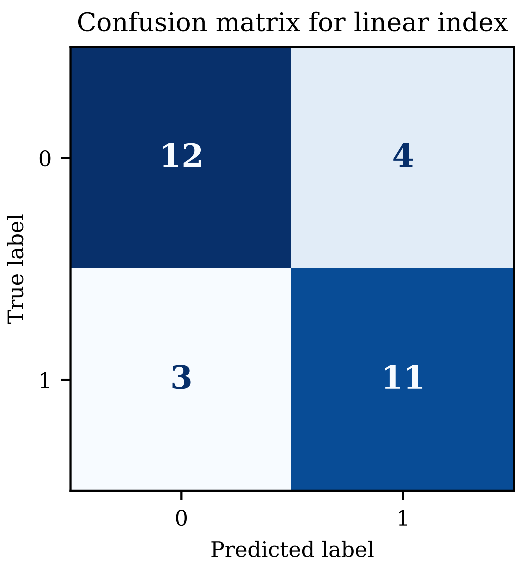
[60]:
```
y_test_pred = pipe_lr.predict(X_test)

acc_test = accuracy_score(y_test, y_test_pred)
pre_test = precision_score(y_test, y_test_pred)
rec_test = recall_score(y_test, y_test_pred)

print(f'Accuracy on test sample: {acc_test:.3f}')
print(f'Precision on test sample: {pre_test:.3f}')
print(f'Recall on test sample: {rec_test:.3f}')
```

```
Accuracy on test sample: 0.767
Precision on test sample: 0.733
Recall on test sample: 0.786
```

Using polynomials, the model predicts around 77% of cases in the test sample correctly. Plotting the confusion matrix, we also see that more of the true-predicted combinations are along the diagonals.

[61]:
```
# Plot confusion matrix from predicted values
ConfusionMatrixDisplay.from_predictions(
    y_test,
    y_test_pred,
    colorbar=False,
    cmap='Blues',
    text_kw={'fontsize': 12, 'fontweight': 'bold'},
).ax_.set_title('Confusion matrix for linear index')
```

[61]:
```
Text(0.5, 1.0, 'Confusion matrix for linear index')
```



2.1.4 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 be used to prevent overfitting.

Since we don't know the "best" regularization parameter C, we use LogisticRegressionCV to perform cross-validation.

As is the case with most specific cross-validation classes in scitkit-learn, `LogisticRegressionCV` does not support pipelines, so we need to manually create the polynomials and apply standardization on the training set before we run cross-validation:

[62]:
```
from sklearn.linear_model import LogisticRegressionCV

# Pipeline to create polynomial features and standardize them
transform = make_pipeline(
    PolynomialFeatures(degree=degree, include_bias=False), StandardScaler()
)

# Fit and transform training data
X_train_poly = transform.fit_transform(X_train)
```

We can now run the cross-validation. For this we choose the L2 penalty (the same as for Ridge regression), we use 10 folds, and tell `LogisticRegressionCV` to choose 50 values of C which are uniformly spaced in logs between the values 1e-4 and 1e4.

[63]:
```
# Create and run Logistic regression cross-validation
lrcv = LogisticRegressionCV(
    Cs=50,                      # Number of C to try, spaced uniformly in logs
    cv=10,                      # Number of folds
    fit_intercept=True,         # Fit intercept
    penalty='l2',               # Use L2 penalty
    max_iter=1000,              # Maximum iterations (increase from default)
    n_jobs=4,                   # Number of parallel jobs
    random_state=1234,          # Random seed
)

# Run cross-validation
lrcv.fit(X_train_poly, y_train)
```

```
[63]: LogisticRegressionCV(Cs=50, cv=10, max_iter=1000, n_jobs=4, random_state=1234)
```

Once the cross-validation is done, we can extract the best penalty parameter C from the array attribute `C_`.

In this case, the optimal C is given by

```
[64]: print(f'Best C: {lrcv.C_[0]:.4f}')
```

```
Best C: 35.5648
```

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.

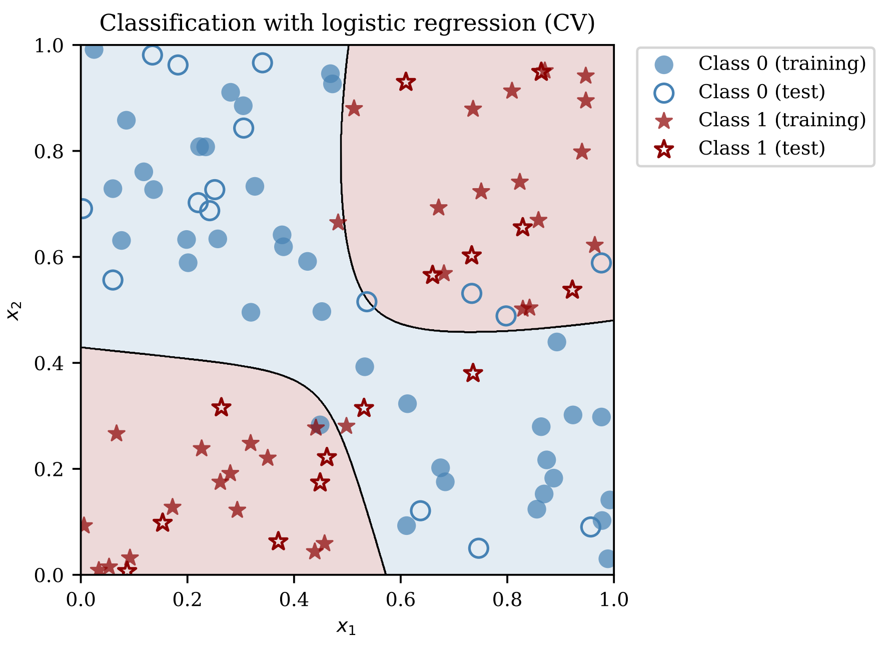
Finally, we recreate the whole pipeline we had earlier but use the optimal penalty parameter. This is not necessary if we just want to compute predicted values or accuracy metrics, but the plotting functions we have created expected the estimator pipeline to also create the polynomial features and perform standardization.

```
[65]: # Create pipeline using cross-validated C
lr_opt = make_pipeline(
    PolynomialFeatures(degree=degree, include_bias=False),
    StandardScaler(),
    LogisticRegression(
        C=lrcv.C_[0],
        fit_intercept=True,
        penalty='l2',
        max_iter=1000,
        random_state=1234
    ),
).fit(X_train, y_train)
```

We reuse the plotting code from above to plot the decision boundaries for the optimal model:

```
[66]: # Create x-values used to evaluate decisions
xvalues = np.linspace(0, 1, 1000)
ax = plot_classes(X_train, y_train, X_test, y_test)
plot_decision_boundary(ax, xvalues, lr_opt)
ax.set_title('Classification with logistic regression (CV)')
```

```
[66]: Text(0.5, 1.0, 'Classification with logistic regression (CV)')
```



There are slight improvements compared to the model without the penalty term. We can also confirm that the accuracy score for the test data increased from around 77% to 83%:

[67]:
```
# Compute accuracy of cross-validated model on test data
y_test_pred = lr_opt.predict(X_test)

acc_test = accuracy_score(y_test, y_test_pred)
pre_test = precision_score(y_test, y_test_pred)
rec_test = recall_score(y_test, y_test_pred)

print(f'Accuracy on test sample: {acc_test:.3f}')
print(f'Precision on test sample: {pre_test:.3f}')
print(f'Recall on test sample: {rec_test:.3f}')
```

```
Accuracy on test sample: 0.833
Precision on test sample: 0.800
Recall on test sample: 0.857
```

Note that the model above has a second hyperparameter, the maximum number of polynomial degrees, which we fixed at 5 for the entire cross-validation. In theory, we could perform the cross-validation *jointly* on the polynomial degree and C at the same time for even better results.

2.1.5 Validation curves

We conclude this section with plotting the validation curve for the above example as we vary the penalty parameter C. Recall that the validation curve plots some metric of model performance (in this case the accuracy score) computed from the validation sample against a range of values for a hyperparameter, in this case C (see the scikit-learn documentation for details).

We can leverage scikit-learn's `validation_curve()` function to perform this task for us. Since `validation_curve()` does work with pipelines, we first create the usual pipeline to create the polynomial features, perform standardization, and add a logistic regression estimator where we specify penalty to be of L2 type.

[68]:
```
pipe_lr = make_pipeline(
    PolynomialFeatures(degree=degree, include_bias=False),
    StandardScaler(),
    LogisticRegression(
        fit_intercept=True,
        penalty='l2',
        max_iter=1000,
        random_state=1234
    ),
)
```

`validation_curve()` uses two arguments to specify which parameter should be varied:

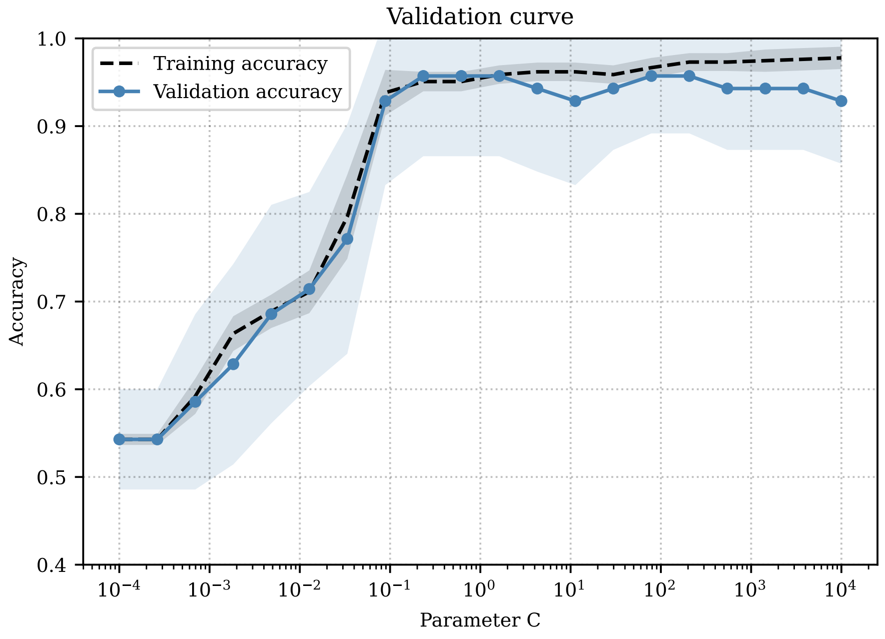
- `param_name` specifies the attribute which stores the parameter to be varied. In case of pipelines, this name takes the form `STEPNAME_ATTRIBUTE` where `STEPNAME` is the name of the step in the pipeline, and `ATTRIBUTE` is the name of the attribute of that particular step to be varied.
- `param_range` specifies the range of parameter values to be varied along the validation curve.

For our specific logistic regression pipeline, the call to `validation_curve()` looks as follows:

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

# Define range of values for C, spaced uniformly in logs
param_range = np.logspace(-4, 4, 20)

# Compute validation curve
train_scores, test_scores = validation_curve(
    estimator=pipe_lr,
    X=X_train,
    y=y_train,
    param_name='logisticregression__C',
    param_range=param_range,
    cv=10,
    n_jobs=4,
)
```

The module `lecture12_classifiers` implements the function `plot_accuracy_validation_curve()` which we can use to plot the validation curve for a given range of parameter values:

[70]:
```
from lecture12_classifiers import plot_accuracy_validation_curve

plot_accuracy_validation_curve(param_range, train_scores, test_scores)
```



As the plot below suggests, for small C (i.e., for very large penalty strengths), the model underfits the data both in the training and test samples, leading to low accuracy. For very high values of C, on the other hand, the model overfits, leading to lower accuracy in the test sample (but increasing accuracy in the training sample).

2.2 Other commonly used classifiers

In this section, we study three other widely used classifiers implemented in scikit-learn:

1. Support vector machines (SVM);
2. Decision trees; and
3. Random forests

These classifiers should be viewed as alternatives to logistic regression studied earlier. In fact, because of the composable API of scikit-learn, it is usually trivial to switch out the `LogisticRegression` class in a processing and estimation pipeline, so one can (and should) easily experiment with several classifiers to determine which yields the best accuracy for a given task.

When running any of these classifiers, we need to be aware that they differ in terms of data processing, estimation and cross-validation:

1. Some classifiers like `LogisticRegression` have a fixed functional form (given by the sigmoid function), so we might need to expand the feature space by creating polynomial interactions or similar, as we have done previously. This step is not needed for decision trees or random forests, and it is most likely not needed for SVM because we can select a different kernel to create a nonlinear model instead of resorting to a polynomial basis.
2. Some classifiers *require* feature standardization (in particular, this is true for logistic regression with regularization and SVM), whereas this step is not necessary for decision trees or random forests.
3. Classifiers differ in their hyperparameters, and different types of cross-validation might need to be performed for different classifiers.

2.2.1 Support vector machines (SVM)

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. You can think of a kernel as a function that operates on the features and projects them onto a different 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 transformed using this kernel, the individual classes become much easier to separate.

Linear kernel

For illustration purposes, we first create the a new artificial data set for classification with more observations and more noise to magnify the difference between SVM kernels and hyperparameters (with the original demo sample, all but the simplest linear SVM yield basically the same result).

[71]:
```
# Create demo data for binary classification
sigma_eps = 0.25

# Create data set
```

```
X, y = create_class_data(N=200, sigma=sigma_eps)

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, stratify=y, test_size=0.3, random_state=1234
)
```

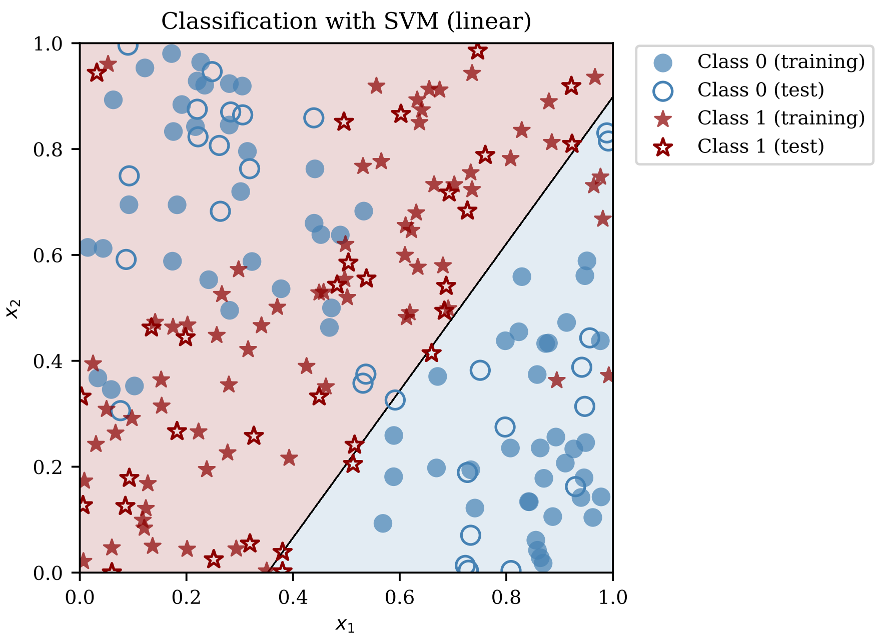
We start with the simplest linear kernel which is selected using the `kernel='linear'` argument.

[72]:
```
from sklearn.svm import SVC
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import make_pipeline

# Create pipeline with with linear SVM
pipe_svm = make_pipeline(
    StandardScaler(),
    SVC(kernel='linear', C=1.0, random_state=1234)
).fit(X_train, y_train)
```

Plotting the decision boundaries, we see that the linear kernel leads to a single linear decision boundary:

[73]:
```
xvalues = np.linspace(0, 1, 1000)
ax = plot_classes(X_train, y_train, X_test, y_test)
plot_decision_boundary(ax, xvalues, pipe_svm)
ax.set_title('Classification with SVM (linear)')
```

[73]:
```
Text(0.5, 1.0, 'Classification with SVM (linear)')
```



This model is not doing overly great in terms of accuracy, classifying only about 71% of test samples correctly:

[74]:
```
y_train_pred_svm = pipe_svm.predict(X_train)
y_test_pred_svm = pipe_svm.predict(X_test)

acc_train = accuracy_score(y_train, y_train_pred_svm)
acc_test = accuracy_score(y_test, y_test_pred_svm)

print(f'Accuracy on training sample: {acc_train:.3f}')
print(f'Accuracy on test sample: {acc_test:.3f}')
```

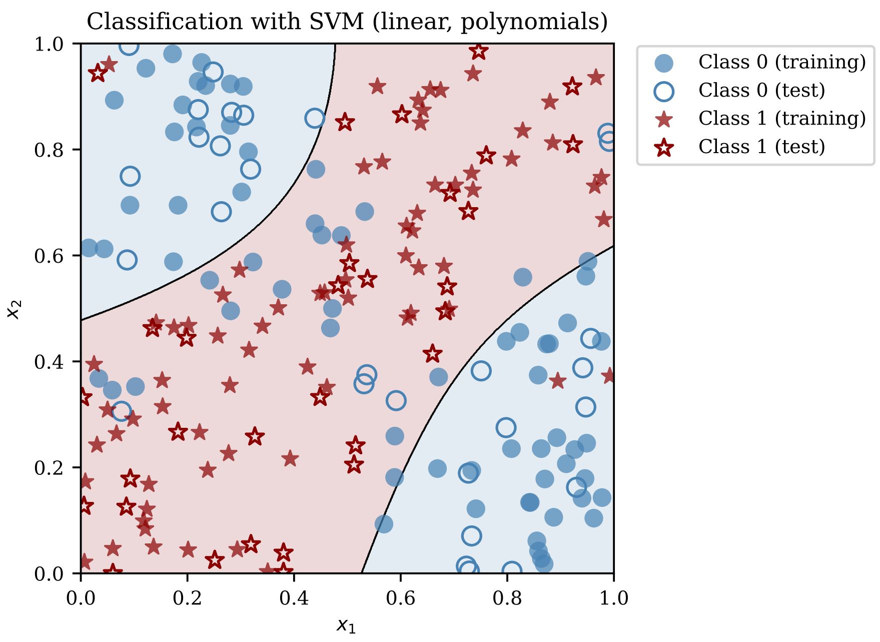
```
Accuracy on training sample: 0.729
Accuracy on test sample: 0.683
```

As with logistic regression, we can easily augment this model by adding polynomial interactions of features:

[75]:
```
# Fit linear SVM with polynomial features

# Maximum polynomial degree
degree = 5

# Create pipeline with polynomial features and SVM
pipe_svm_poly = make_pipeline(
    PolynomialFeatures(degree=degree, include_bias=False),
    StandardScaler(),
    SVC(kernel='linear', C=1.0, random_state=1234)
).fit(X_train, y_train)
```

[76]:
```
xvalues = np.linspace(0, 1, 1000)
ax = plot_classes(X_train, y_train, X_test, y_test)
plot_decision_boundary(ax, xvalues, pipe_svm_poly)
ax.set_title('Classification with SVM (linear, polynomials)')
```

[76]:
```
Text(0.5, 1.0, 'Classification with SVM (linear, polynomials)')
```



This yields considerably higher accuracy on the test sample:

[77]:
```
y_train_pred_svm = pipe_svm_poly.predict(X_train)
y_test_pred_svm = pipe_svm_poly.predict(X_test)

acc_train = accuracy_score(y_train, y_train_pred_svm)
acc_test = accuracy_score(y_test, y_test_pred_svm)

print(f'Accuracy on training sample: {acc_train:.3f}')
print(f'Accuracy on test sample: {acc_test:.3f}')
```

```
Accuracy on training sample: 0.850
Accuracy on test sample: 0.883
```

Gaussian kernel

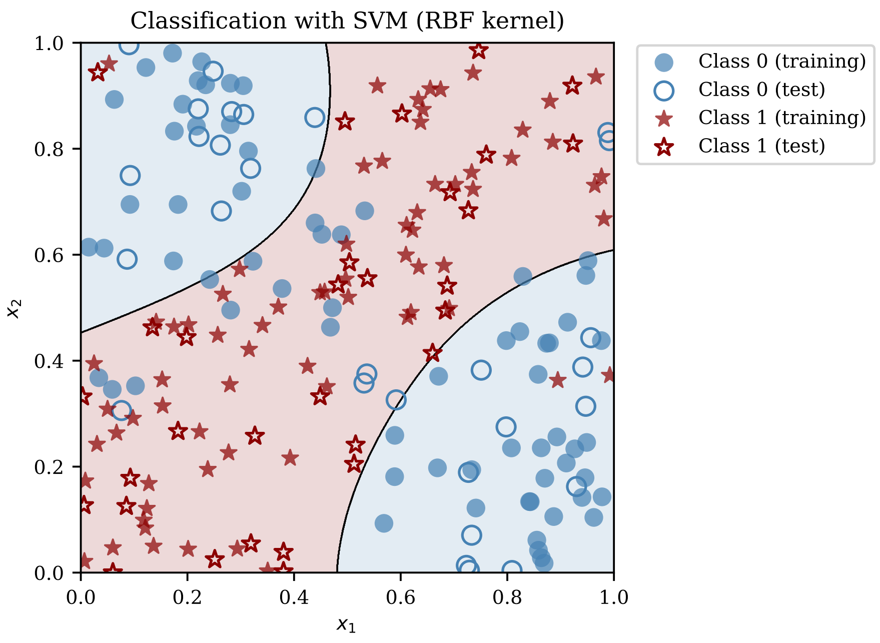
However, it is not necessary to introduce polynomial features with SVM. Alternatively, we can operate with just the two original features, but choose a different kernel. By default, the kernel used by `SVC` is `kernel='rbf'` which stands for the *radial basis function*. This kernel is also called the Gaussian kernel.

We rerun the above code, omitting the polynomial features but instead selecting the Gaussian kernel:

[78]:
```
# Create SVM with RBF kernel
pipe_svm_rbf = make_pipeline(
    StandardScaler(),
    SVC(kernel='rbf', C=1.0, gamma='scale', random_state=1234)
).fit(X_train, y_train)
```

[79]:
```
ax = plot_classes(X_train, y_train, X_test, y_test)
plot_decision_boundary(ax, xvalues, pipe_svm_rbf)
ax.set_title('Classification with SVM (RBF kernel)')
```

[79]:
```
Text(0.5, 1.0, 'Classification with SVM (RBF kernel)')
```



The decision boundaries are quite similar to the linear kernel with polynomials, as is the accuracy of the fitted model:

[80]:
```
y_train_pred_svm = pipe_svm_rbf.predict(X_train)
y_test_pred_svm = pipe_svm_rbf.predict(X_test)

acc_train = accuracy_score(y_train, y_train_pred_svm)
acc_test = accuracy_score(y_test, y_test_pred_svm)

print(f'Accuracy on training sample: {acc_train:.3f}')
print(f'Accuracy on test sample: {acc_test:.3f}')
```

```
Accuracy on training sample: 0.879
Accuracy on test sample: 0.883
```

So far, we haven't done any hyperparameter tuning. The RBF kernel has *two* hyperparameter arguments:

1. `C`, the regularization penalty strength (which we encountered before); and

2. `gamma`, which scales the Euclidean distance between any two feature vectors.

With two hyperparameters, we cannot resort to functions such as the `validation_curve()` we used earlier to plot how accuracy changes with a hyperparameter. Instead, we use the `GridSearchCV` class which performs parameter tuning over a grid of *several* parameters (in this case, over `C` and `gamma`). The parameter grid is specified using the `param_grid` argument which accepts a dictionary with a list of candidate values for *each* parameter. The grid search is then performed over the Cartesian product of these candidate values, so in the code below, 100×100 different combinations of parameters are evaluated.

[81]:
```
from sklearn.model_selection import GridSearchCV

# Define range of values for C and gamma, spaced uniformly in logs
param_range = np.logspace(-1, 2, 100)

# 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
gs_rbf = GridSearchCV(
    estimator=pipe_svm_rbf,     # Model to tune
    param_grid=param_grid,      # Dictionary of hyperparameter grids
    cv=10,                      # Number of folds
    refit=True,                 # Refit best model
    n_jobs=8                    # 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_`:

[82]:
```
print(f'Best accuracy: {gs_rbf.best_score_:.3f}')
print(f'Best parameters: {gs_rbf.best_params_}')
```

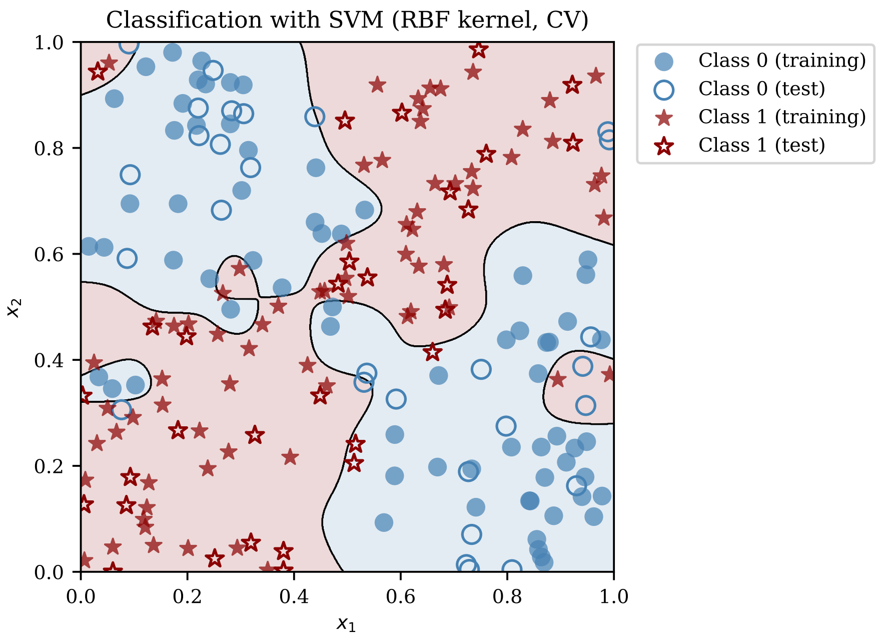
```
Best accuracy: 0.907
Best parameters: {'svc__C': np.float64(53.366992312063125), 'svc__gamma':
np.float64(10.722672220103231)}
```

Note that we don't need to re-estimate the model with these optimal parameters since we specified `refit=True` when creating an instance of `GridSearchCV`, so the resulting object already reflects the optimal model.

Finally, we can plot the decision boundaries for the cross-validated SVM model:

[83]:
```
ax = plot_classes(X_train, y_train, X_test, y_test)
plot_decision_boundary(ax, xvalues, gs_rbf)
ax.set_title('Classification with SVM (RBF kernel, CV)')
```

[83]:
```
Text(0.5, 1.0, 'Classification with SVM (RBF kernel, CV)')
```



As you can suspect from the figure, the cross-validated model is in fact overfitting more than the one with default hyperparameters. This is confirmed by the accuracy score, which is *lower* on the test sample than it was before.

[84]:
```
y_train_pred_svm = gs_rbf.predict(X_train)
y_test_pred_svm = gs_rbf.predict(X_test)

acc_train = accuracy_score(y_train, y_train_pred_svm)
acc_test = accuracy_score(y_test, y_test_pred_svm)

print(f'Accuracy on training sample: {acc_train:.3f}')
print(f'Accuracy on test sample: {acc_test:.3f}')
```

```
Accuracy on training sample: 1.000
Accuracy on test sample: 0.817
```

2.2.2 Decision trees

Decision trees are a nonlinear classifier which 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 highest information gain.

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 prune the tree by specifying a maximum tree depth.

Decision trees are implemented in the `DecisionTreeClassifier` class in scikit-learn.

To demonstrate their use, we recreate the artificial classification sample we used for logistic regression earlier.

[85]:
```
# Create demo data for binary classification
sigma_eps = 0.2

# Create data set
X, y = create_class_data(N=100, sigma=sigma_eps)

# Split data into training and test sets
```

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

We now grow the tree by fitting it to the training data. As explained above, we limit the maximum tree depth to 3 layers by specifying the argument `max_depth=3`.

It should be stressed that decision trees are a fully nonlinear model, and there is no need 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.

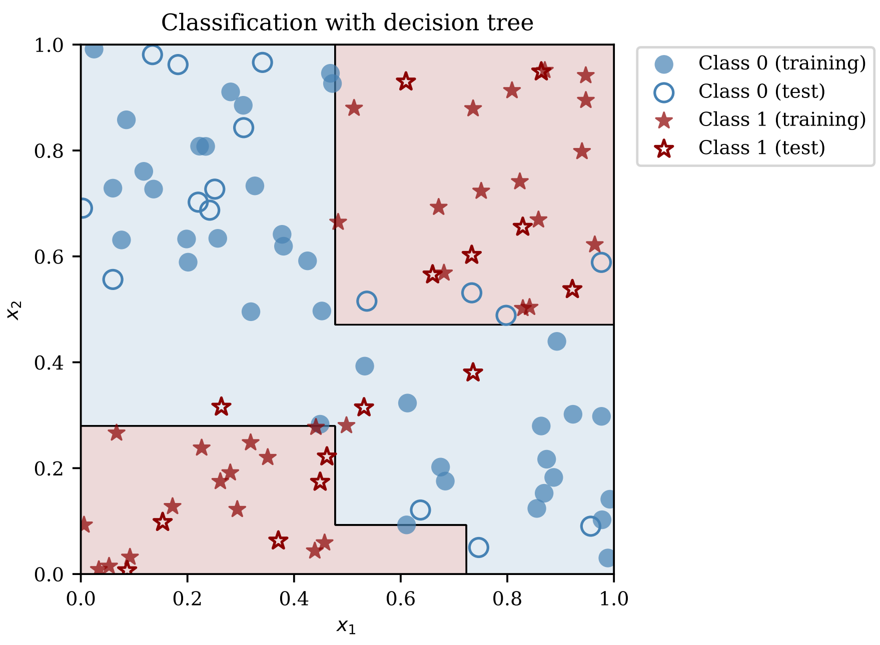
[86]:
```
from sklearn.tree import DecisionTreeClassifier

# Don't need feature scaling for trees
tree = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=1234)

# Grow tree on training data
tree.fit(X_train, y_train)
```

[86]:
```
DecisionTreeClassifier(max_depth=3, random_state=1234)
```

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.

[87]:
```
ax = plot_classes(X_train, y_train, X_test, y_test)
plot_decision_boundary(ax, xvalues, tree)
ax.set_title('Classification with decision tree')
```

[87]:
```
Text(0.5, 1.0, 'Classification with decision tree')
```



Computing the accuracy score, we see that the decision tree produces the a worse result than the cross-validated logistic regression we used earlier:

[88]:
```
y_train_pred_tree = tree.predict(X_train)
y_test_pred_tree = tree.predict(X_test)

acc_train = accuracy_score(y_train, y_train_pred_tree)
acc_test = accuracy_score(y_test, y_test_pred_tree)
```

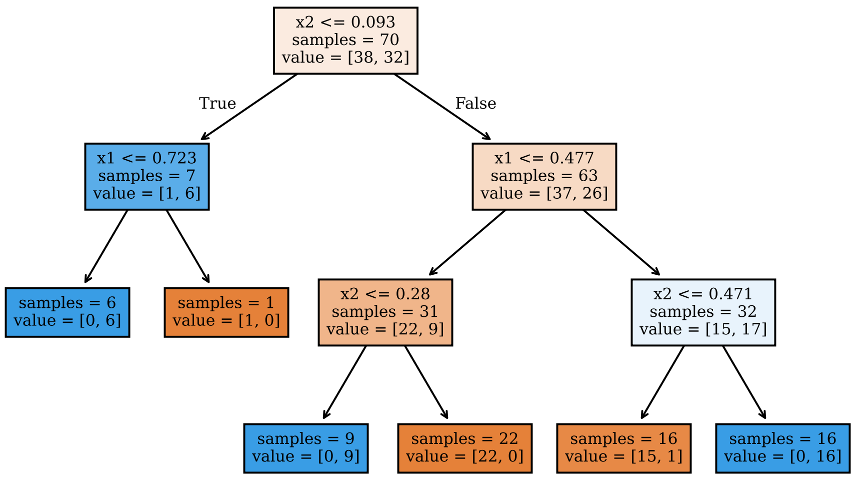
```
print(f'Accuracy on training sample: {acc_train:.3f}')
print(f'Accuracy on test sample: {acc_test:.3f}')
```

```
Accuracy on training sample: 0.986
Accuracy on test sample: 0.767
```

scikit-learn offers a convenience function which allows us to visualize the decision tree. At each node, the first line states the criterion used to split the sample, the second line reports the sample size at the node before the split has been applied, and the third line reports the number of observations for each class label.

For example, at the top node, the sample is split according to whether the feature x_{2i} is larger than 0.093 or not. As you can see, most of the bottom leafs are pure since they contain only observations of a single class.

[89]:
```
from sklearn.tree import plot_tree

fig, ax = plt.subplots(figsize=(8, 5))
_ = plot_tree(tree, filled=True, feature_names=['x1', 'x2'], fontsize=8, impurity=False,
  ↪ax=ax)
```



We can use the `GridSearchCV` cross-validator to find the optimal tree depth for this sample as follows:

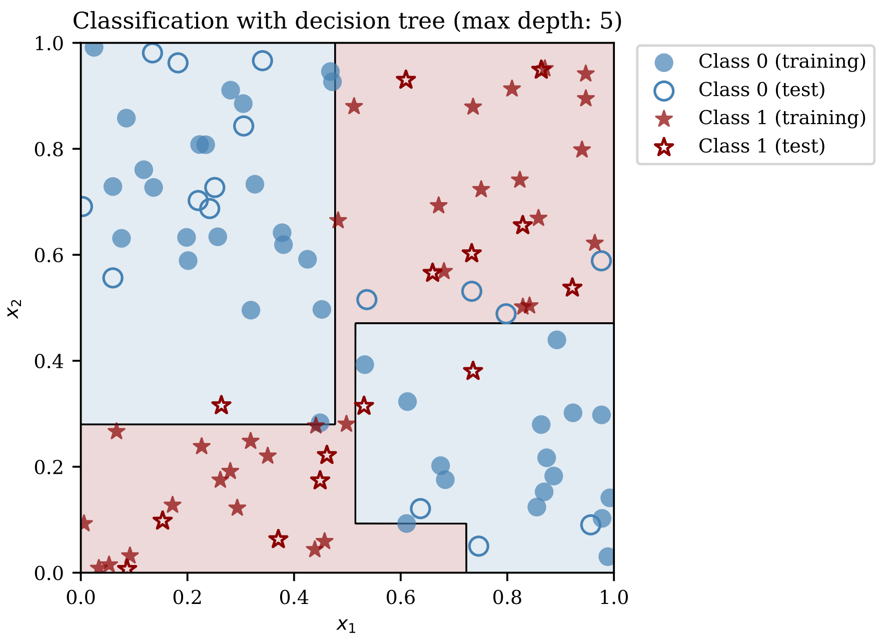
[90]:
```
from sklearn.model_selection import GridSearchCV

# Define range of values for max_depth
param_range = np.arange(1, 11)

treecv = GridSearchCV(estimator=tree,
    param_grid={'max_depth': param_range},
    cv=10,
    refit=True,
    n_jobs=4
).fit(X_train, y_train)

print(f'Best accuracy: {treecv.best_score_:.3f}')
print(f'Best parameters: {treecv.best_params_}')
```

```
Best accuracy: 0.886
Best parameters: {'max_depth': np.int64(5)}
```

As you can see, the optimal cross-validated maximum depth is computed to be 5.

```
[91]: ax = plot_classes(X_train, y_train, X_test, y_test)
      plot_decision_boundary(ax, xvalues, treecv)
      ax.set_title(
          f'Classification with decision tree (max depth: {treecv.best_params_["max_depth"]})'
      )
```

```
[91]: Text(0.5, 1.0, 'Classification with decision tree (max depth: 5)')
```



The cross-validated maximum depth leads to a small increase in classification accuracy on the test sample:

```
[92]: y_train_pred_tree = treecv.predict(X_train)
      y_test_pred_tree = treecv.predict(X_test)

      acc_train = accuracy_score(y_train, y_train_pred_tree)
      acc_test = accuracy_score(y_test, y_test_pred_tree)

      print(f'Accuracy on training sample: {acc_train:.3f}')
      print(f'Accuracy on test sample: {acc_test:.3f}')
```

```
Accuracy on training sample: 1.000
Accuracy on test sample: 0.767
```

2.2.3 Random forest

A random forest classifier combines multiple decision trees on various subsamples of the data and averages their result. 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. The most important arguments are the number of decision trees to grow (`n_estimators`) and the maximum tree depth (`max_depth`).

Just like decision trees, a random forest is not sensitive to scaling, so we can skip the usual preprocessing pipeline.

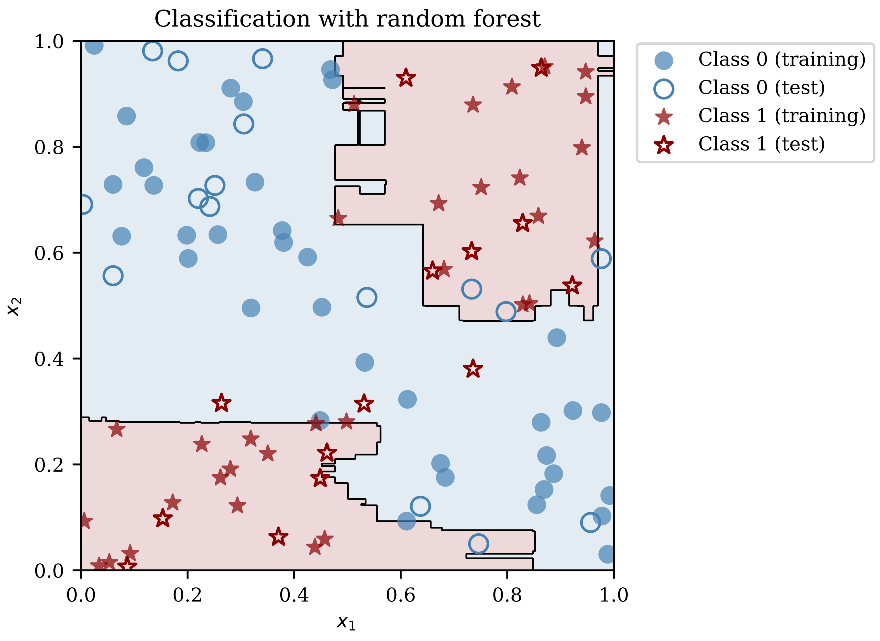
```
[93]: from sklearn.ensemble import RandomForestClassifier
```

```
# Create random forest classifier, fit on training data
forest = RandomForestClassifier(
    n_estimators=100, max_depth=3, random_state=1234, n_jobs=4
).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.

[94]:
```
ax = plot_classes(X_train, y_train, X_test, y_test)
plot_decision_boundary(ax, xvalues, forest)
ax.set_title('Classification with random forest')
```

[94]:
```
Text(0.5, 1.0, 'Classification with random forest')
```



As you can see, the random forest already has a higher prediction accuracy than single tree, even without cross-validation.

[95]:
```
y_train_pred_forest = forest.predict(X_train)
y_test_pred_forest = forest.predict(X_test)

acc_train = accuracy_score(y_train, y_train_pred_forest)
acc_test = accuracy_score(y_test, y_test_pred_forest)

print(f'Accuracy on training sample: {acc_train:.3f}')
print(f'Accuracy on test sample: {acc_test:.3f}')
```

```
Accuracy on training sample: 0.986
Accuracy on test sample: 0.800
```

We can of course use `GridSearchCV` to determine the maximum tree depth (`max_depth`) or the number of trees to grow (`n_estimators`) for the random forest. This works analogously to what we did for decision trees:

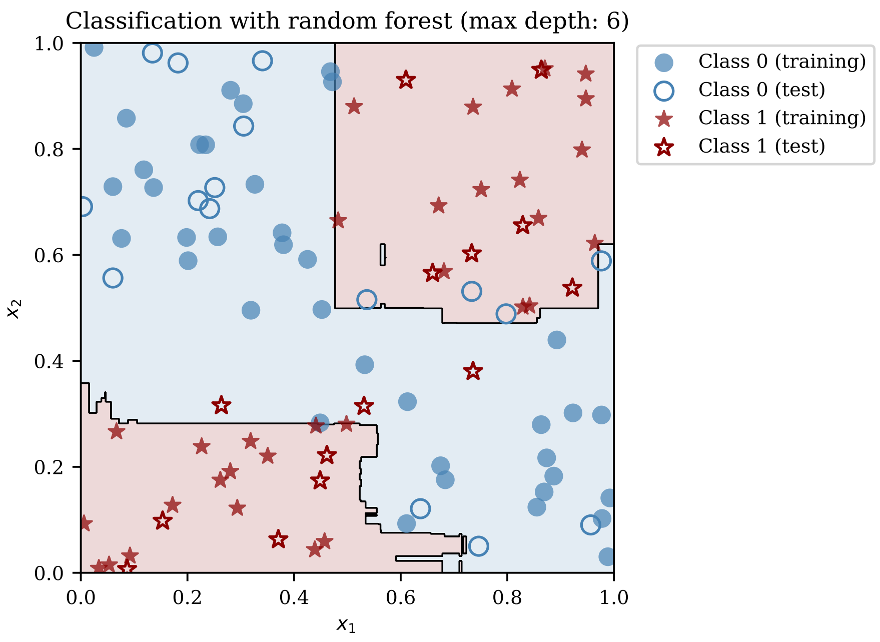
[96]:
```
from sklearn.model_selection import GridSearchCV

# Define range of values for max_depth
max_depth_grid = np.arange(1, 11)

# Define range of values for n_estimators
```

```
n_estimators_grid = np.arange(100, 200, 10)

# Run grid search over two hyperparameters
forestcv = GridSearchCV(
    estimator=forest,
    param_grid={'max_depth': max_depth_grid, 'n_estimators': n_estimators_grid},
    cv=10,
    refit=True,
    n_jobs=8,
).fit(X_train, y_train)

print(f'Best accuracy: {forestcv.best_score_:.3f}')
print(f'Best parameters: {forestcv.best_params_}')
```

```
Best accuracy: 0.929
Best parameters: {'max_depth': np.int64(6), 'n_estimators': np.int64(130)}
```

[97]:
```
ax = plot_classes(X_train, y_train, X_test, y_test)
plot_decision_boundary(ax, xvalues, forestcv)
ax.set_title(f'Classification with random forest (max depth: {forestcv.
↪best_params_["max_depth"]})')
```

[97]:
```
Text(0.5, 1.0, 'Classification with random forest (max depth: 6)')
```



As the output below shows, the cross-validated maximum tree depth of 6 does not lead to any more improvements in accuracy.

[98]:
```
y_train_pred_forest = forest.predict(X_train)
y_test_pred_forest = forest.predict(X_test)

acc_train = accuracy_score(y_train, y_train_pred_forest)
acc_test = accuracy_score(y_test, y_test_pred_forest)

print(f'Accuracy on training sample: {acc_train:.3f}')
print(f'Accuracy on test sample: {acc_test:.3f}')
```

```
Accuracy on training sample: 0.986
Accuracy on test sample: 0.800
```