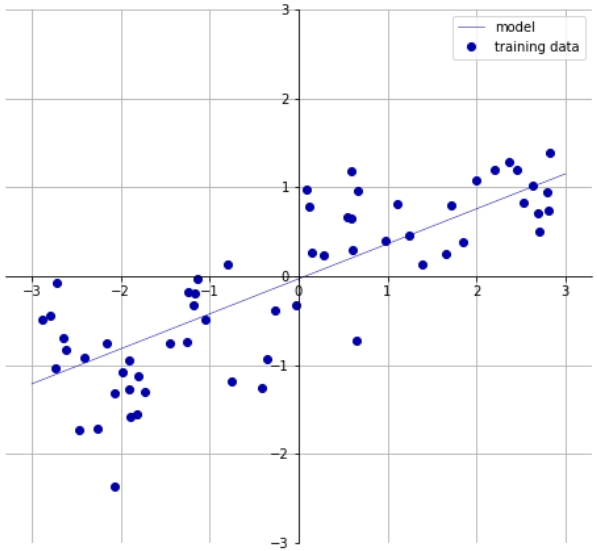
Linear models for regression

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

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

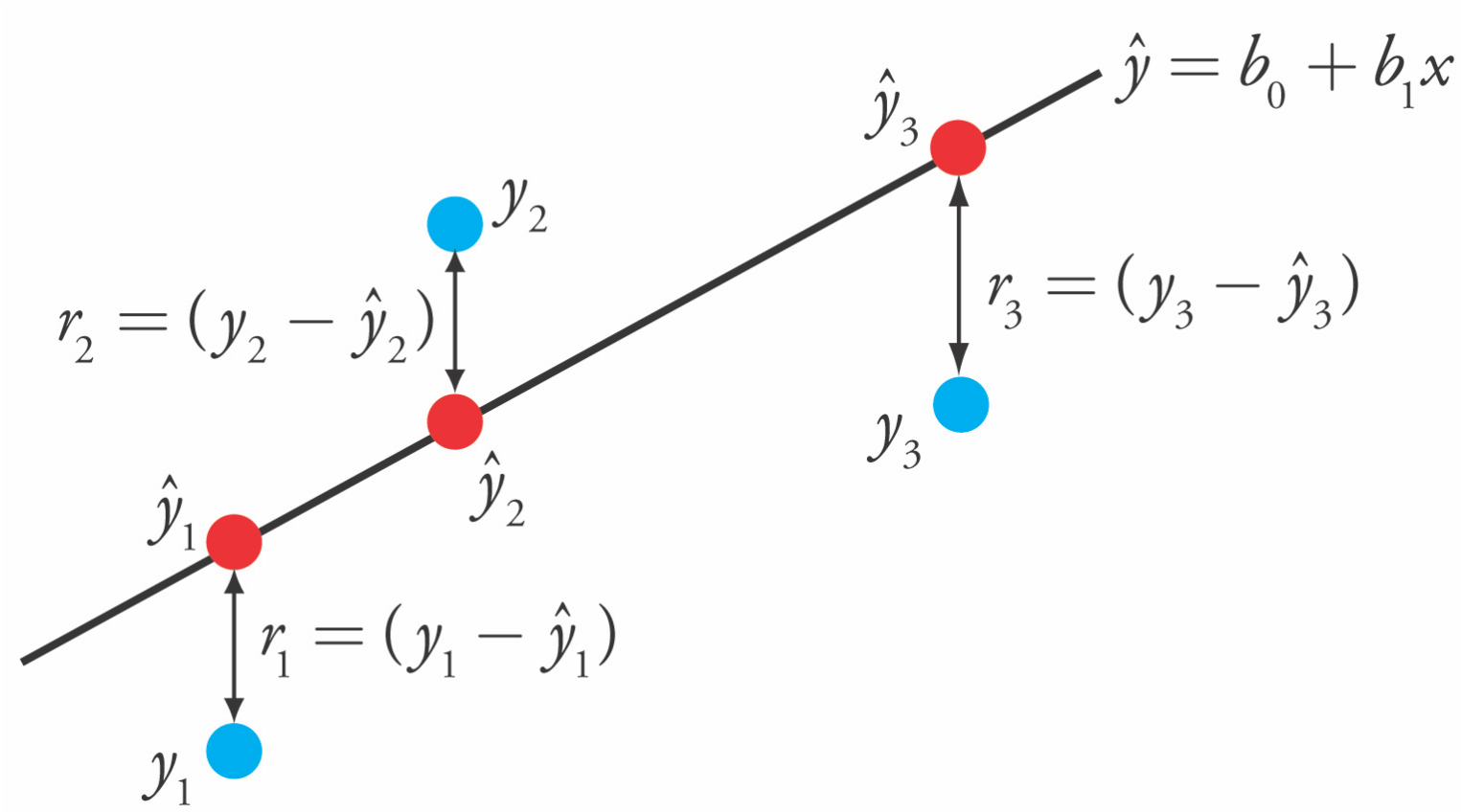
There are many different algorithms, differing in how w and b are learned from the training data.

```
w[0]: 0.393906  b: -0.03180
4
```



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.

```
Weights (coefficients): [ -402.752   -50.071  -133.317   -12.002   -12.
711    28.305    54.492
   -51.734    25.26     36.499   -10.104   -19.629   -21.368    14.647
  2895.054  1510.269   117.995   -26.566    31.249   -31.446    45.254
  1283.496 -2246.003   222.199    -0.466    40.766   -13.436   -19.096
    -2.776   -80.971     9.731     5.133    -0.788    -7.603    33.672
   -11.505    66.267   -17.563    42.983     1.277     0.61     57.187
    14.082    55.34    -30.348    18.812   -13.777    60.979   -12.579
   -12.002   -17.698   -34.028     7.15     -8.41     16.986   -12.941
   -11.806    57.133   -17.581     1.696    27.218   -16.745    75.03
   -30.272    47.78    -40.541     5.504    21.531    25.366   -49.485
    28.109    10.469   -71.559   -23.74      9.574    -3.788     1.214
    -4.72     41.238   -37.702    -2.156   -26.296   -33.202    45.932
   -23.014   -17.515   -14.085   -20.49     36.525   -94.897   143.234
   -15.674   -14.973   -28.613   -31.252    24.565   -17.805     4.035
     1.711    34.474    11.219     1.143     3.737    31.385]
Bias (intercept): 31.645174100828186

Training set score (R^2):
0.95
Test set score (R^2): 0.61
```

Ridge regression

- Same formula as linear regression
- Adds a penalty term to the least squares sum : $\alpha \sum_i w_i^2$
- 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`.

```
ridge = Ridge().fit(X_train, y_train)
```

```
Training set score:
0.89
Test set score: 0.75
```

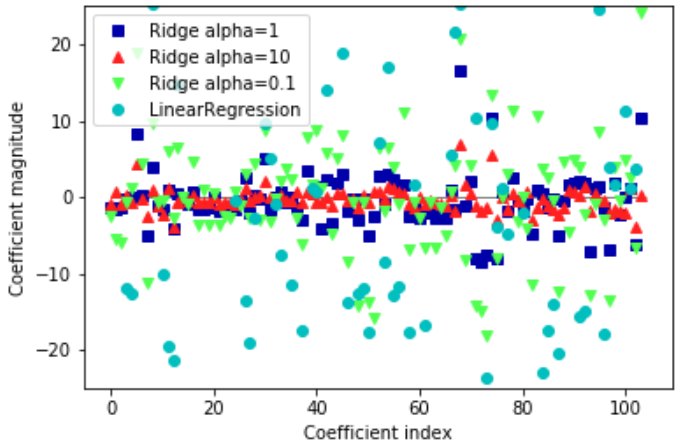
Test set score is higher and training set score lower: less overfitting!

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

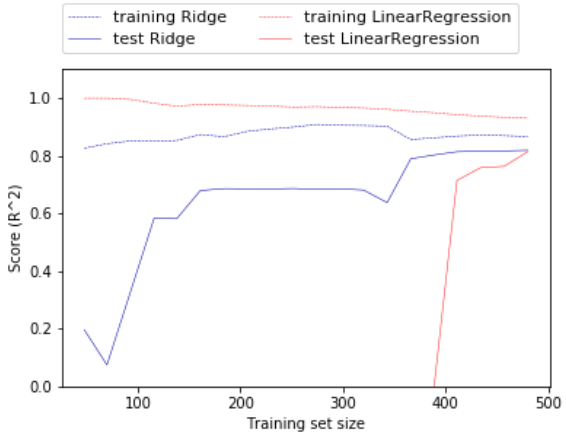
- Increasing alpha forces coefficients to move more toward zero (more regularization)
- Decreasing alpha allows the coefficients to be less restricted (less regularization)

```
Training set score (alpha=10):
0.79
Test set score (alpha=10): 0.64

Training set score (alpha=0.1):
0.93
Test set score (alpha=0.1): 0.77
```

We can plot the weight values for differents levels of regularization. We see that increasing regularizations decreases the values of the coefficients.



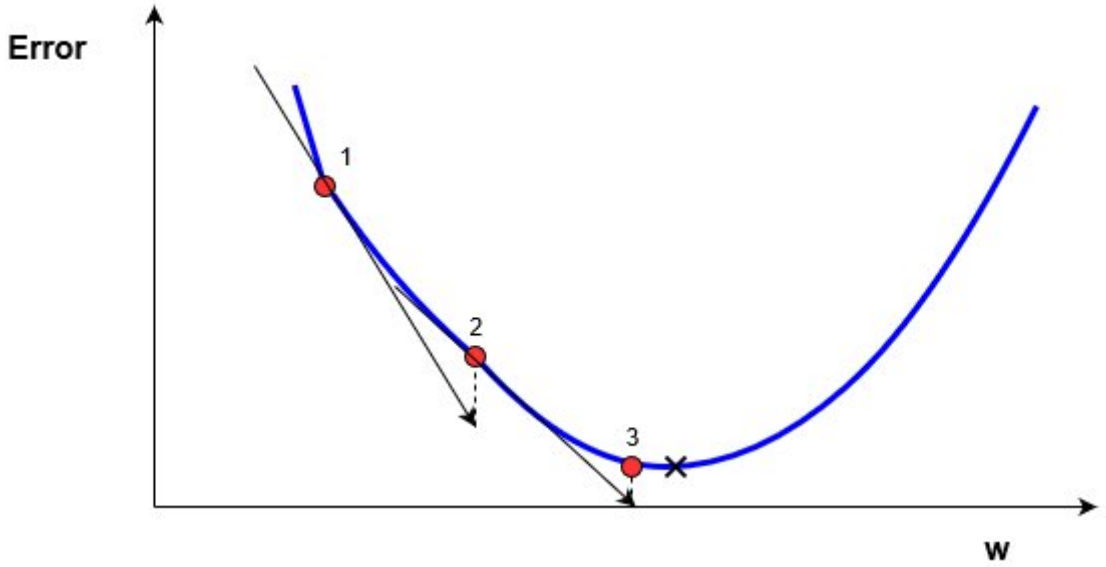
Another way to understand the influence of regularization is to fix a value of alpha but vary the amount of training data available. With enough training data, regularization becomes less important: ridge and linear regression will have the same performance.



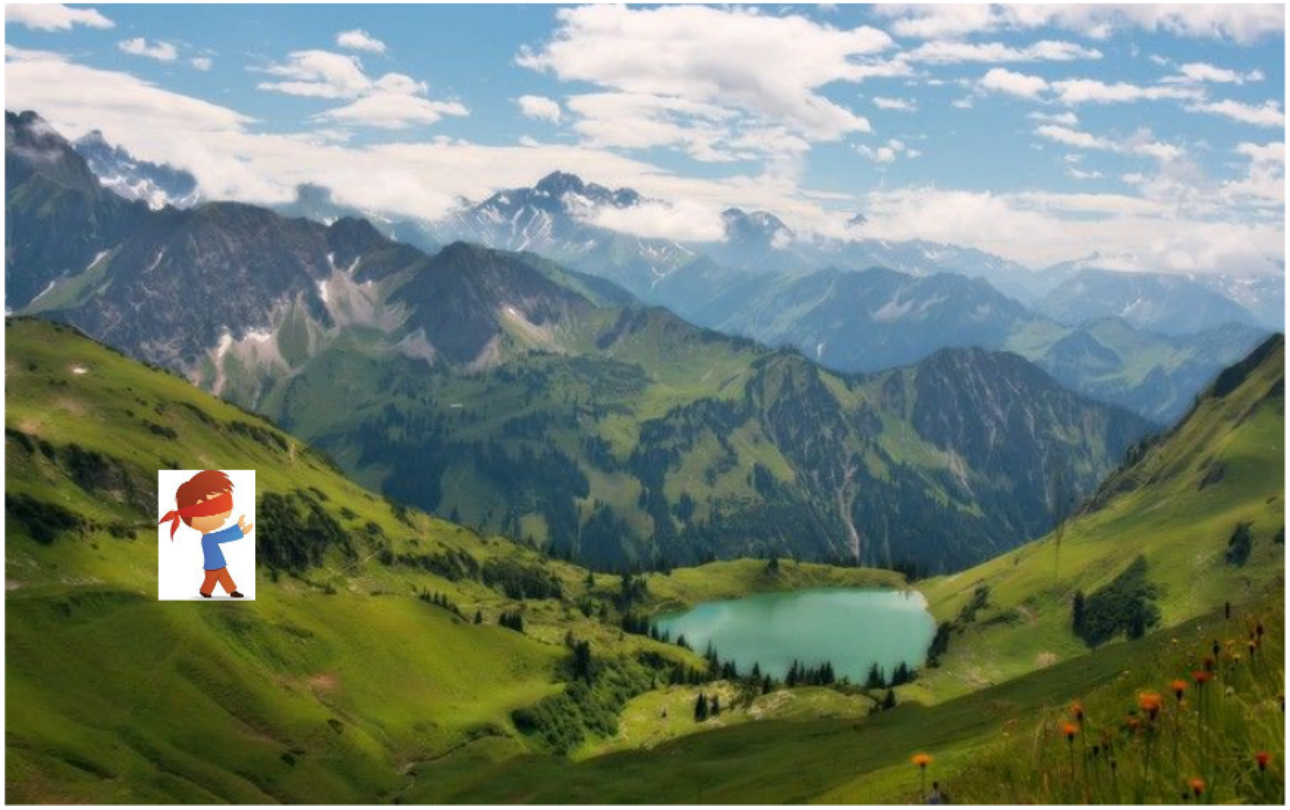
Lasso

- Another form of regularization
- Adds a penalty term to the least squares sum : $\alpha \sum_i |w_i|$
- 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`

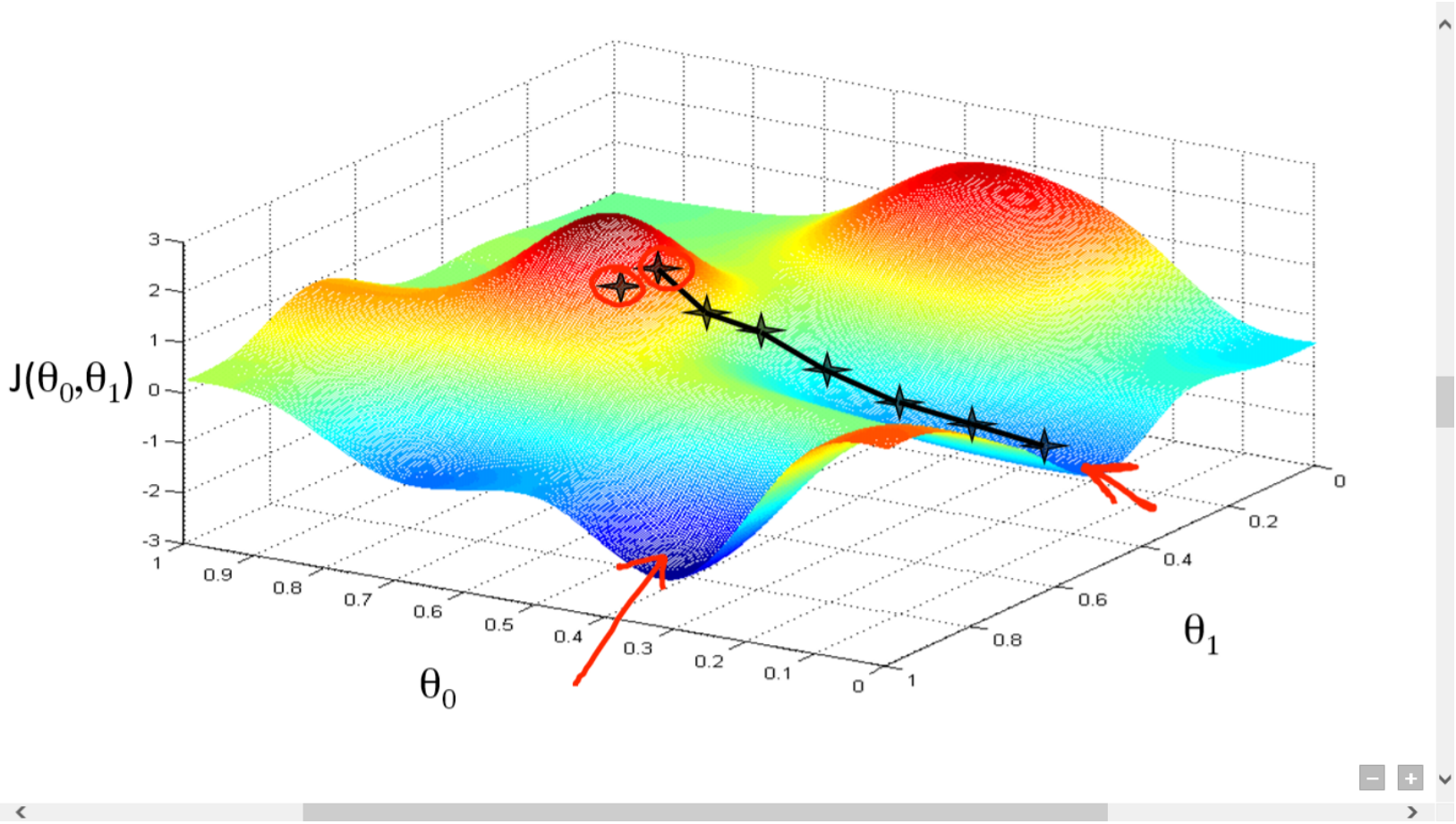
Gradient Descent



Gradient Descent



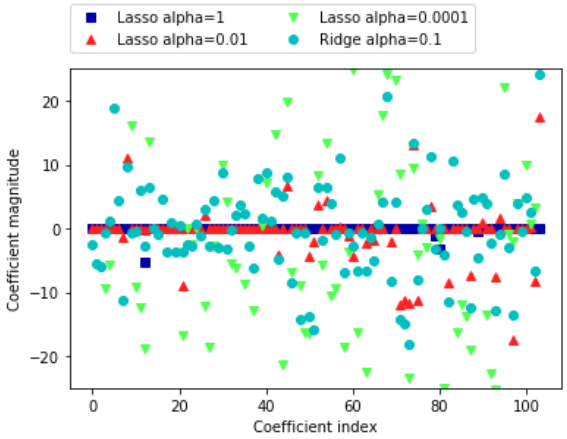
Gradient Descent



```
lasso = Lasso(alpha=1, max_iter=1000).fit(X_train, y_train)
```

```
alpha=1.0, max_iter=10
00
Training set score: 0.
29
Test set score: 0.21
Number of features use
d: 4

alpha=0.01, max_iter=10
0000
Training set score: 0.9
0
Test set score: 0.77
Number of features use
d: 33

alpha=1e-05, max_iter=10
0000
Training set score: 0.95
Test set score: 0.62
Number of features used:
103
```

We can again analyse what happens to the weigths. Increasing regularization under L1 leads to many coefficients becoming exactly 0.



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*:

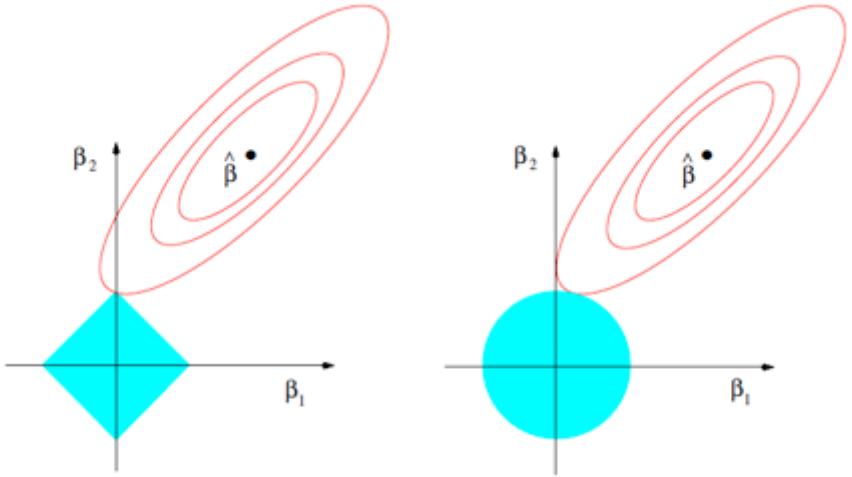


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are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

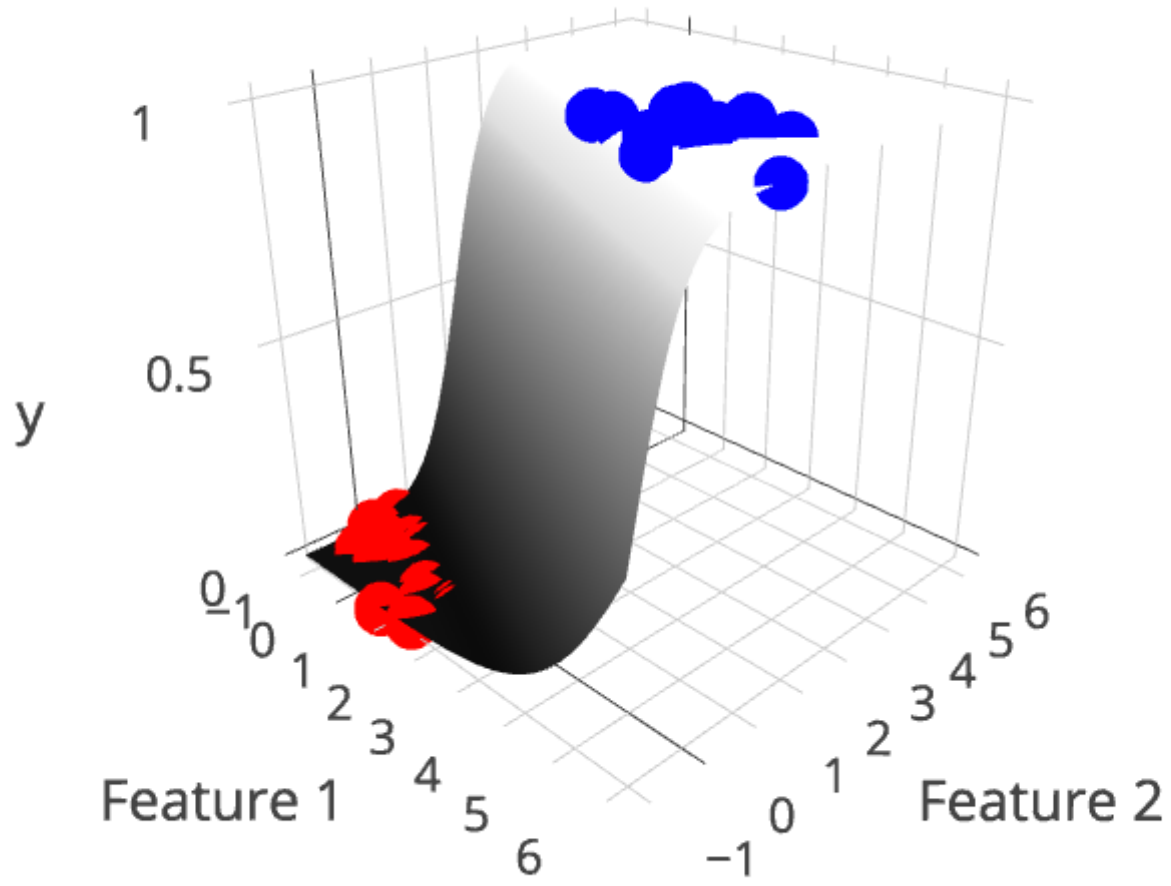
The logistic model uses the *logistic* (or *sigmoid*) function to estimate the probability that a given sample belongs to class 1:

$$z = f(x) = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p$$

$$\hat{y} = Pr[1|x_1, \ldots, x_k] = g(z) = \frac{1}{1 + e^{-z}}$$

- The logistic function is chosen because it maps values (-Inf,Inf) to a probability [0,1]
- We add a new dimension for the dependent variable y and fit the logistic function g(z) so that it separates the samples as good as possible. The positive (blue) points are mapped to 1 and the negative (red) points to 0.
- After fitting, the logistic function provides the probability that a new point is positive. If we need a binary prediction, we can threshold at 0.5.
- There are different ways to find the optimal parameters w that fit the training data best

On 2-dimensional data:



Fitting (solving): cross-entropy

- We define the difference (error) between the actual probabilies (frequencies) p_i and the predicted probabilities q_i is the cross-entropy $H(p, q)$:

$$H(p, q) = -\sum_i p_i log(q_1)$$

- Note: Instead of minimizing cross-entropy $H(p, q)$, you can maximize *log-likelihood* $-H(p, q)$, and hence this is also called *maximum likelihood* estimation
- In binary classification, $i = 0, 1$ and $p_1 = y$, $p_0 = 1 - y$, $q_1 = \hat{y}$, $q_0 = 1 - \hat{y}$
- And thus:

$$H(p, q) = -ylog(\hat{y}) - (1 - y)log(1 - \hat{y})$$

Fitting (solving): cross-entropy loss

- Loss function: the average of all cross-entropies in the sample (of N data points):

$$L(\mathbf{w}) = \sum_{n=1}^{N} H(p_n, q_n) = \sum_{n=1}^{N} \Big[- y_n log(\hat{y_n}) - (1 - y_n) log(1 - \hat{y_n}) \Big]$$

  with

$$\hat{y_n} = \frac{1}{1 + e^{\mathbf{w} \cdot \mathbf{x}}}$$

- This is called *logistic loss*, *log loss* or *cross-entropy loss*
- We can (and should always) add a regularization term, either L1 or L2, e.g. for L2:

$$L'(\mathbf{w}) = L(\mathbf{w}) + \alpha \sum_{i} w_i^2$$

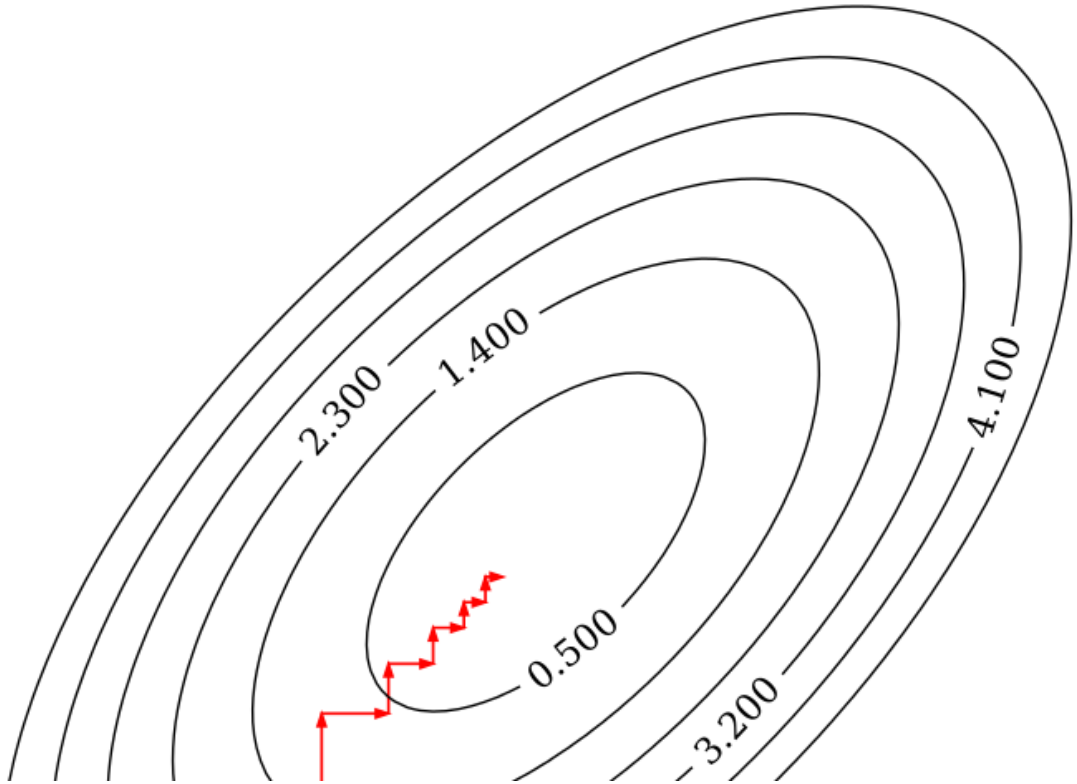
  - Note: in sklearn, the regularization parameter is called C instead of α

Fitting (solving): optimization methods

- There are different ways to optimize cross-entropy loss.
- Gradient descent
  - The logistic function is differentiable, so we can use (stochastic) gradient descent
  - Stochastic Average Gradient descent (SAG): only updates gradient in one direction at each step
- Newton-Rhapson (or Newton Conjugate Gradient):
  - Finds optima by computing second derivatives (more expensive)
  - Works well if solution space is (near) convex
  - Also known as *iterative re-weighted least squares*
- Quasi-Newton methods
  - Approximate, faster to compute
  - E.g. Limited-memory Broyden–Fletcher–Goldfarb–Shanno (`lbfgs`)

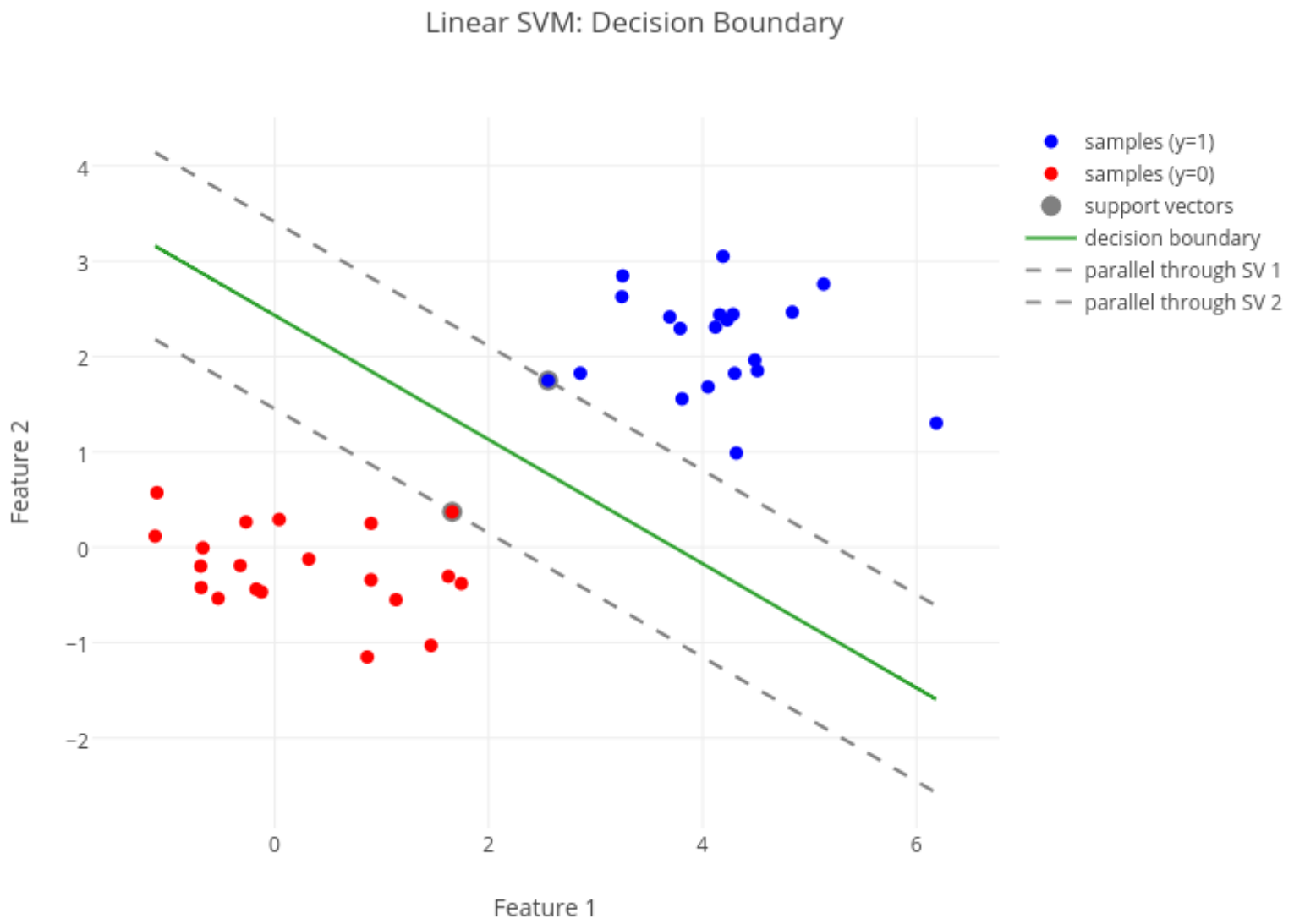
Fitting (solving): optimization methods

- Coordinate descent (default, called `liblinear` in sklearn)
  - In every iteration, optimizes a single coordinate, using a coordinate selection rule (e.g. round robin)
  - Faster iterations, may converge more slowly
  - Applicable for both differential and non-differential loss functions



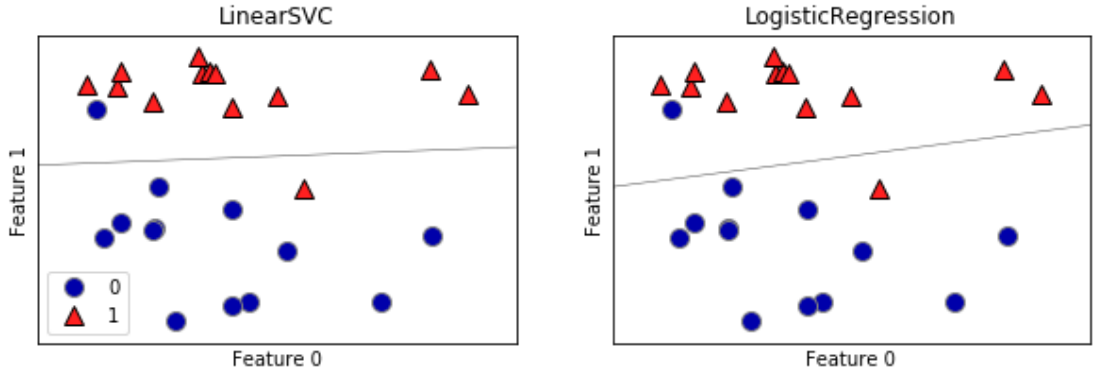
Linear Support Vector Machine

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.

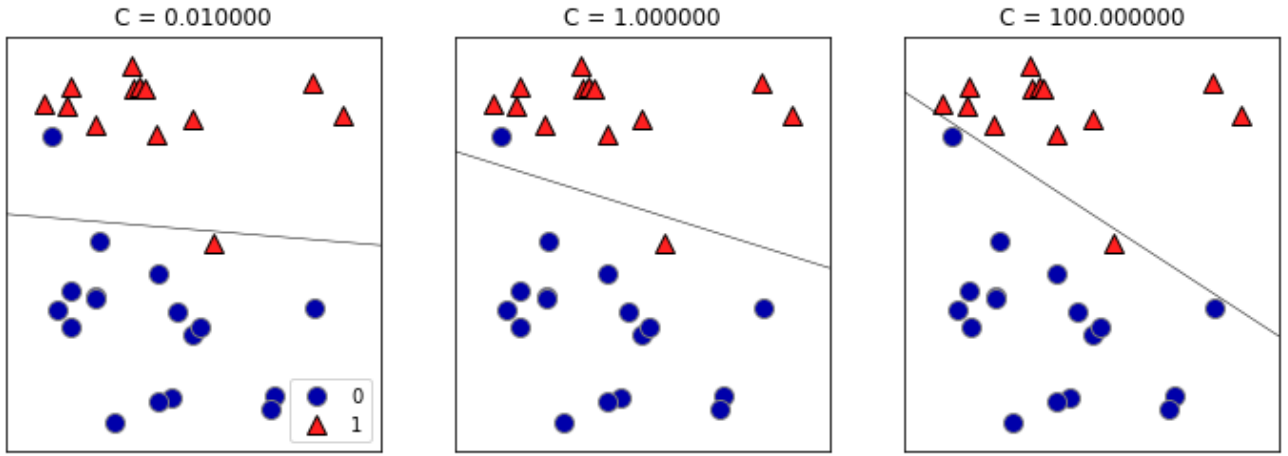
Comparison


LinearSVC
Feature 1
Feature 0
0
1
LogisticRegression
Feature 1
Feature 0

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.

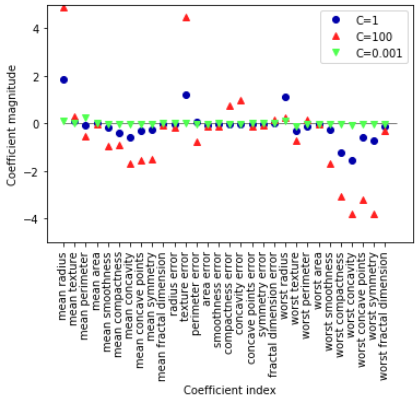


Model selection: Logistic regression

```
logreg = LogisticRegression(C=1).fit(X_train, y_train)
```

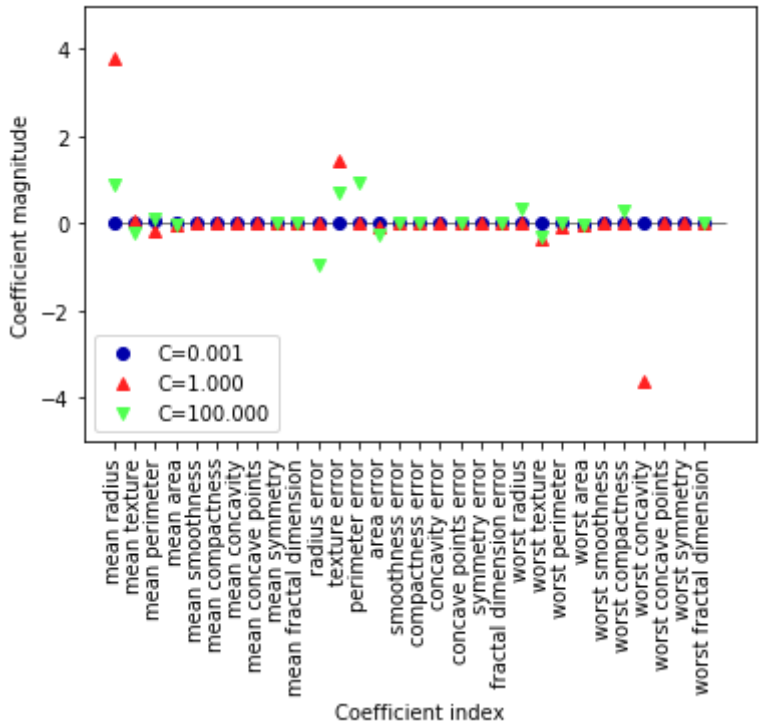
```
C=1.0
Training set score:
0.953
Test set score: 0.958

C=100
Training set score:
0.969
Test set score: 0.965

C=0.01
Training set score:
0.934
Test set score: 0.930
```

Effect of C on model parameters:



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

```
Training accuracy of l1 logreg with C=0.001: 0.
91
Test accuracy of l1 logreg with C=0.001: 0.92
Training accuracy of l1 logreg with C=1.000: 0.
96
Test accuracy of l1 logreg with C=1.000: 0.96
Training accuracy of l1 logreg with C=100.000:
0.99
Test accuracy of l1 logreg with C=100.000: 0.98
```

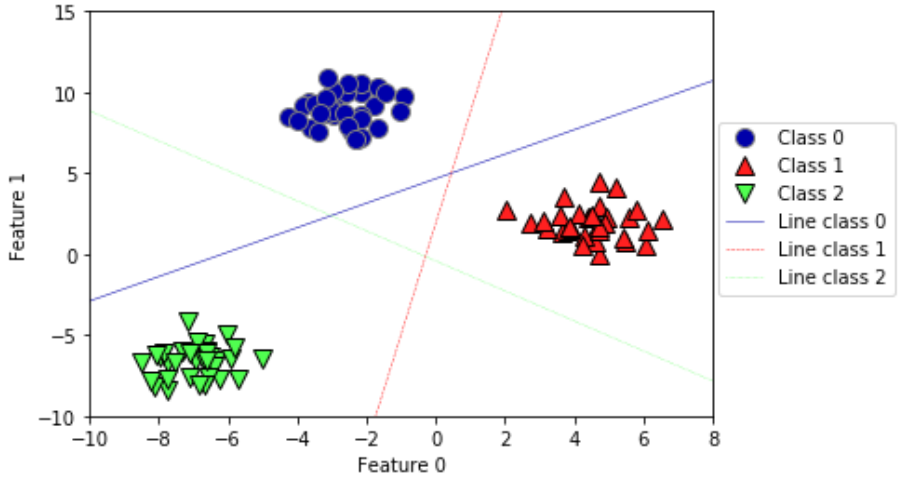


Linear Models for multiclass classification

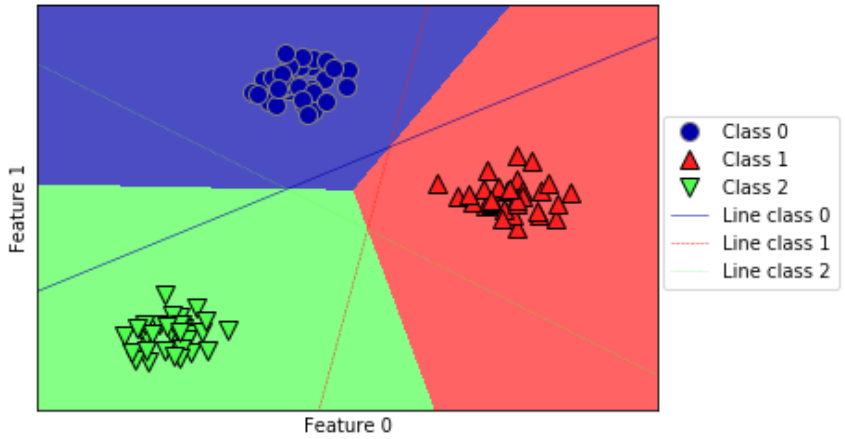
Common technique: one-vs.-rest approach:

- 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:



Actual predictions (decision boundaries):



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:

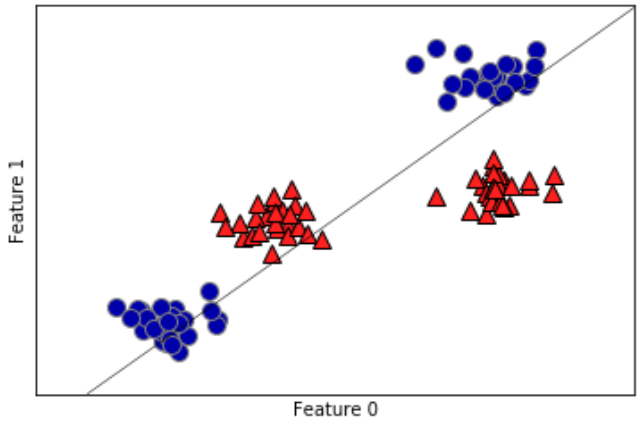
- ElasticNet regression: allows L1 vs L2 trade-off
- SGDClassifier/SGDRegressor: optimize w_i,b with stochastic gradient descent (more scalable)

Consider linear models when:

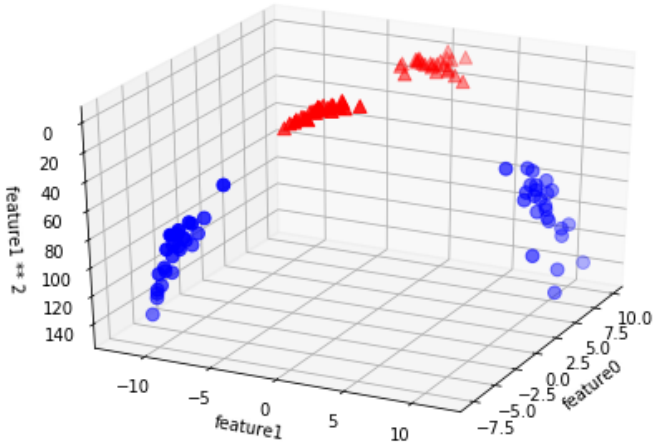
- number of features is large compared to the number of samples
  - other algorithms perform better in low-dimensional spaces
- very large datasets (fast to train and predict)
  - other algorithms become (too) slow

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.

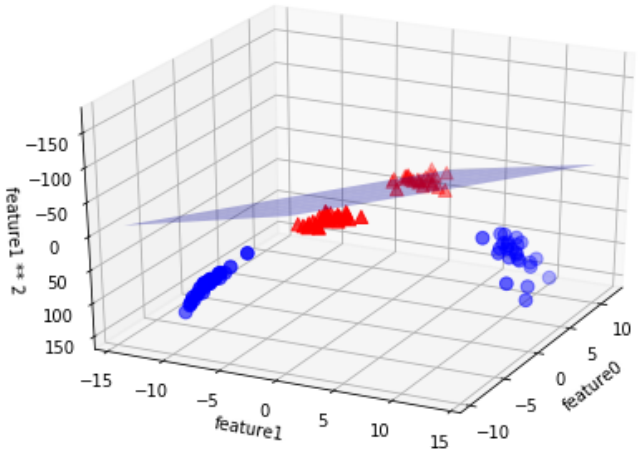


While in the previous picture the classes (blue and red) cannot be linearly separated, imagine that we have another dimension that tells us more about each class.



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.



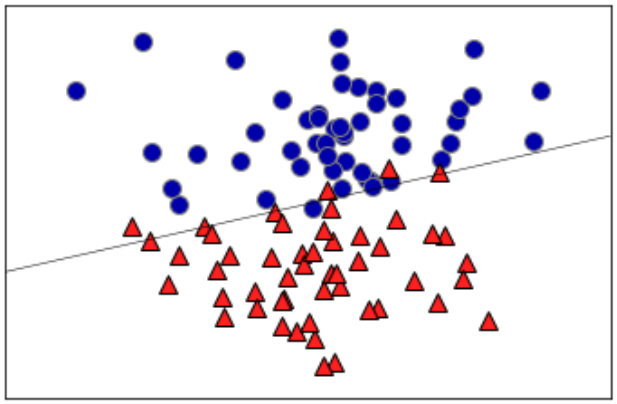
Uncertainty estimates from classifiers

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

Final predictions by `LogisticRegression`:



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.

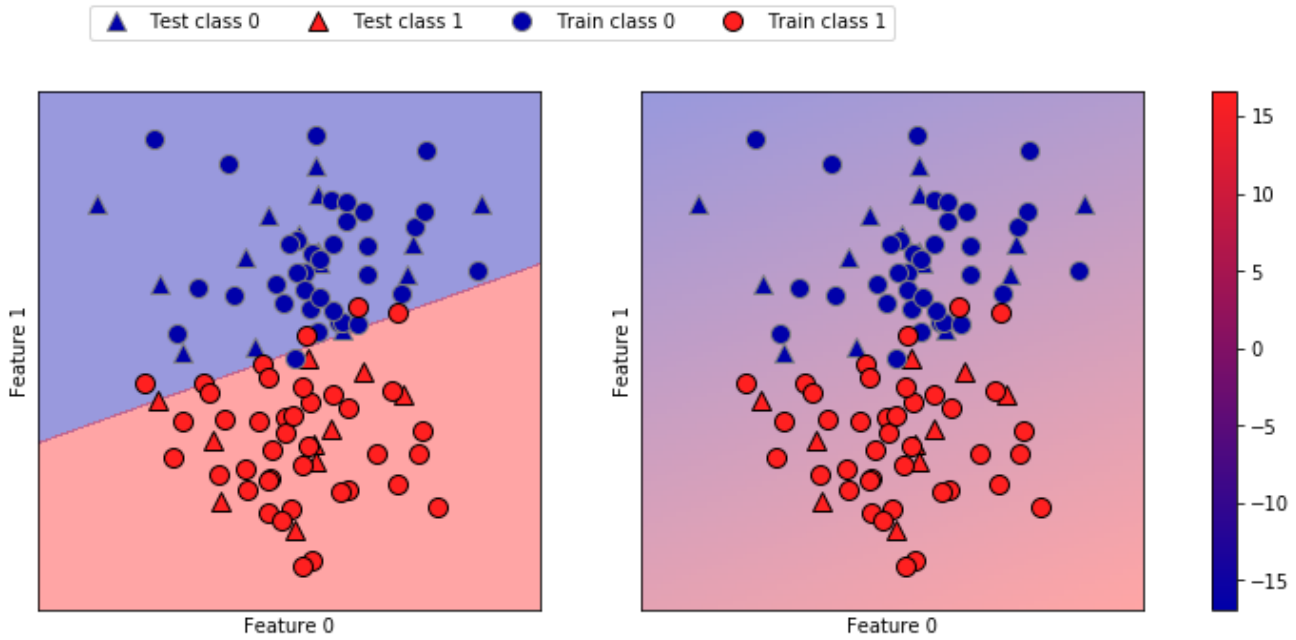
- Positive values indicate a preference for the "positive" class
- Negative values indicate a preference for the "negative" (other) class

```
Decision function:
[ 0.527  4.314  5.92   2.899  4.751 -7.
035]

Thresholded decision function (>
0):
[ True  True  True  True  True Fal
se]
Predictions:
['red' 'red' 'red' 'red' 'red' 'bl
ue']
```

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

```
Decision function minimum: -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.

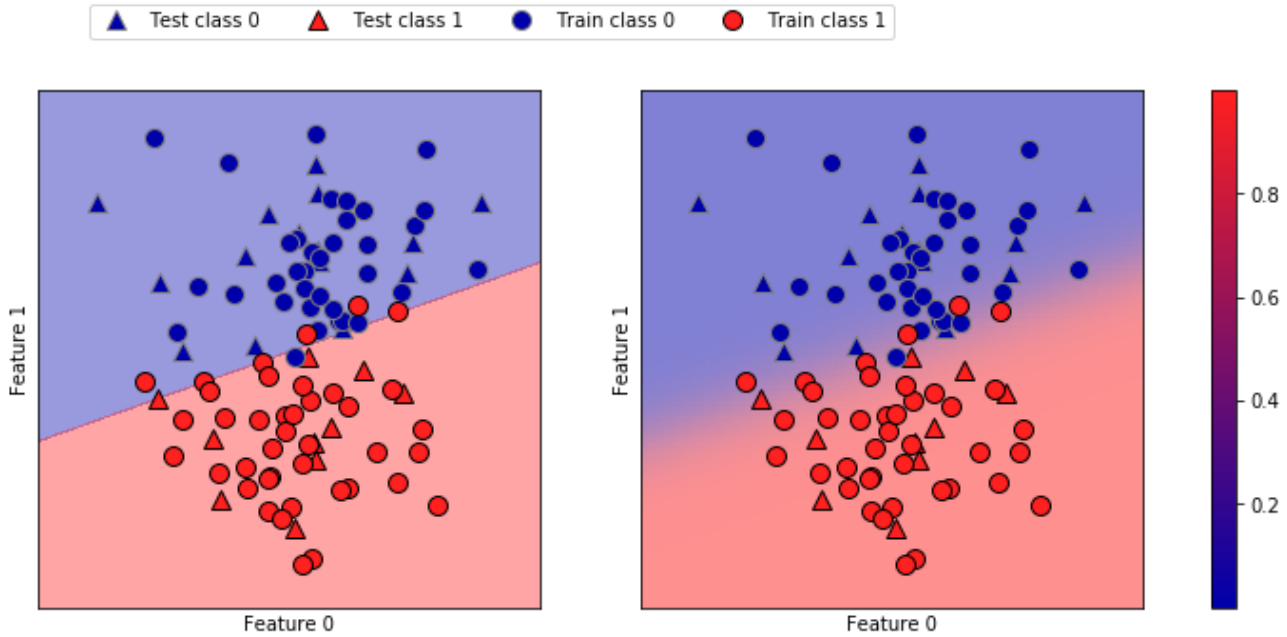


Predicting probabilities

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

```
Shape of probabilities: (2
5, 2)
Predicted probabilities:
[[ 0.371  0.629]
 [ 0.013  0.987]
 [ 0.003  0.997]
 [ 0.052  0.948]
 [ 0.009  0.991]
 [ 0.999  0.001]]
```

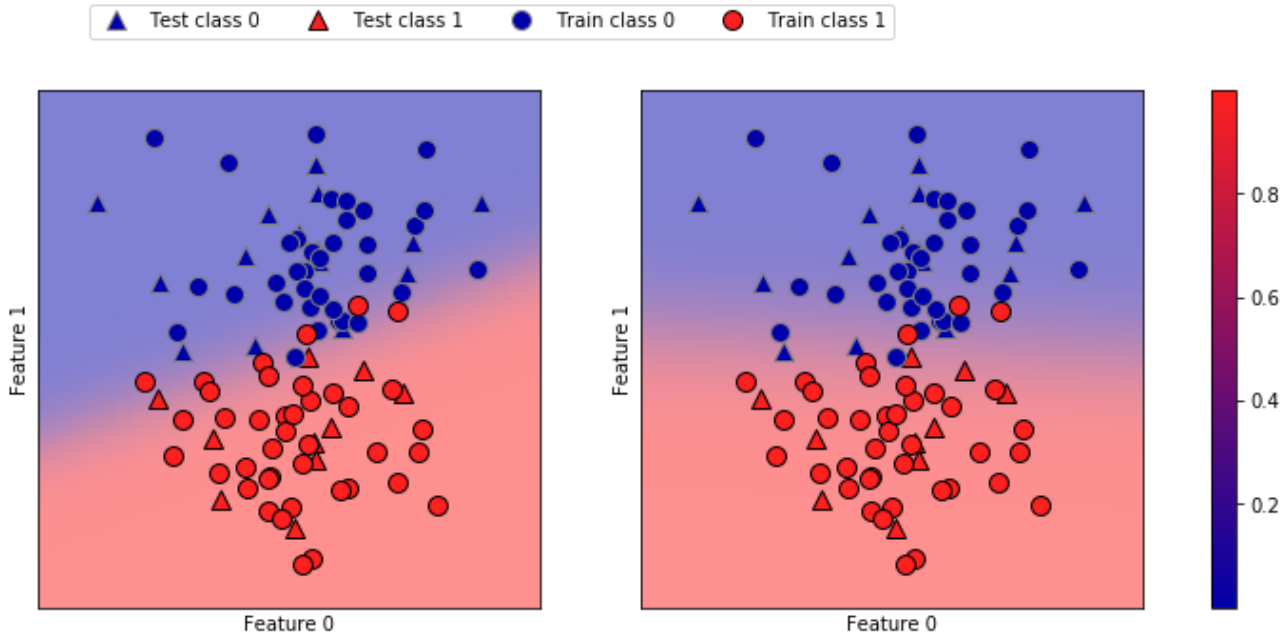
We can visualize them again. Note that the gradient looks different now.



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.
- Calibration techniques (http://scikit-learn.org/stable/modules/calibration.html) can calibrate models in post-processing. They will be covered in later lectures.

Compare logistic regression and linear SVM



Uncertainty in multi-class classification

- `decision_function` and `predict_proba` also work in the multiclass setting
- always have shape (n_samples, n_classes)
- Example on the Iris dataset, which has 3 classes:

```
Decision function:
[[ -4.744   0.102  -1.0
84]
 [  3.699  -1.937 -10.9
76]
 [-10.128   0.898   4.2
62]
 [ -4.504  -0.5    -0.9
2 ]
 [ -4.881   0.249  -1.5
12]
 [  3.369  -1.644 -10.1
67]]
Predicted probabilitie
s:
[[ 0.011  0.668  0.321]
 [ 0.886  0.114  0.   ]
 [ 0.     0.419  0.581]
 [ 0.016  0.561  0.423]
 [ 0.01   0.749  0.241]
 [ 0.857  0.143  0.
]]
```

Summary

- Nearest neighbors
  - For small datasets, excellent baseline, easy to explain.
- Linear models
  - Go-to as a first algorithm to try, good for very large datasets, good for very high-dimensional data.
- Many more models to come
- We first need to learn how to select between them (and their hyperparameters): model selection