Lecture 2: Linear models

Joaquin Vanschoren, Eindhoven University of Technology

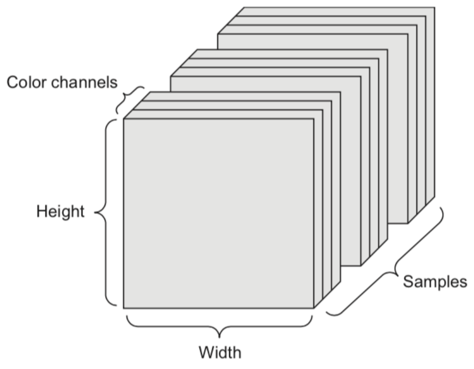
The Mathematics behind Machine Learning

- To understand machine learning algorithms, it often helps to describe them mathematically.
- To avoid confusion, let's specify a precise notation

Basic notation

- A *scalar* is a simple numeric value, denoted by italic letter: $x = 3.24$
- A *vector* is a 1D ordered array of n scalars, denoted by bold letter: $\mathbf{x} = [3.24, 1.2]$
    - A vector can represent a *point* in an n-dimensional space, given a *basis*.
    - x_i denotes the ith element of a vector, thus $x_0 = 3.24$.
        - Note: some other courses use $x^{(i)}$ notation
- A *set* is an *unordered* collection of unique elements, denote by caligraphic capital: $\mathcal{S} = \{3.24, 1.2\}$
- A *matrix* is a 2D array of scalars, denoted by bold capital:
  $\mathbf{X} = \begin{bmatrix} 3.24 & 1.2 \\ 2.24 & 0.2 \end{bmatrix}$
    - It can represent a set of points in an n-dimensional space, given a *basis*.
    - $\mathbf{X}_i$ denotes the ith *row* of the matrix
    - $\mathbf{X}_{i,j}$ denotes the *element* in the ith row, jth column, thus $\mathbf{X}_{0,1} = 2.24$
- The *standard basis* for a Euclidean space is the set of unit vectors
    - Data can also be represented in a non-standard basis (e.g. polynomials) if useful

- A *tensor* is an *k*-dimensional array of data, denoted by an italic capital: $\boldsymbol{T}$
    - *k* is also called the *order*, *degree*, or *rank*
    - $\boldsymbol{T}_{i,j,k,\ldots}$ denotes the element or sub-tensor in the corresponding position
    - A set of color images can be represented by:
        - a 4D tensor (sample x height x weight x color channel)
        - a 2D tensor (sample x flattened vector of pixel values)



Basic operations

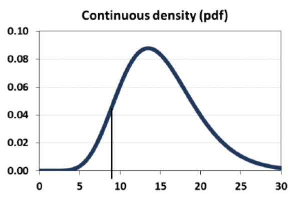
- Sums and products are denoted by capital Sigma and capital Pi:
$$\sum_{i=0}^{n} = x_0 + x_1 + \ldots + x_p \qquad \prod_{i=0}^{n} = x_0 \cdot x_1 \cdot \ldots \cdot x_p$$
- Operations on vectors are *element-wise*: e.g.
$\mathbf{x} + \mathbf{z} = [x_0 + z_0, x_1 + z_1, \ldots, x_p + z_p]$
- Dot product
$\mathbf{wx} = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{p} w_i \cdot x_i = w_0 \cdot x_0 + w_1 \cdot x_1 + \ldots + w_p \cdot x_p$
- Matrix product $\mathbf{Wx} = \begin{bmatrix} \mathbf{w_0} \cdot \mathbf{x} \\ \ldots \\ \mathbf{w_p} \cdot \mathbf{x} \end{bmatrix}$
- A function $f(x) = y$ relates an input element x to an output y
  - It has a *local minimum* at $x = c$ if $f(x) \geq f(c)$ in interval $(c - \epsilon, c + \epsilon)$
  - It has a *global minimum* at $x = c$ if $f(x) \geq f(c)$ for any value for x
- A vector function consumes an input and produces a vector: $\mathbf{f}(\mathbf{x}) = \mathbf{y}$
- $\max\limits_{x \in X} f(x)$ returns the highest value f(x) for any x
- $\operatorname{argmax}\limits_{c \in C} f(x)$ returns the element c that maximizes f(c)

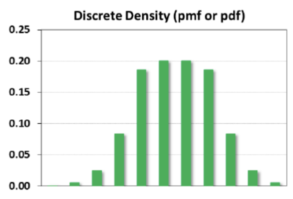
Gradients

- A *derivative* f' of a function f describes how fast f grows or decreases
- The process of finding a derivative is called differentiation
  - Derivatives for basic functions are known
  - For non-basic functions we use the *chain rule*: $F(x) = f(g(x)) \rightarrow F'(x) = f'(g(x))g'(x)$
- A function is *differentiable* if it has a derivate in any point of it's domain
  - It's *continuously differentiable* if f' is itself a function
  - It's *smooth* if $f', f'', f''', \ldots$ all exist
- A *gradient* ∇f is the derivate of a function in multiple dimensions
  - It is a vector of *partial derivatives*: $\nabla f = \left[\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \ldots\right]$
  - E.g. $f = 2x_0 + 3x_1^2 - \sin(x_2) \rightarrow \nabla f = [2, 6x_1, -cos(x_2)]$

Probabilities

- A random variable X can be continuous or discrete
- A probability distribution of a discrete variable is the list of probabilities for each possible value
    - Also called the *probability mass function* (*pmf*)
    - The *expectation* (or *mean*) $\mathbb{E}[X] = \mu_X = \sum_{i=1}^{k}[x_i \cdot \boldsymbol{Pr}(X = x_i)]$
    - The standard deviation $\sigma = \sqrt{\mathbb{E}[(X-\mu)^2]}$ and variance $var(X) = \sigma^2$
- A probability distribution f_X of a continuous variable X is described by a *probability density function* (*pdf*)
    - The *expectation* is given by $\mathbb{E}[X] = \int_{\mathbb{R}} x f_X(x) dx$





- f_x is usually unknown, we only have a sample of the data $S_X = \{x_i\}_{i=1}^N$
- An *unbiased estimator* $\hat{\theta}(S_X)$ of some statistic θ calculated using a sample (S_X) drawn from its unknown probability distribution has the property $\mathbb{E}[\hat{\theta}(S_X)] = \theta$
  - It can be shown that the *sample mean* $\frac{\sum_{i=1}^N x_i}{N}$ is an unbiased estimator of an unknown $\mathbb{E}[X]$

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.

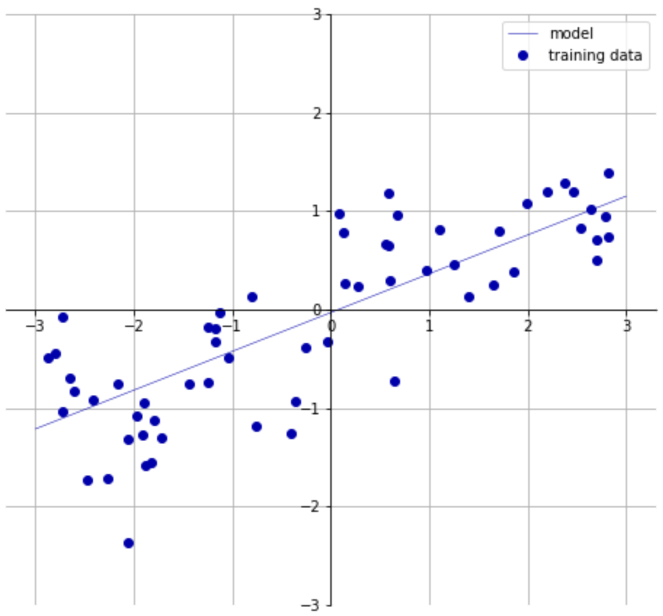
Linear models for regression

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

$$\hat{y} = \mathbf{w}\mathbf{x} + b = \sum_{i=0}^{p} w_i \cdot x_i + b = w_0 \cdot x_0 + w_1 \cdot x_1 + \ldots + w_p \cdot x_p + b$$

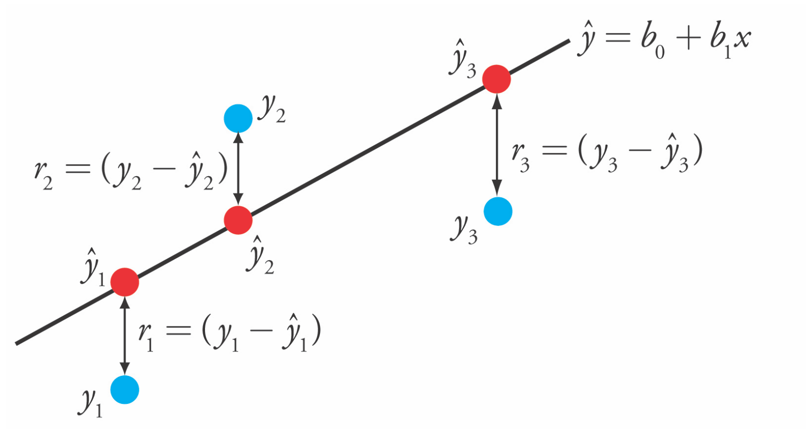
There are many different algorithms, differing in how w and b are learned from the training data.

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



Linear Regression aka Ordinary Least Squares

- Finds the parameters w and b that minimize the *mean squared error* between predictions (red) and the true regression targets (blue), y, on the training set.
    - MSE: Sum of the squared differences between the predictions and the true values (residuals r).
- 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.

```
lr = LinearRegression().fit(X_train, y_train)
```

```
Weights (coefficients): [ -412.711   -52.243  -131.899   -12.004   -15.511
28.716    54.704
   -49.535    26.582    37.062   -11.828   -18.058   -19.525    12.203
  2980.781  1500.843   114.187   -16.97     40.961   -24.264    57.616
  1278.121 -2239.869   222.825    -2.182    42.996   -13.398   -19.389
    -2.575   -81.013     9.66      4.914    -0.812    -7.647    33.784
   -11.446    68.508   -17.375    42.813     1.14  ]
Bias (intercept): 30.93456367364464

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!

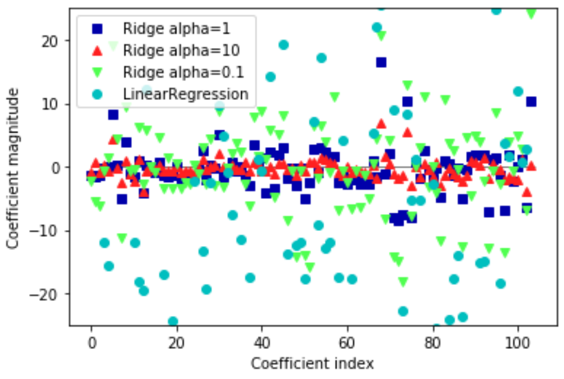
The strength of the regularization can be controlled with the `alpha` parameter. Default is 1.0.

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

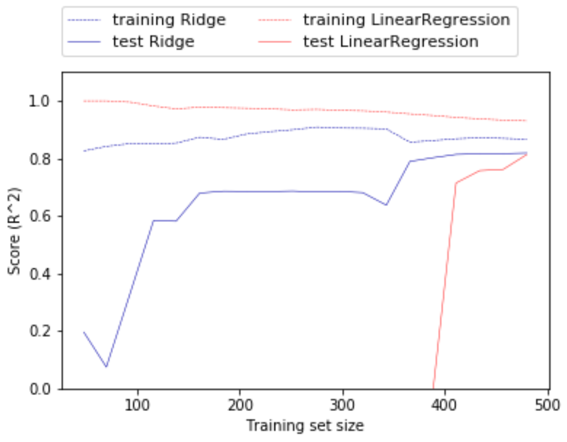
```
Training set score (alpha=10): 0.79
Test set score (alpha=10): 0.64
```

```
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.

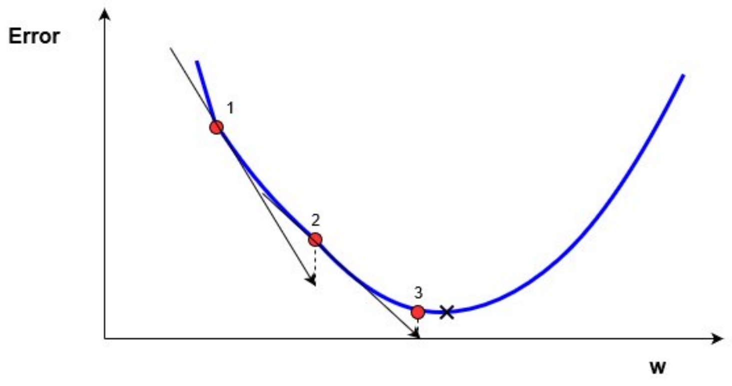


Lasso

- 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.
- Weights are optimized using gradient descent
- New parameter `max_iter`: the maximum number of gradient descent iterations
    - Should be higher for small values of `alpha`

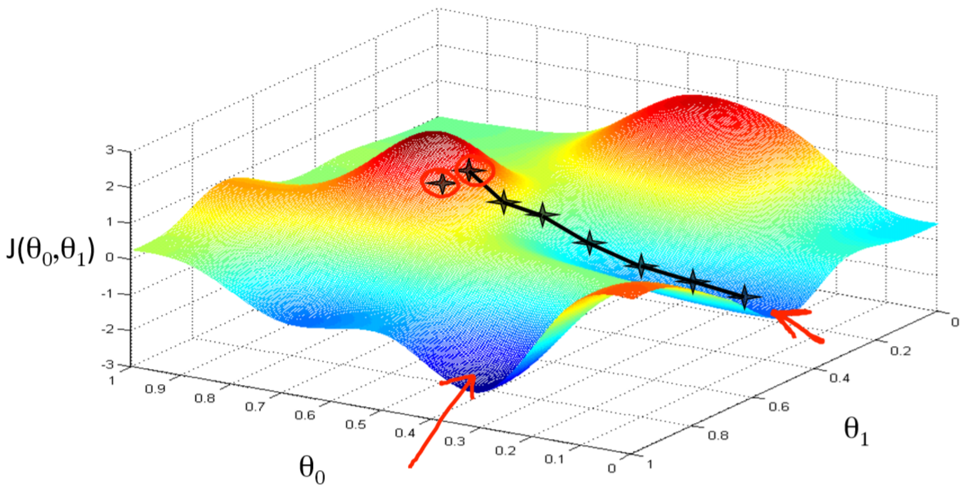
Gradient Descent

- Start with a random set of p weights values
- Compute the derivative of the objective function and use it to find the slope (in p dimensions)
- Update all weights slightly in the direction of the downhill slope
- Repeat for `max_iter` iterations
- Visualization in 1 dimension (for 1 weight):



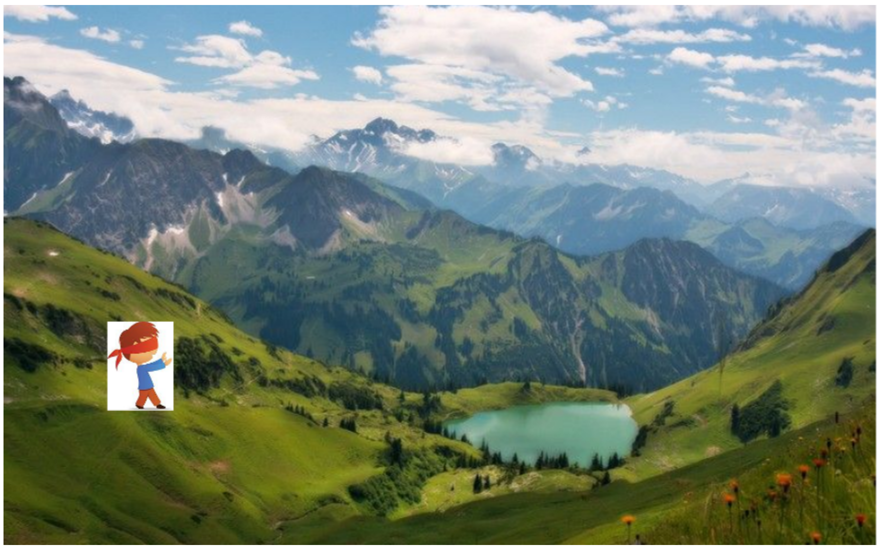
Gradient Descent

In two dimensions:



Gradient Descent

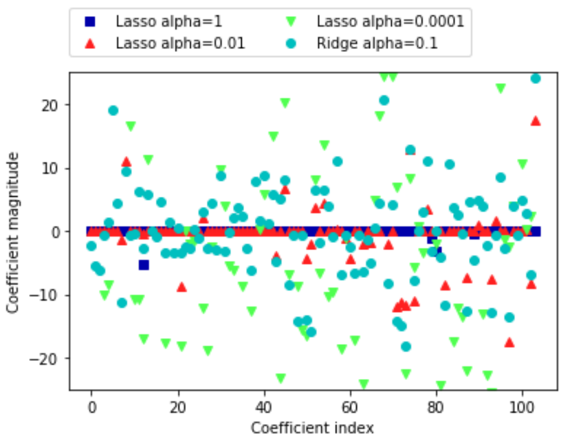
- Intuition: walking downhill using only the slope you "feel" nearby



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

```
alpha=1.0, max_iter=1000
Training set score: 0.29
Test set score: 0.21
Number of features used: 4

alpha=0.01, max_iter=100000
Training set score: 0.90
Test set score: 0.77
Number of features used: 33

alpha=1e-05, max_iter=100000
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
- 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*:

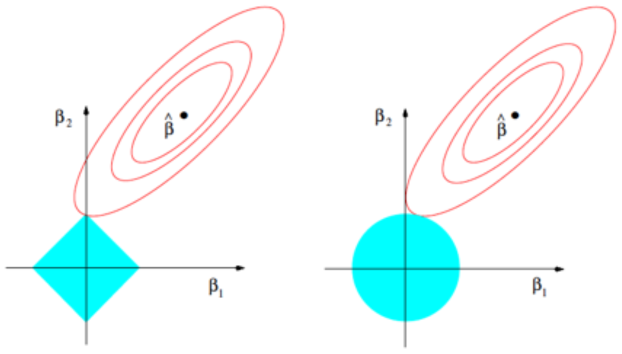


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions $|\beta_1| + |\beta_2| \leq t$ and $\beta_1^2 + \beta_2^2 \leq t^2$, respectively, while the red ellipses are the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

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

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:

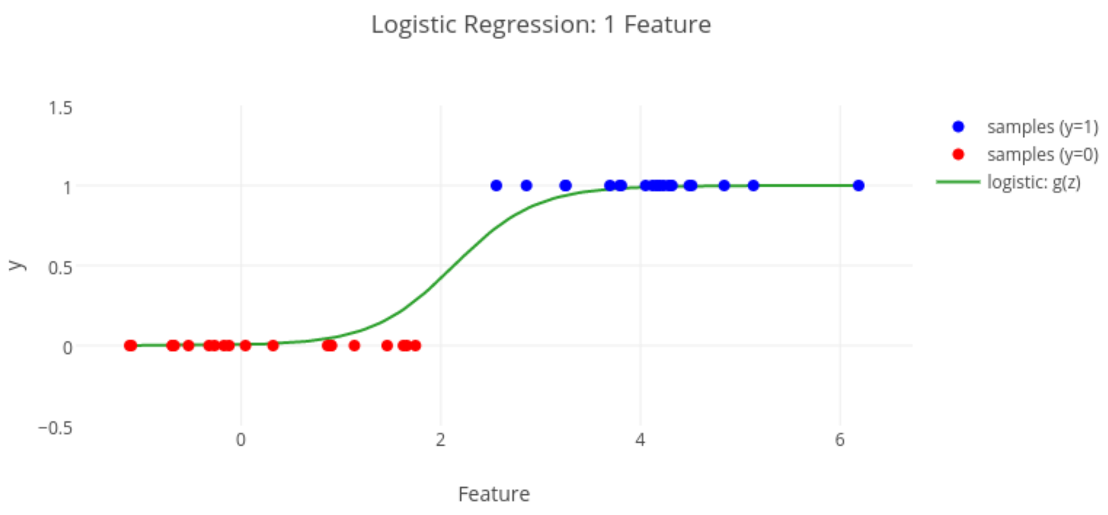
- Logistic regression:
    - `sklearn.linear_model.LogisticRegression`
- Linear Support Vector Machine:
    - `sklearn.svm.LinearSVC`

Logistic regression

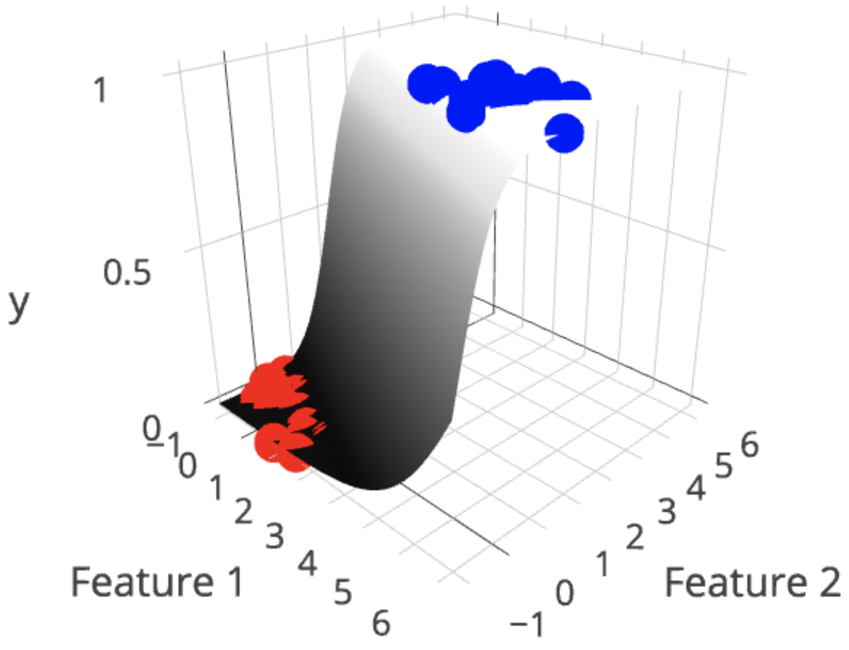
The logistic model uses the *logistic* (or *sigmoid*) function to estimate the probability that a given sample belongs to class 1:

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

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



On 2-dimensional data:



- 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

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

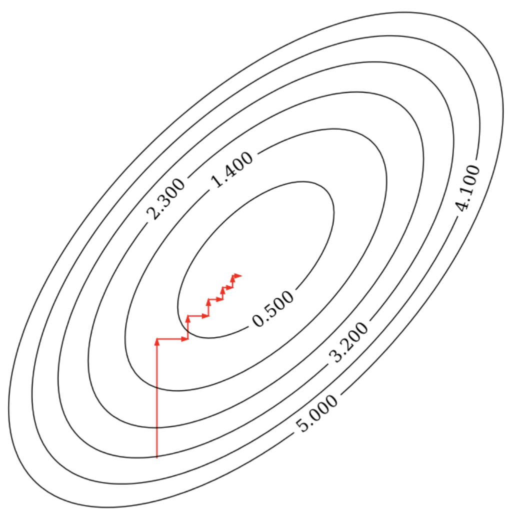
- Loss function: the average of all cross-entropies in the sample (of N data points):
$$L(\mathbf{w}) = \sum_{n=1}^{N} H(p_n, q_n) = \sum_{n=1}^{N} \Big[- y_n log(\hat{y_n}) - (1 - y_n) log(1 - \hat{y_n}) \Big]$$
with
$$\hat{y_n} = \frac{1}{1 + e^{\mathbf{w} \cdot \mathbf{x}}}$$
- This is called *logistic 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`)

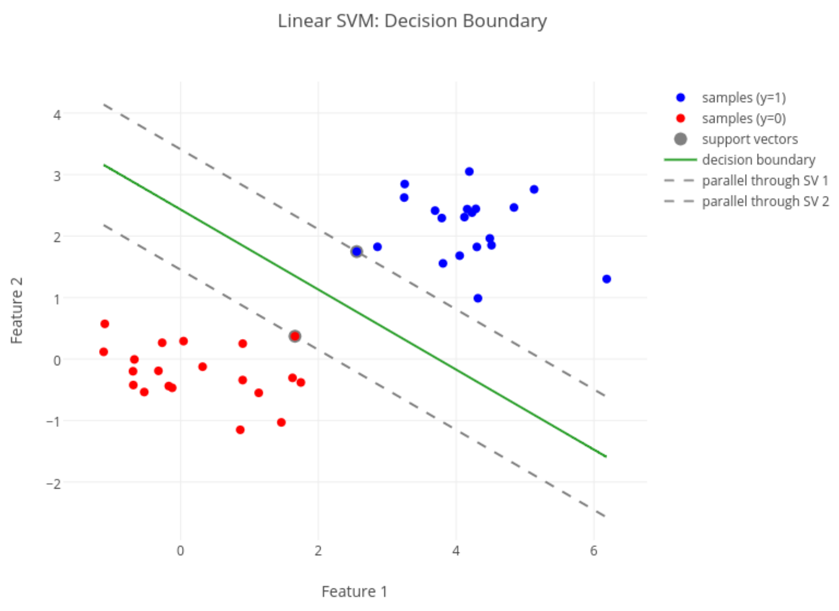
Fitting (solving): optimization methods

- 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 (intuition)

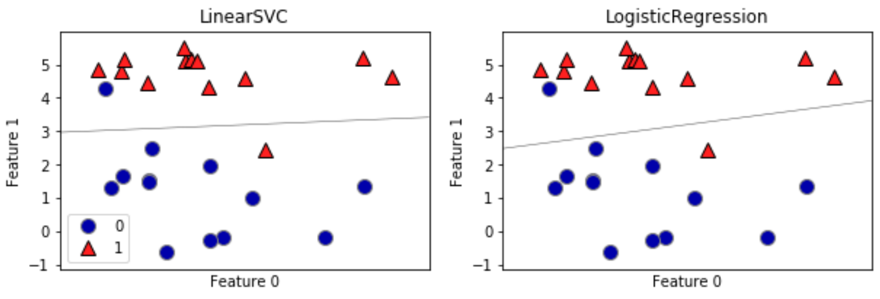
Find hyperplanes (dashed lines) maximizing the *margin* between the classes



Optimization and prediction

- Prediction is identical to (weighted) kNN:
    - Points closest to the red support vector are classified red, others blue
    - A support vector can also have a weight (see later)
- The objective function penalizes every point predicted to be on the `wrong` side of its hyperplane
    - This is called *hinge loss*
- This results in a convex optimization problem solved using the *Langrange Multipliers* method
    - Can also be solved using gradient descent
- This will all be discussed at length later

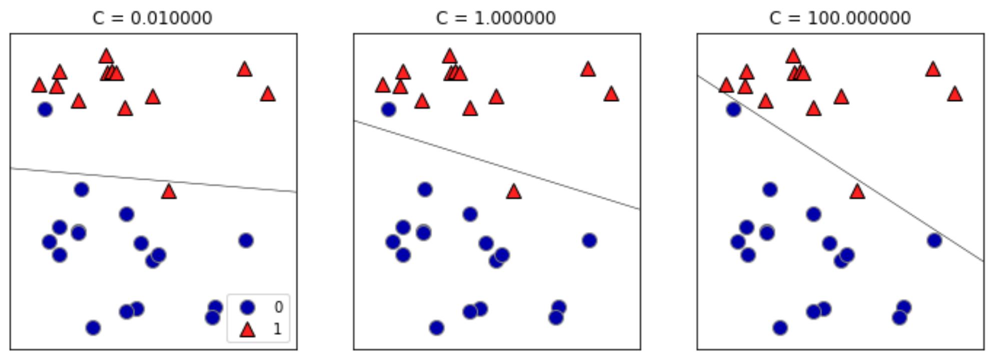
Comparison


LinearSVC
LogisticRegression
Feature 0
Feature 1
5
4
3
2
1
0
−1
0
1

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.

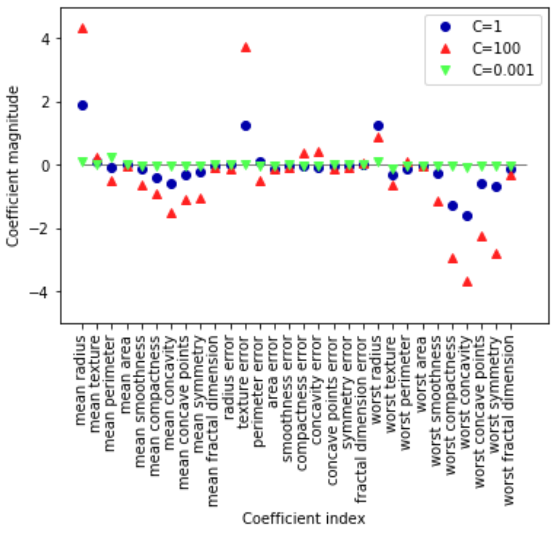


Model selection: Logistic regression

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

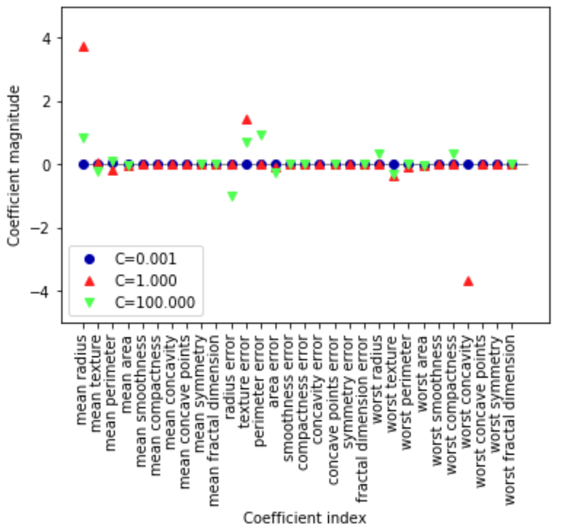
```
C=1.0
Training set score: 0.955
Test set score: 0.958

C=100
Training set score: 0.967
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
```

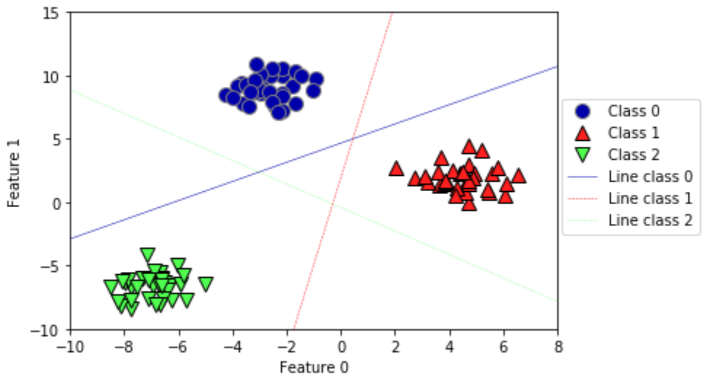


Linear Models for multiclass classification

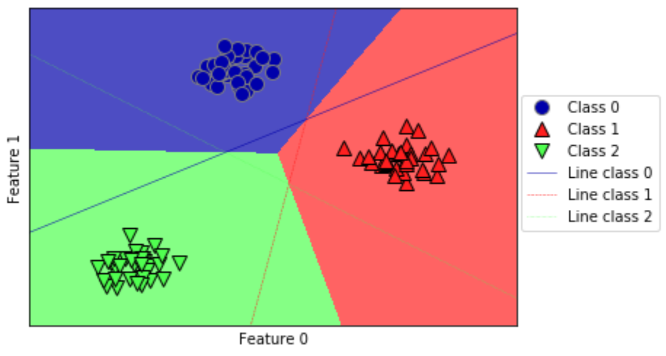
Common technique: one-vs.-rest approach:

- 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:

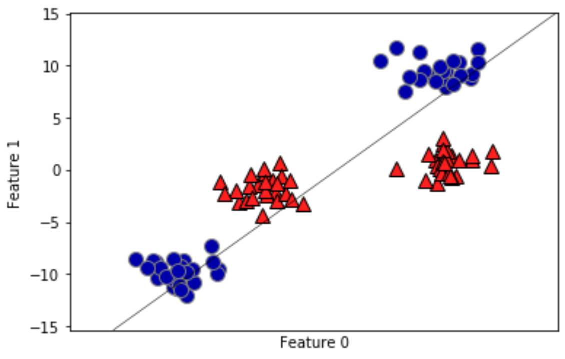
- ElasticNet regression: allows L1 vs L2 trade-off
- SGDClassifier/SGDRegressor: optimize w_i,b with stochastic gradient descent (more scalable)

Consider linear models when:

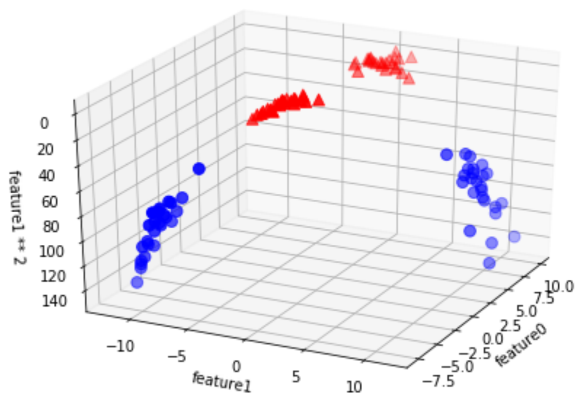
- number of features is large compared to the number of samples
    - other algorithms perform better in low-dimensional spaces
- very large datasets (fast to train and predict)
    - other algorithms become (too) slow

Intuition: why linear models are powerful in high dimension

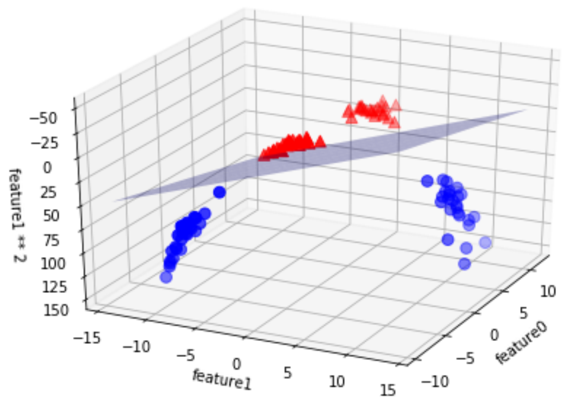
While linear models are limited on low-dimensional data, they can often fit high dimensional data very well.



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.



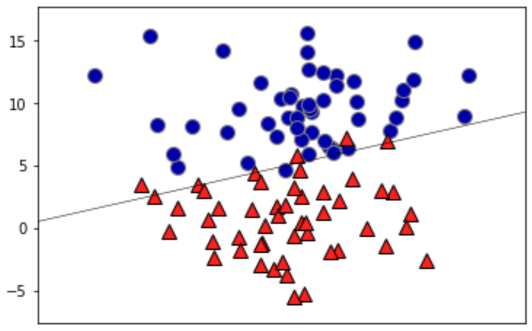
Uncertainty estimates from classifiers

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.

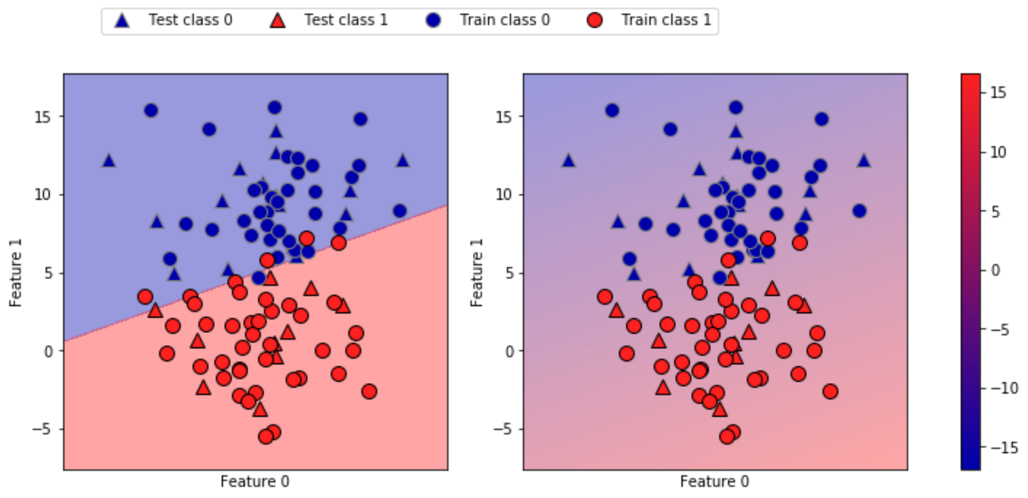
- Positive values indicate a preference for the "positive" class
- Negative values indicate a preference for the "negative" (other) class

```
Decision function:
[ 0.527  4.314  5.92   2.899  4.751 -7.035]


Thresholded decision function (>0):
[ True  True  True  True  True 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.

```
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.

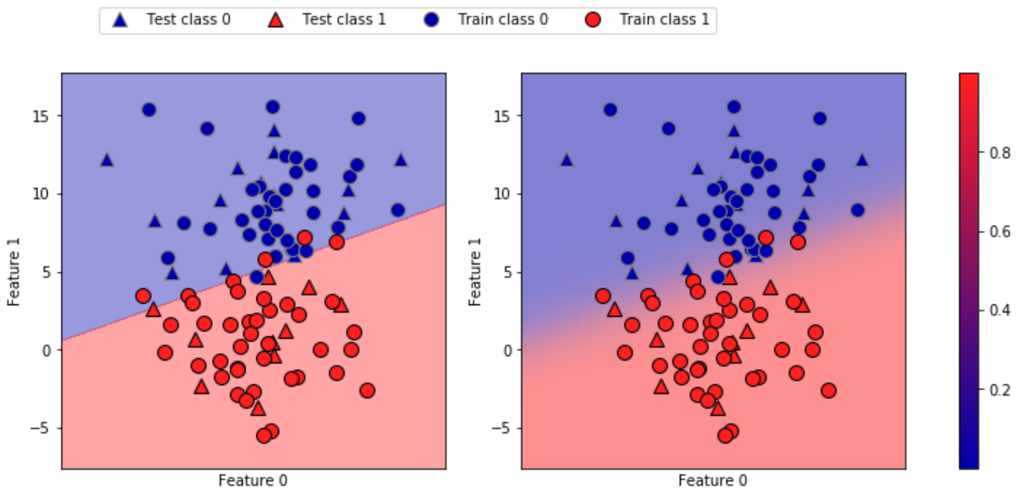


Predicting probabilities

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

```
Shape of probabilities: (25, 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]]
```

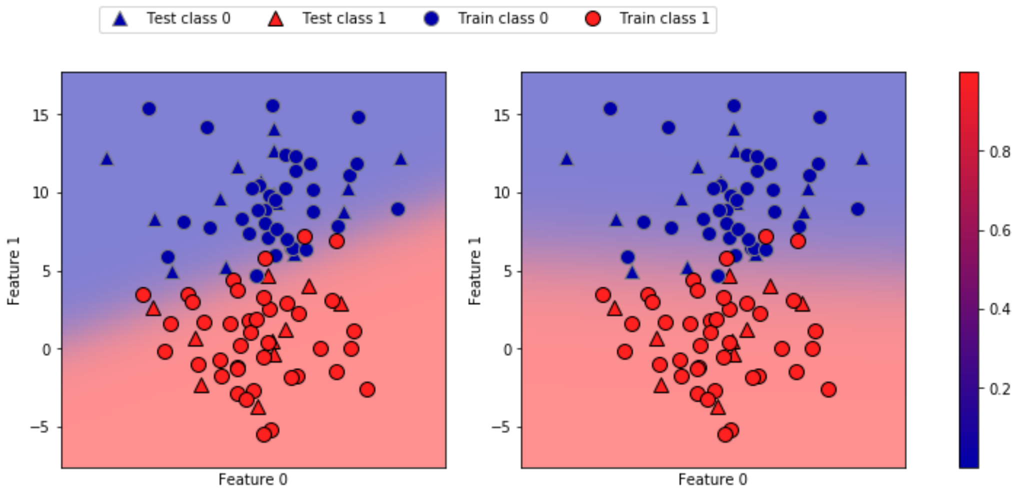
We can visualize them again. Note that the gradient looks different now.



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:
[[ -4.748   0.113  -1.084]
 [  3.683  -1.914 -10.976]
 [-10.126   0.901   4.262]
 [ -4.505  -0.498  -0.92 ]
 [ -4.887   0.265  -1.512]
 [  3.354  -1.622 -10.167]]
Predicted probabilities:
[[0.011 0.669 0.32 ]
 [0.884 0.116 0.   ]
 [0.    0.419 0.581]
 [0.016 0.561 0.423]
 [0.01  0.75  0.24 ]
 [0.854 0.146 0.   ]]
```

Summary

- Linear models
    - Go-to as a first algorithm to try, good for very large datasets, good for very high-dimensional data.
- Regularization is important. Choose between Ridge (L2) or Lasso (L1)
- Each algorithm has its own objective function, and each function can be optimized by certain techniques.
- Regression:
    - Ridge: L2 loss + least squares
    - Lasso: L1 loss + gradient descent
- Classification:
    - Logistic regression: Cross-entropy + gradient descent (or others)
    - SVM: Hinge loss + Langrange multipliers
- Classifiers return a certainty estimate. The actual prediction can be *calibrated* towards a specific goal.
- Multi-class classification can be done using a one-vs-all approach