Linear models

Linear models make a prediction using a linear function of the input features.
Can be very powerful for or datasets with many features.

If you have more features than training data points, any target y can be perfectly modeled (on the training set) as a linear function.

Linear models for regression

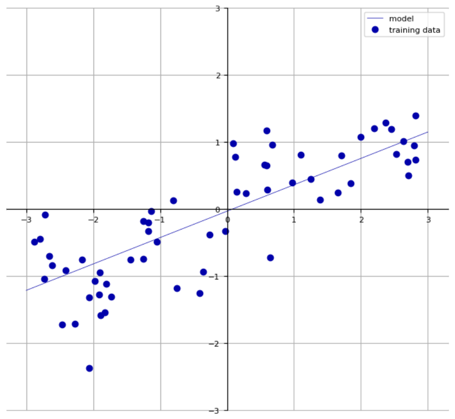
Prediction formula for input features x. w_i and b are the *model parameters* that need to be learned.

$$\hat{y} = w_0 * x_0 + w_1 * x_1 + ... + w_p * x_p + b$$

There are many different algorithms, differing in how w and b are learned from the training data.

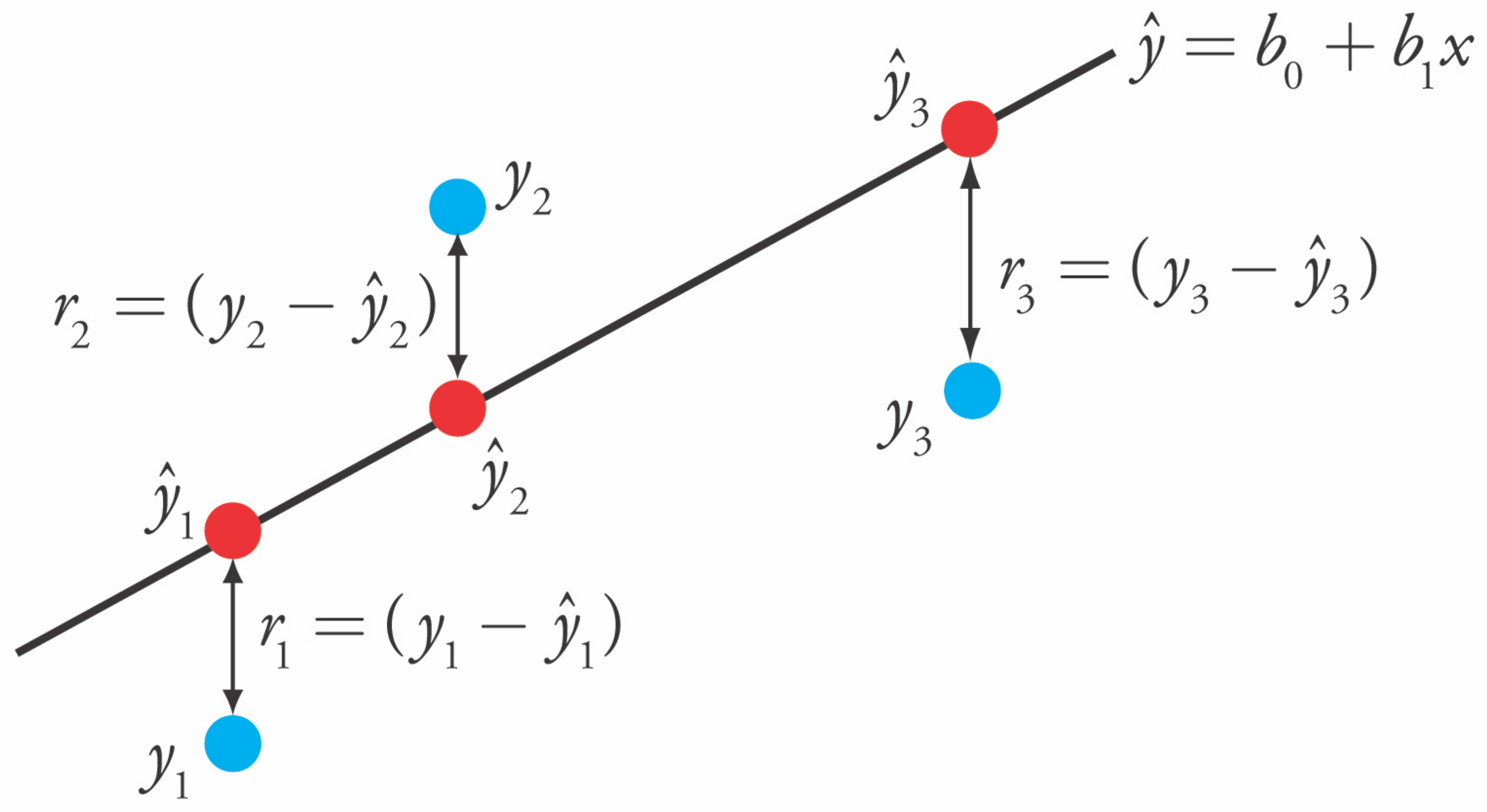
In [2]:
```
mglearn.plots.plot_linear_regression_
wave()
```

```
w[0]: 0.393906  b: -
0.031804
```



Linear Regression aka Ordinary Least Squares

- Finds the parameters w and b that minimize the *mean squared error* between predictions and the true regression targets, y, on the training set.
  - MSE: Sum of the squared differences between the predictions and the true values.
- Convex optimization problem with unique closed-form solution (if you have more data points than model parameters w)
- It has no hyperparameters, thus model complexity cannot be controlled.


$\hat{y} = b_0 + b_1 x$
$\hat{y}_3$
y_2
$r_2 = (y_2 - \hat{y}_2)$
$r_3 = (y_3 - \hat{y}_3)$
y_3
$\hat{y}_2$
$\hat{y}_1$
$r_1 = (y_1 - \hat{y}_1)$
y_1

Linear regression can be found in `sklearn.linear_model`. We'll evaluate it on the Boston Housing dataset.

In [3]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

In [4]:
```
print("Weights (coef
ficients): {}".forma
t(lr.coef_))
print("Bias (interce
pt): {}".format(lr.i
ntercept_))
```

```
Weights (coefficients): [ -402.752   -
50.071  -133.317   -12.002   -12.711
  28.305    54.492
   -51.734    25.26     36.499   -10.1
04   -19.629   -21.368    14.647
  2895.054  1510.269   117.995   -26.5
66    31.249   -31.446    45.254
  1283.496 -2246.003   222.199    -0.4
66    40.766   -13.436   -19.096
    -2.776   -80.971     9.731     5.1
33    -0.788    -7.603    33.672
   -11.505    66.267   -17.563    42.9
83     1.277     0.61     57.187
    14.082    55.34    -30.348    18.8
12   -13.777    60.979   -12.579
   -12.002   -17.698   -34.028     7.1
5     -8.41     16.986   -12.941
   -11.806    57.133   -17.581     1.6
96    27.218   -16.745    75.03
   -30.272    47.78    -40.541     5.5
04    21.531    25.366   -49.485
    28.109    10.469   -71.559   -23.7
4      9.574    -3.788     1.214
    -4.72     41.238   -37.702    -2.1
56   -26.296   -33.202    45.932
   -23.014   -17.515   -14.085   -20.4
9     36.525   -94.897   143.234
   -15.674   -14.973   -28.613   -31.2
52    24.565   -17.805     4.035
     1.711    34.474    11.219     1.1
43     3.737    31.385]
Bias (intercept): 31.645174100825688
```

Ridge regression

- Same formula as linear regression
- Adds a penalty term to the least squares sum : $\lambda \sum_i w_i^2$
- Requires that the coefficients (w) are close to zero.
    - Each feature should have as little effect on the outcome as possible
- Regularization: explicitly restrict a model to avoid overfitting.
- Type of L2 regularization: prefers many small weights
    - L1 regularization prefers sparsity: many weights to be 0, others large

`Ridge` can also be found in `sklearn.linear_model`.

```
In [6]: from sklearn.linear_model import Ridge

        ridge = Ridge().fit(X_train, y_train)
        print("Training set score: {:.2f}".format(ridge.
        score(X_train, y_train)))
        print("Test set score: {:.2f}".format(ridge.scor
        e(X_test, y_test)))
```

```
Training s
et score:
0.89
Test set s
core: 0.75
```

Test set score is higher and training set score lower: less overfitting!

The strength of the regularization can be controlled with the `alpha` parameter. Default is 1.0.

- Increasing alpha forces coefficients to move more toward zero (more regularization)
- Decreasing alpha allows the coefficients to be less restricted (less regularization)

In [7]:

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge1
0.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge10.sc
ore(X_test, y_test)))
```

```
Training
set scor
e: 0.79
Test set
score: 0.
64
```

In [8]:

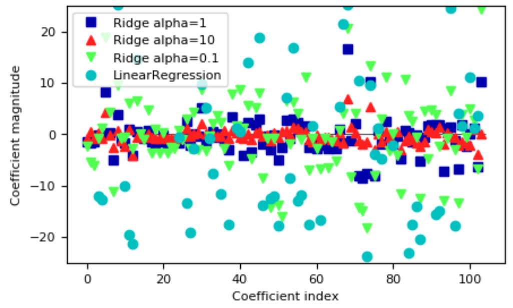
```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge0
1.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge01.sc
ore(X_test, y_test)))
```

```
Training
set scor
e: 0.93
Test set
score: 0.
77
```

We can plot the weight values for differents levels of regularization.

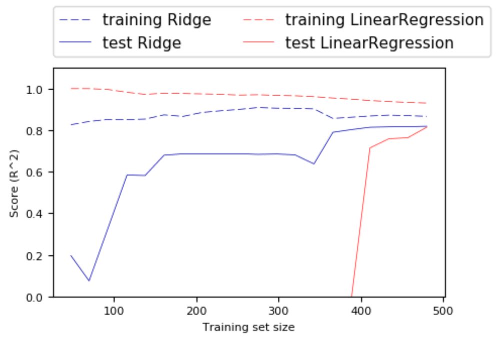
In [9]:
```
plt.plot(ridge.coef_, 's', label=
"Ridge alpha=1")
plt.plot(ridge10.coef_, '^', label
="Ridge alpha=10")
plt.plot(ridge01.coef_, 'v', label
="Ridge alpha=0.1")

plt.plot(lr.coef_, 'o', label="Lin
earRegression")
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude"
)
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```

Out[9]:
```
<matplotlib.legend.Lege
nd at 0x11cc1ed30>
```



Another way to understand the influence of regularization is to fix a value of alpha but vary the amount of training data available. With enough training data, regularization becomes less important: ridge and linear regression will have the same performance.

In [10]:
```
mglearn.plots.plot_ridge_n_samples
()
```



Lasso

- Another form of regularization
- Adds a penalty term to the least squares sum : $\lambda \sum_i |w_i|$
- Prefers coefficients to be exactly zero (L1 regularization).
- Some features are entirely ignored by the model: automatic feature selection.
- Same parameter `alpha` to control the strength of regularization.
- New parameter `max_iter`: the maximum number of iterations
  - Should be higher for small values of `alpha`

In [11]:
```
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso
.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso.sco
re(X_test, y_test)))
print("Number of features used: {}".format(np.s
um(lasso.coef_ != 0)))
```

```
Training se
t score: 0.
29
Test set sc
ore: 0.21
Number of f
eatures use
d: 4
```

In [12]:
```
# we increase the default setting of "max_ite
r",
# otherwise the model would warn us that we sho
uld increase max_iter.
lasso001 = Lasso(alpha=0.01, max_iter=100000).f
it(X_train, y_train)
print("Training set score: {:.2f}".format(lasso
001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso001.
score(X_test, y_test)))
print("Number of features used: {}".format(np.s
um(lasso001.coef_ != 0)))
```

```
Training se
t score: 0.
90
Test set sc
ore: 0.77
Number of f
eatures use
d: 33
```

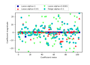
In [13]:

```
lasso00001 = Lasso(alpha=0.0001, max_iter=10000
0).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso
00001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso0000
1.score(X_test, y_test)))
print("Number of features used: {}".format(np.s
um(lasso00001.coef_ != 0)))
```

```
Training s
et score:
0.95
Test set s
core: 0.64
Number of
features u
sed: 94
```

We can again analyse what happens to the weigths:

In [14]:
```
plt.plot(lasso.coef_, 's', label="Lasso alpha=1")
plt.plot(lasso001.coef_, '^', label="Lasso alpha=0.01"
)
plt.plot(lasso00001.coef_, 'v', label="Lasso alpha=0.0
001")
plt.plot(ridge01.coef_, 'o', label="Ridge alpha=0.1")

plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude");
```



Interpreting L1 and L2 loss

- Red ellipses are the contours of the least squares error function
- In blue are the constraints imposed by the L1 (left) and L2 (right) loss functions
- For L1, the likelihood of hitting the objective with the corners is higher
  - Weights of other coefficients are 0, hence sparse representations
- For L2, it could intersect at any point, hence non-zero weights
- From *Elements of Statistical Learning*:

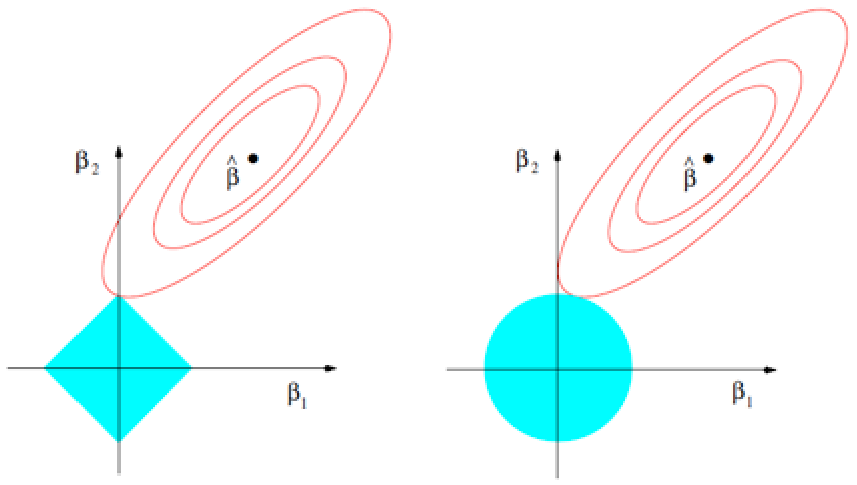


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

Linear models for Classification

Aims to find a (hyper)plane that separates the examples of each class.
For binary classification (2 classes), we aim to fit the following function:

$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p + b > 0$$

When $\hat{y} < 0$, predict class -1, otherwise predict class +1

There are many algorithms for learning linear classification models, differing in:

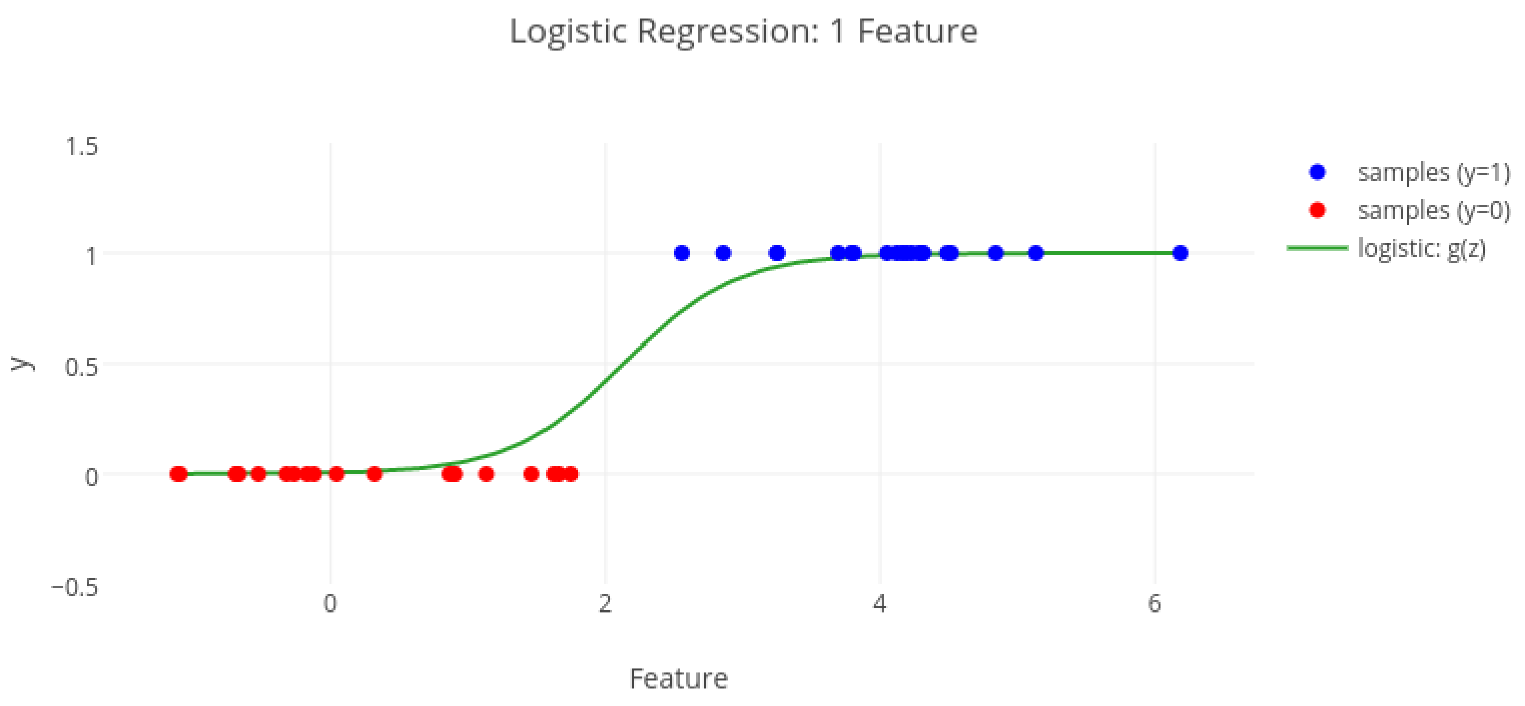
- Loss function: evaluate how well the linear model fits the training data
- Regularization techniques

Most common techniques:

- Logistic regression:
  - `sklearn.linear_model.LogisticRegression`
- Linear Support Vector Machine:
  - `sklearn.svm.LinearSVC`

Logistic regression

Fits a logistic regression curve/surface to the data



- Logistic regression predicts the target using the logarithm of the class probability:
$$Pr[1|x_1, \ldots, x_k] \rightarrow log(\frac{Pr[1|x_1, \ldots, x_k]}{1 - Pr[1|x_1, \ldots, x_k]})$$
- *Logit transformation* maps [0,1] to (-Inf,Inf)
- Resulting class probability (the green curve in the figure above):
$$Pr[1|x_1, \ldots, x_k] = \frac{1}{1 + exp(-(w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p))})$$
- Parameters w are found from training data using *maximum likelihood*

Maximum likelihood

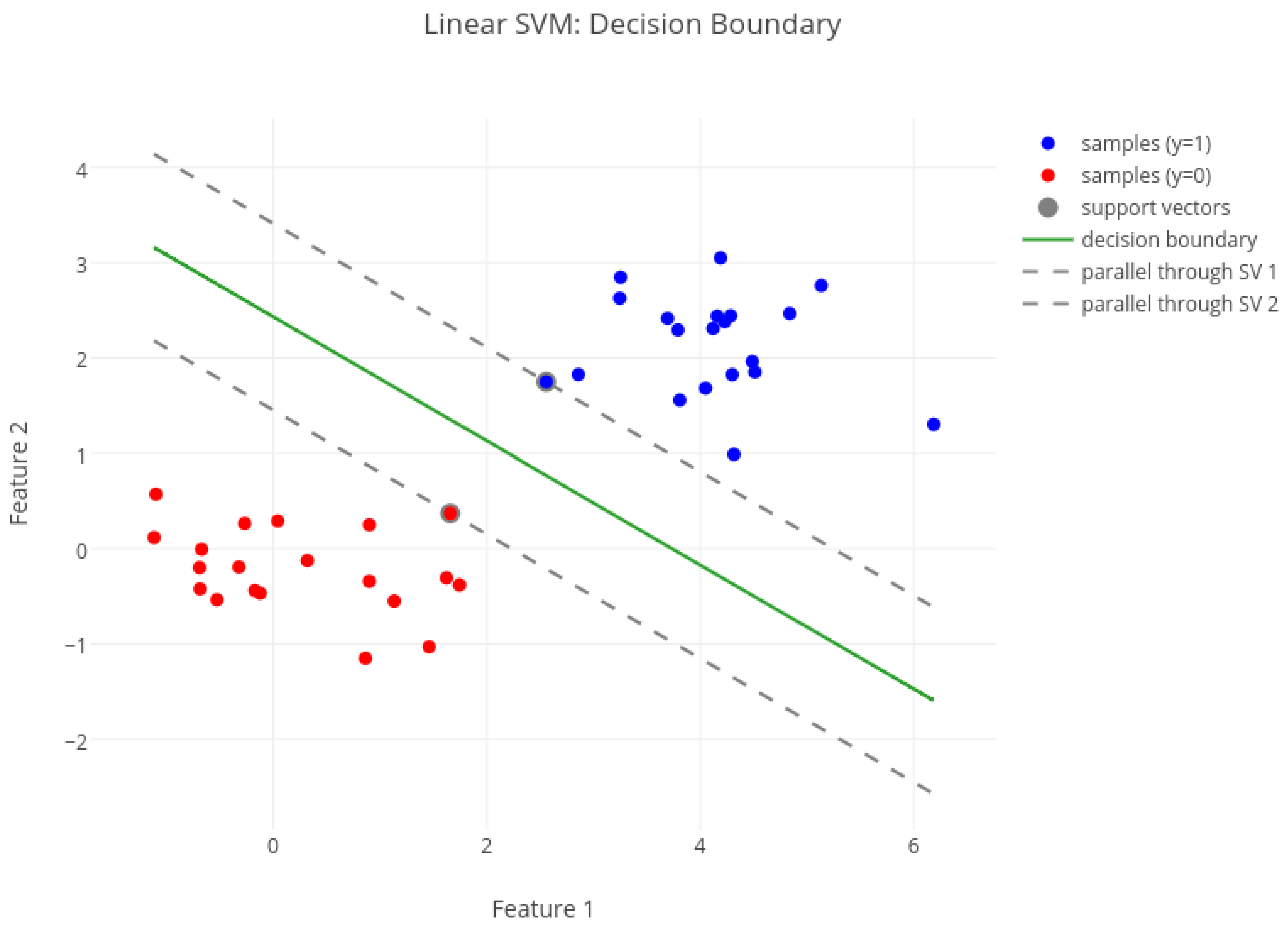
- Aim: maximize the probability of the observed training data with respect to the final model parameters
- We can use logaritms of probabilities and maximize conditional *log-likelihood* instead of the product of probabilities

$$\sum_{i=1}^{n}(1-y^{(i)})log(1-Pr[1|x_1^{(i)},\ldots,x_k^{(i)}])+y^{(i)}Pr[1|x_1^{(i)},\ldots,x_k^{(i)}]$$

  - Class values $y^{(i)}$ are either 0 or 1
- Weights w_i need to be chosen to maximize log-likelihood
  - This can be done using *iterative re-weighted least squares*
  - Other optimization methods can be used as well

Linear Support Vector Machine

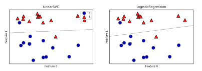
Find hyperplane maximizing the *margin* between the classes



Prediction is identical to weighted kNN: find the support vector that is nearest, according to a distance measure (kernel) and a weight for each support vector.

Comparison

In [15]:
```
from sklearn.linear_model import LogisticRegressi
on
from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for model, ax in zip([LinearSVC(), LogisticRegres
sion()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=
False, eps=0.5,
                                    ax=ax, alpha
=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y,
 ax=ax)
    ax.set_title("{}".format(clf.__class__.__name
__))
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
axes[0].legend();
```

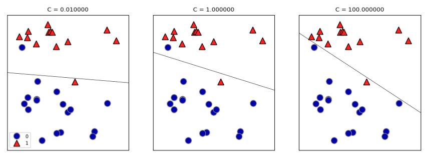


Both methods can be regularized:

- L2 regularization by default, L1 also possible
- C parameter: inverse of strength of regularization
    - higher C: less regularization
    - penalty for misclassifying points while keeping w_i close to 0

High *C* values (less regularization): fewer misclassifications but smaller margins.

```
In [16]: mglearn.plots.plot_linear_svc_regularization()
```



Model selection: Logistic regression

In [17]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_s
plit(
    cancer.data, cancer.target, stratify=cancer
.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_tr
ain)
print("Training set score: {:.3f}".format(logre
g.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg.sc
ore(X_test, y_test)))
```

```
Training s
et score:
0.953
Test set s
core: 0.95
8
```

In [18]:

```
logreg100 = LogisticRegression(C=100).fit(X_trai
n, y_train)
print("Training set score: {:.3f}".format(logreg
100.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg100.
score(X_test, y_test)))
```

```
Training s
et score:
0.979
Test set s
core: 0.96
5
```

In [19]:

```
logreg001 = LogisticRegression(C=0.01).fit(X_tra
in, y_train)
print("Training set score: {:.3f}".format(logreg
001.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg001.
score(X_test, y_test)))
```

```
Training s
et score:
0.934
Test set s
core: 0.93
0
```

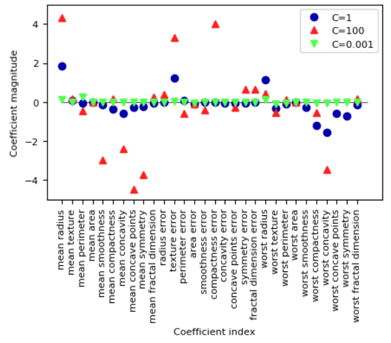
Effect of *C* on model parameters:

In [20]:

```
plt.plot(logreg.coef_.T, 'o', label="C=
1")
plt.plot(logreg100.coef_.T, '^', label=
"C=100")
plt.plot(logreg001.coef_.T, 'v', label=
"C=0.001")
plt.xticks(range(cancer.data.shape[1]),
cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.legend()
```

Out[20]:

```
<matplotlib.legen
d.Legend at 0x11e
301eb8>
```



Idem with L1 regularization (`penalty='l1'`):

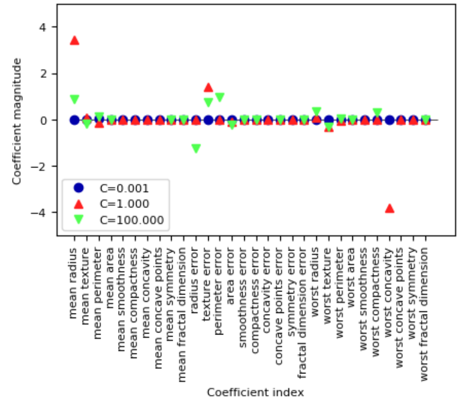
In [21]:

```
for C, marker in zip([0.001, 1, 100],
 ['o', '^', 'v']):
    lr_l1 = LogisticRegression(C=C, p
enalty="l1").fit(X_train, y_train)
    print("Training accuracy of l1 lo
greg with C={:.3f}: {:.2f}".format(
          C, lr_l1.score(X_train, y_t
rain)))
    print("Test accuracy of l1 logreg
 with C={:.3f}: {:.2f}".format(
          C, lr_l1.score(X_test, y_te
st)))
    plt.plot(lr_l1.coef_.T, marker, l
abel="C={:.3f}".format(C))

plt.xticks(range(cancer.data.shape[1
]), cancer.feature_names, rotation=90
)
plt.hlines(0, 0, cancer.data.shape[1
])
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")

plt.ylim(-5, 5)
plt.legend(loc=3)
```

```
Training accuracy of
l1 logreg with C=0.00
1: 0.91
Test accuracy of l1 l
ogreg with C=0.001:
0.92
Training accuracy of
l1 logreg with C=1.00
0: 0.96
Test accuracy of l1 l
ogreg with C=1.000:
0.96
Training accuracy of
l1 logreg with C=100.
000: 0.99
Test accuracy of l1 l
ogreg with C=100.000:
0.98
```

Out[21]:

```
<matplotlib.legend.Le
gend at 0x107f5f400>
```



Linear Models for multiclass classification

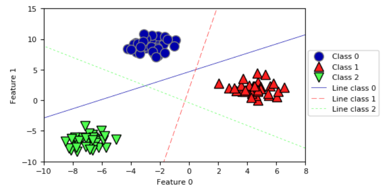
Common technique: one-vs.-rest approach:

- A binary model is learned for each class vs. all other classes
- Creates as many binary models as there are classes
- Every binary classifiers makes a prediction, the one with the highest score (>0) wins

Build binary linear models:

```
In [22]:
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
linear_svm = LinearSVC().fit(X, y)

mglearn.discrete_scatter(X[:, 0], X[:, 1
], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_
svm.coef_, linear_svm.intercept_,
                                  mglearn
.cm3.colors):
    plt.plot(line, -(line * coef[0] + int
ercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(['Class 0', 'Class 1', 'Class
 2', 'Line class 0', 'Line class 1',
            'Line class 2'], loc=(1.01,
0.3))
```

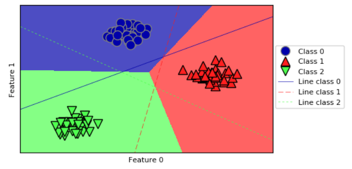
```
Out[22]:
<matplotlib.legen
d.Legend at 0x11c
babd30>
```



Actual predictions (decision boundaries):

In [23]:
```
mglearn.plots.plot_2d_classification(linea
r_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1],
 y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_s
vm.coef_, linear_svm.intercept_,
                                  mglearn.
cm3.colors):
    plt.plot(line, -(line * coef[0] + inte
rcept) / coef[1], c=color)
plt.legend(['Class 0', 'Class 1', 'Class
 2', 'Line class 0', 'Line class 1',
            'Line class 2'], loc=(1.01, 0.
3))
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Out[23]:
```
<matplotlib.tex
t.Text at 0x11c
c9e978>
```



Strengths, weaknesses and parameters

Regularization parameters:

- Regression: alpha (higher values, simpler models)
  - Ridge (L2), Lasso (L1), LinearRegression (None)
- Classification: C (smaller values, simpler models)
  - LogisticRegression or SVC (both have L1/L2 option)

L1 vs L2:

- L2 is default
- Use L1 if you assume that few features are important
  - Or, if model interpretability is important

Other options:

- ElasticNet regression: allows L1 vs L2 trade-off
- SGDClassifier/SGDRegressor: optimize w_i,b with stochastic gradient descent (more scalable)

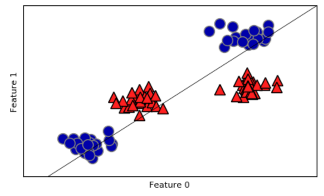
Consider linear models when:

- number of features is large compared to the number of samples
  - other algorithms perform better in low-dimensional spaces
- very large datasets (fast to train and predict)
  - other algorithms become (too) slow

Intuition: why linear models are powerful in high dimension

While linear models are limited on low-dimensional data, they can often fit high dimensional data very well.

In [24]:
```
from sklearn.svm import LinearSVC
X, y = make_blobs(centers=4, random_state=8
)
y = y % 2 # Reduces 4 classes to 2
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm,
 X)
mglearn.discrete_scatter(X[:, 0], X[:, 1],
y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1");
```



While in the previous picture the classes (blue and red) cannot be linearly separated, imagine that we have another dimension that tells us more about each class.

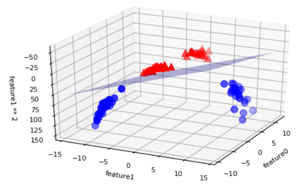
In [25]:
```
# add the square of the first feature (this is just one way
 to add new features)
X_new = np.hstack([X, X[:, 1:] ** 2])


from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# visualize in 3D
ax = Axes3D(figure, elev=-152, azim=-26)
# plot first all the points with y==0, then all with y == 1
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c
='b',
           cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2
], c='r', marker='^',
           cmap=mglearn.cm2, s=60)
ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1 ** 2");
```

Now we can fit a linear model

Note: We will come back to this when discussing *kernelization,* in which we construct new dimensions on purpose.

```
In [26]:  linear_svm_3d = LinearSVC().fit(X_new, y)
          coef, intercept = linear_svm_3d.coef_.ravel
          (), linear_svm_3d.intercept_

          # show linear decision boundary
          figure = plt.figure()
          ax = Axes3D(figure, elev=-152, azim=-26)
          xx = np.linspace(X_new[:, 0].min() - 2, X_n
          ew[:, 0].max() + 2, 50)
          yy = np.linspace(X_new[:, 1].min() - 2, X_n
          ew[:, 1].max() + 2, 50)

          XX, YY = np.meshgrid(xx, yy)
          ZZ = (coef[0] * XX + coef[1] * YY + interce
          pt) / -coef[2]
          ax.plot_surface(XX, YY, ZZ, rstride=8, cstr
          ide=8, alpha=0.3)
          ax.scatter(X_new[mask, 0], X_new[mask, 1],
          X_new[mask, 2], c='b',
                     cmap=mglearn.cm2, s=60)
          ax.scatter(X_new[~mask, 0], X_new[~mask, 1
          ], X_new[~mask, 2], c='r', marker='^',
                     cmap=mglearn.cm2, s=60)

          ax.set_xlabel("feature0")
          ax.set_ylabel("feature1")
          ax.set_zlabel("feature1 ** 2")
```

```
Out[26]:  <matplotlib.te
          xt.Text at 0x1
          1c8db6a0>
```



Uncertainty estimates from classifiers

Classifiers can often provide uncertainty estimates of predictions.
In practice, you are often interested in how certain a classifier is about each class prediction (e.g. cancer treatments).

Scikit-learn offers 2 functions. Often, both are available for every learner, but not always.

- decision_function: returns floating point value for each sample
- predict_proba: return probability for each class

In [27]:

```
# create and split a synthetic dataset
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_blobs
X, y = make_blobs(centers=2, cluster_std=2.5, random_sta
te=8)

# we rename the classes "blue" and "red"
y_named = np.array(["blue", "red"])[y]

# we can call train test split with arbitrary many array
s
# all will be split in a consistent manner
X_train, X_test, y_train_named, y_test_named, y_train, y
_test = \
    train_test_split(X, y_named, y, random_state=0)

# build the logistic regression model
lr = LogisticRegression()
lr.fit(X_train, y_train_named)

mglearn.plots.plot_2d_separator(lr, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y);
```

The Decision Function

In the binary classification case, the return value of decision_function is of shape (n_samples,), and it returns one floating-point number for each sample. The first class (class 0) is considered negative, the other (class 1) positive.

This value encodes how strongly the model believes a data point to belong to the “positive” class.

- Positive values indicate a preference for the "positive" class
- Negative values indicate a preference for the "negative" (other) class

In [28]:

```
# show the first few entries of decisio
n_function
print("Decision function:\n{}".format(l
r.decision_function(X_test)[:6]))
```

```
Decision function:
[ 0.527  4.314  5.
92   2.899  4.751
-7.035]
```

In [29]:

```
# Recover the predictions by looking a
t the sign
print("Thresholded decision function:
\n{}".format(
      lr.decision_function(X_test)[:6]
 > 0))
print("Predictions:\n{}".format(lr.pre
dict(X_test)[:6]))
```

```
Thresholded decision
function:
[ True  True  True
True  True False]
Predictions:
['red' 'red' 'red'
'red' 'red' 'blue']
```

The range of decision_function can be arbitrary, and depends on the data and the model parameters. This makes it sometimes hard to interpret.

In [30]:

```
decision_function = lr.decision_func
tion(X_test)
print("Decision function minimum:
{:.2f} maximum: {:.2f}".format(
      np.min(decision_function), np.
max(decision_function)))
```

```
Decision function mini
mum: -10.48 maximum:
8.61
```

We can visualize the decision function as follows, with the actual decision boundary left and the values of the decision boudaries color-coded on the right. Note how the test examples are labeled depending on the decision function.

```
In [31]: fig, axes = plt.subplots(1, 2, figsize=(13, 5))

         mglearn.tools.plot_2d_separator(lr, X, ax=axes[0], alpha
         =.4,
                                         fill=True, cm=mglearn.cm
         2)
         scores_image = mglearn.tools.plot_2d_scores(lr, X, ax=ax
         es[1],
                                                     alpha=.4, cm
         =mglearn.ReBl)

         for ax in axes:
             # plot training and test points
             mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1],
          y_test,
                                      markers='^', ax=ax)
             mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1
         ], y_train,
                                      markers='o', ax=ax)
             ax.set_xlabel("Feature 0")
             ax.set_ylabel("Feature 1")
         cbar = plt.colorbar(scores_image, ax=axes.tolist())
         cbar.set_alpha(1)
         cbar.draw_all()
         axes[0].legend(["Test class 0", "Test class 1", "Train c
         lass 0",
                         "Train class 1"], ncol=4, loc=(.1, 1.1
         ));
```

Predicting probabilities

The output of predict_proba is a *probability* for each class, with one column per class. They sum up to 1.

In [32]:
```
print("Shape of probabilities: {}".format(lr
.predict_proba(X_test).shape))
# show the first few entries of predict_prob
a
print("Predicted probabilities:\n{}".format(
      lr.predict_proba(X_test[:6])))
```

```
Shape of prob
abilities: (2
5, 2)
Predicted pro
babilities:
[[0.371 0.62
9]
 [0.013 0.98
7]
 [0.003 0.99
7]
 [0.052 0.94
8]
 [0.009 0.99
1]
 [0.999 0.00
1]]
```

We can visualize them again. Note that the gradient looks different now.

In [33]:
```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot_2d_separator(
    lr, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2
)
scores_image = mglearn.tools.plot_2d_scores(
    lr, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl, function
='predict_proba')

for ax in axes:
    # plot training and test points
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y
_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1],
 y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
# don't want a transparent colorbar
cbar = plt.colorbar(scores_image, ax=axes.tolist())
cbar.set_alpha(1)
cbar.draw_all()
axes[0].legend(["Test class 0", "Test class 1", "Train cla
ss 0",
                "Train class 1"], ncol=4, loc=(.1, 1.1));
```

Interpreting probabilities

- The class with the highest probability is predicted.
- How well the uncertainty actually reflects uncertainty in the data depends on the model and the parameters.
  - An overfitted model tends to make more certain predictions, even if they might be wrong.
  - A model with less complexity usually has more uncertainty in its predictions.
- A model is called *calibrated* if the reported uncertainty actually matches how correct it is — A prediction made with 70% certainty would be correct 70% of the time.
  - LogisticRegression returns well calibrated predictions by default as it directly optimizes log-loss
  - Linear SVM are not well calibrated. They are *biased* towards points close to the decision boundary.
- Techniques exist (http://scikit-learn.org/stable/modules/calibration.html) to calibrate models in post-processing. More in the next lecture.

Compare logistic regression and linear SVM

In [34]:
```
from sklearn.svm import SVC
svc = SVC(kernel="linear",C=0.1,probability=True).fit(X_tr
ain, y_train_named)

fig, axes = plt.subplots(1, 2, figsize=(13, 5))

lr_image = mglearn.tools.plot_2d_scores(
    lr, X, ax=axes[0], alpha=.5, cm=mglearn.ReBl, function
='predict_proba')
svc_image = mglearn.tools.plot_2d_scores(
    svc, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl, functio
n='predict_proba')

for ax in axes:
    # plot training and test points
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y
_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1],
 y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
# don't want a transparent colorbar
cbar = plt.colorbar(lr_image, ax=axes.tolist())
cbar.set_alpha(1)
cbar.draw_all()
axes[0].legend(["Test class 0", "Test class 1", "Train cla
ss 0",
                "Train class 1"], ncol=4, loc=(.1, 1.1));
```

Uncertainty in multi-class classification

- decision_function and predict_proba methods also work in the multiclass setting
- always have shape (n_samples, n_classes), except for decision_function in the binary case (which only returns the values for the positive class)

Example on the Iris dataset, which has 3 classes

In [35]:

```
from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_s
plit(
    iris.data, iris.target, random_state=42)

lr2 = LogisticRegression()
lr2 = lr2.fit(X_train, y_train)

print("Decision function:\n{}".format(lr2.decis
ion_function(X_test)[:6, :]))
# show the first few entries of predict_proba
print("Predicted probabilities:\n{}".format(lr2
.predict_proba(X_test)[:6]))
```

```
Decision fu
nction:
[[ -4.744
0.102  -1.0
84]
 [  3.699
-1.937 -10.
976]
 [-10.128
0.898   4.2
62]
 [ -4.504
-0.5    -0.
92 ]
 [ -4.881
0.249  -1.5
12]
 [  3.369
-1.644 -10.
167]]
Predicted p
robabilitie
s:
[[0.011 0.6
68 0.321]
 [0.886 0.1
14 0.   ]
 [0.    0.4
19 0.581]
 [0.016 0.5
61 0.423]
 [0.01  0.7
49 0.241]
```