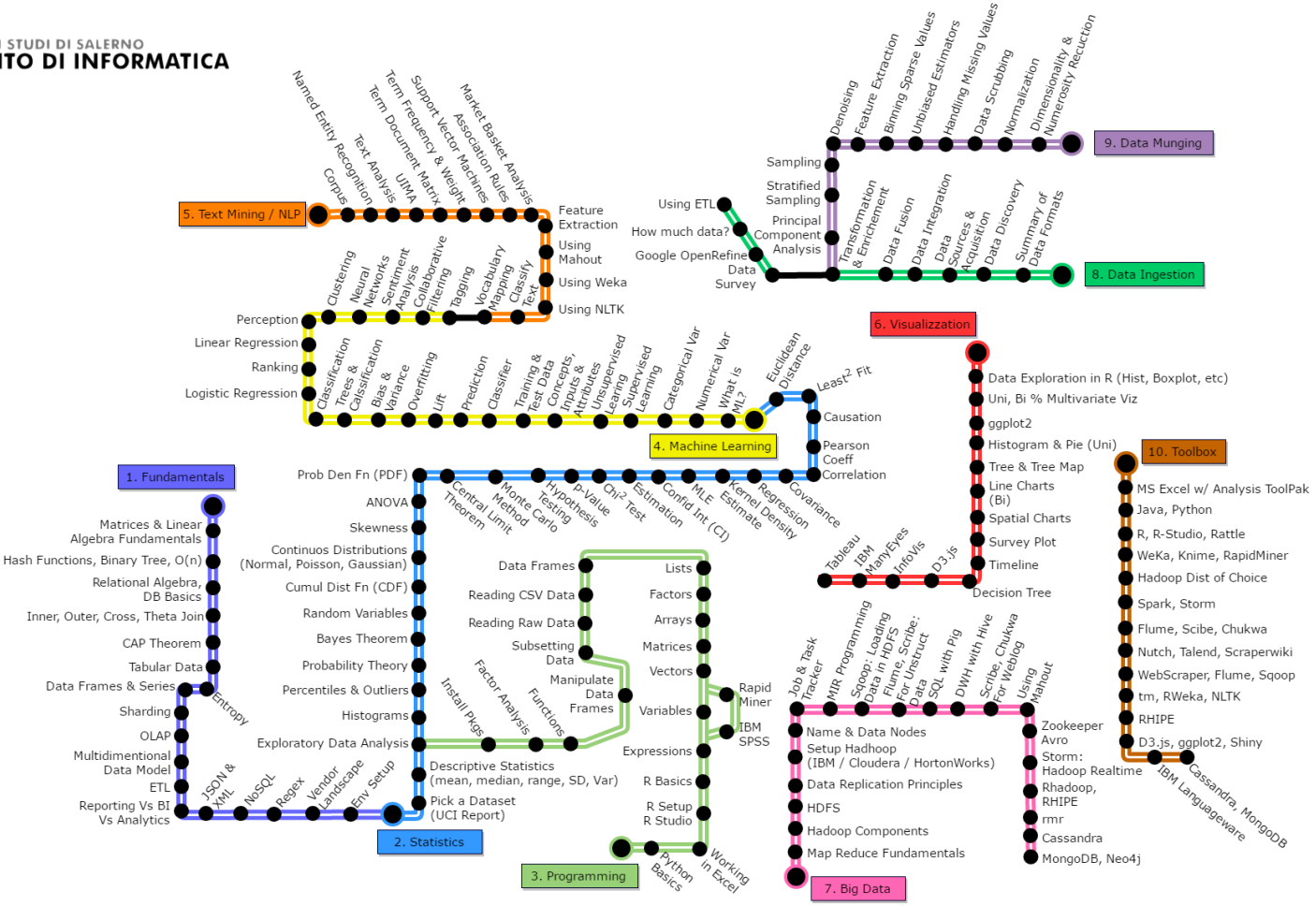
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



Fondamenti di Data Science e Machine Learning

Training Models (Chapter 4 Geron's Book)

Prof. Giuseppe Polese, aa 2024-25

Outline (1)

- Training Models
- Linear Regression
  - The Normal Equation
  - Computational Complexity
- Gradient Descent
- Polynomial Regression
  - Learning Curves
  - The Bias/Variance Tradeoff

Outline (2)

- Regularized Linear Models
  - Ridge Regression
  - Lasso Regression
  - Elastic Net
  - Early Stopping
- Logistic Regression
  - Estimating Probabilities
  - Training and Cost Function
  - Decision Boundaries
- Softmax Regression

Training Models (1)

- We will start by looking at the Linear Regression model
  - one of the simplest models
- We will discuss two very different ways to train it:
  - Using a direct "closed-form" equation that directly computes the model parameters that best fit the model to the training set
    - the model parameters that minimize the cost function over the training set
  - Using an iterative optimization approach, called Gradient Descent (GD), that gradually tweaks the model parameters to minimize the cost function over the training set

Training Models (2)

- We look at Polynomial Regression
  - A more complex model that can fit nonlinear datasets
- This model has more parameters than Linear Regression: it is more prone to overfitting the training data
  - We will look at how to detect whether or not this is the case, using learning curves;
  - We will look at several regularization techniques that can reduce the risk of overfitting the training set;
  - We will look at two more models that are commonly used for classification tasks: Logistic Regression and Softmax Regression

Linear Regression (1)

- A linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the bias term (also called the intercept term)

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

- $\hat{y}$ is the predicted value
- n is the number of features
- x_i is the ith feature value
- θ_j is the jth model parameter
  - including the bias term θ_0 and the feature weights $\theta_1, \ldots, \theta_n$

Linear Regression (2)

- Linear Regression model prediction using a vectorized form

$$\hat{y} = h_\theta(x) = \theta^T \cdot x$$

- θ is the model's parameter vector
  - including the bias term θ_0 and the feature weights θ_1 to θ_n
- θ^T is the transponse of θ
- x is the instance's feature vector, containing x_0 to x_n, with x_0 always equal to 1.
- $\theta^T \cdot x$ is the dot product of θ^T and x
- h_θ is the hypothesis function
  - using model parameters θ

The Normal Equation

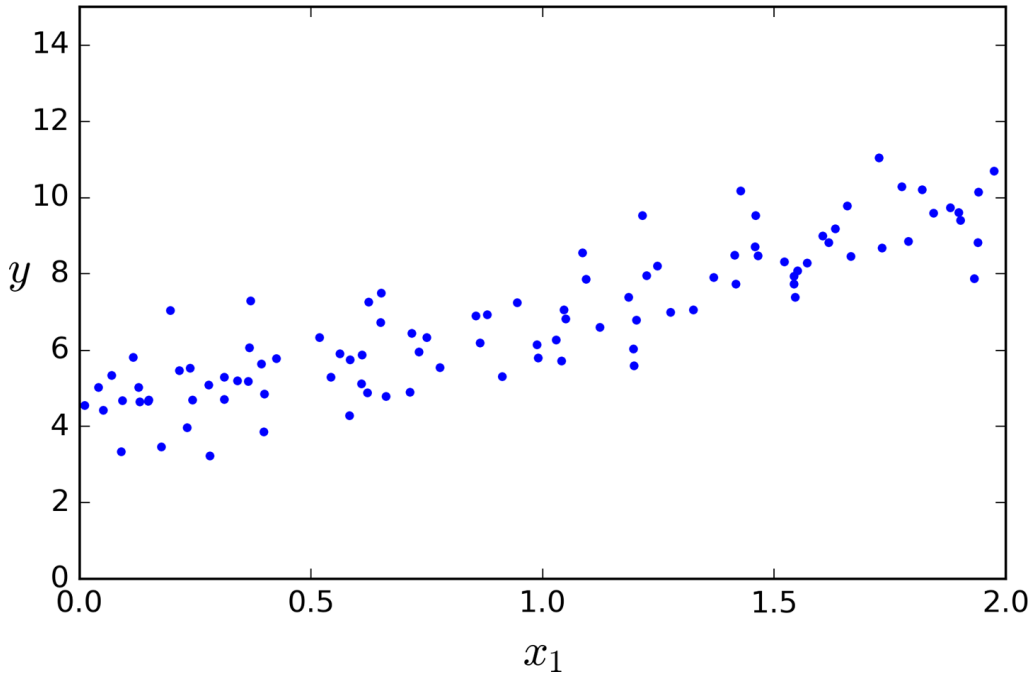
- To find the value of θ that minimizes the cost function, there is a closed-form solution
  - A mathematical equation that gives the result directly
  - This is called the Normal Equation

$$\hat{\theta} = (X^T \cdot X)^{-1} \cdot X^T \cdot y$$

- $\hat{\theta}$ is the value of θ that minimizes the cost function
- y is the vector of target values containing $y^{(1)}$ to $y^{(m)}$

Linear Regression: An example (1)

- Let's generate some linear-looking data to test this equation




```
import numpy as np
import matplotlib.pyplot as plt
#Generate values
m = 100
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
#Define Plot
plt.plot(X, y, "b.")
plt.xlabel("x_1")
plt.ylabel("y")
plt.show()
```

The actual function that we used to generate the data is:
$y = 4 + 3x_1 +$ Gaussian noise

Linear Regression: An example (2)

- Now let's compute $\hat{\theta}$ using the Normal Equation.
  - We will use the `inv()` function from NumPy's Linear Algebra module (`np.linalg`) to compute the inverse of a matrix, and
  - the `dot()` method for matrix multiplication
- While the function to generate the data is $y = 4 + 3x_1 +$ Gaussian noise, the equation yields:

```
…
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
print(theta_best)
```

```
array([[ 4.21509616],
       [ 2.77011339]])
```

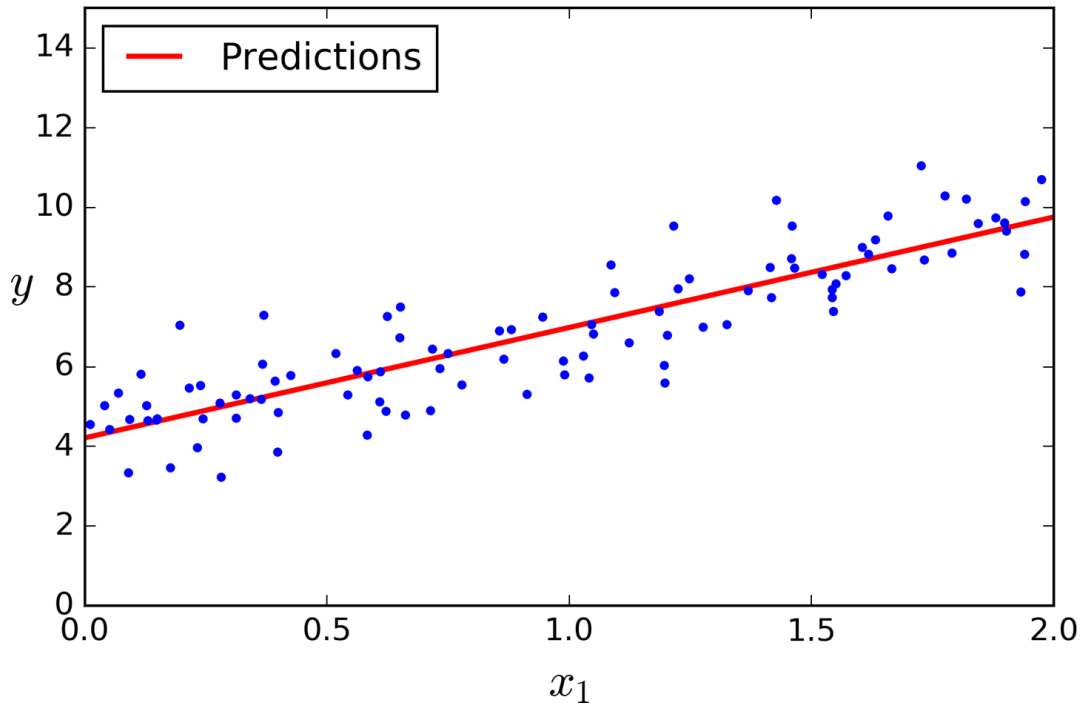
Linear Regression in scikit-learn (1)

- Results are close enough, but the noise made it impossible to recover the exact parameters of the original function
  - $\theta_0 = 4.215$ instead of $\theta_0 = 4$
  - $\theta_1 = 2.770$ instead of $\theta_1 = 3$
- Now you can make predictions using $\hat{\theta}$

```
import numpy as np
import matplotlib.pyplot as plt

#Generate values
m = 100
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
……
```

Linear Regression in scikit-learn (2)

```
…
#Apply regression
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
y_predict = X_new_b.dot(theta_best)
#Define Plot
plt.plot(X, y, "b.")
plt.plot(X_new, y_predict, "r-", linewidth=2, label="Predictions")
plt.xlabel("x_1")
plt.ylabel("y", rotation=0)
plt.legend(loc="upper left")
plt.show()
```



Computational Complexity (1)

- The Normal Equation computes the inverse of $X^T \cdot X$, which is an $n \times n$ matrix
  - where n is the number of features
- The computational complexity of inverting such a matrix ranges from $O(n^{2.4})$ to $O(n^3)$ (depending on the implementation).
  - If you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$
- The Normal Equation gets very slow when the number of features grows large (e.g., 100,000)

Computational Complexity (2)

- On the positive side, this equation is linear with regards to the number of instances in the training set (it is $O(m)$)
  - It handles large training sets efficiently, provided they can fit in memory
- Once you have trained your Linear Regression model (using the Normal Equation or any other algorithm), predictions are very fast
  - the computational complexity is linear wrt. both the number of instances you want to make predictions on and the number of features
- Making predictions on twice as many instances (or twice as many features) will just take roughly twice as much time

Training a Linear Regression Model

- There exist different ways to train a Linear Regression model
  - better suited for cases where there are a large number of features, or
  - better suited for cases where there are many training instances to fit in memory
- We discuss the Gradient Descendent

Gradient Descent (1)

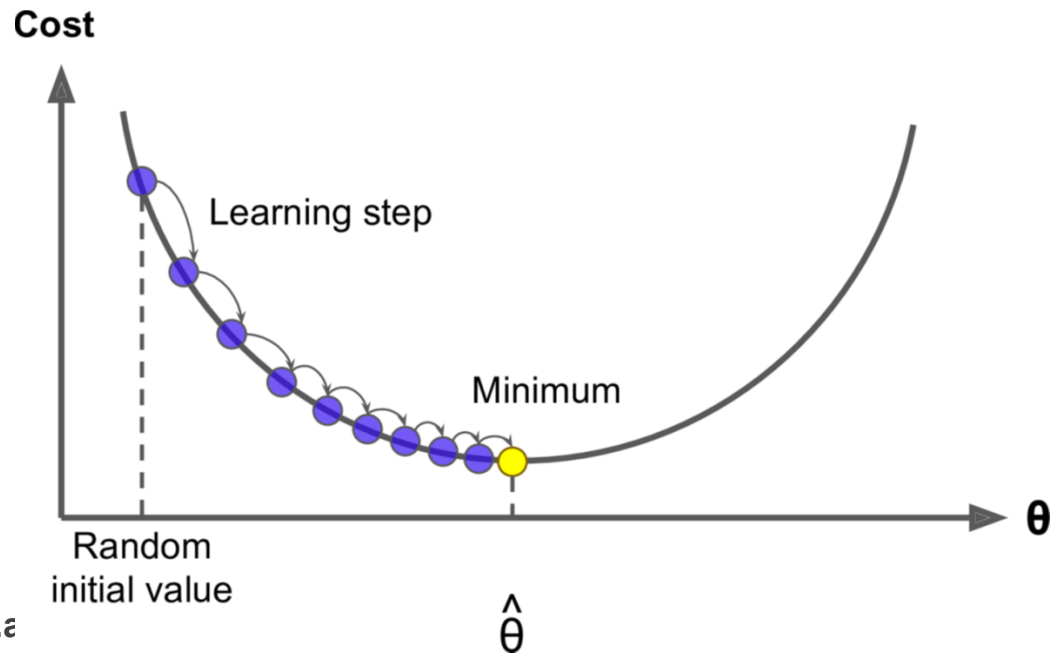
- Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems
- The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function

Gradient Descent (2)

- Suppose you are lost in the mountains in a dense fog, and you can only feel the slope of the ground below your feet
- A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope
- This is exactly what Gradient Descent does
  - It measures the local gradient of the error function with regards to the parameter vector θ, and
  - it goes in the direction of descending gradient
  - Once the gradient is zero, you have reached a minimum

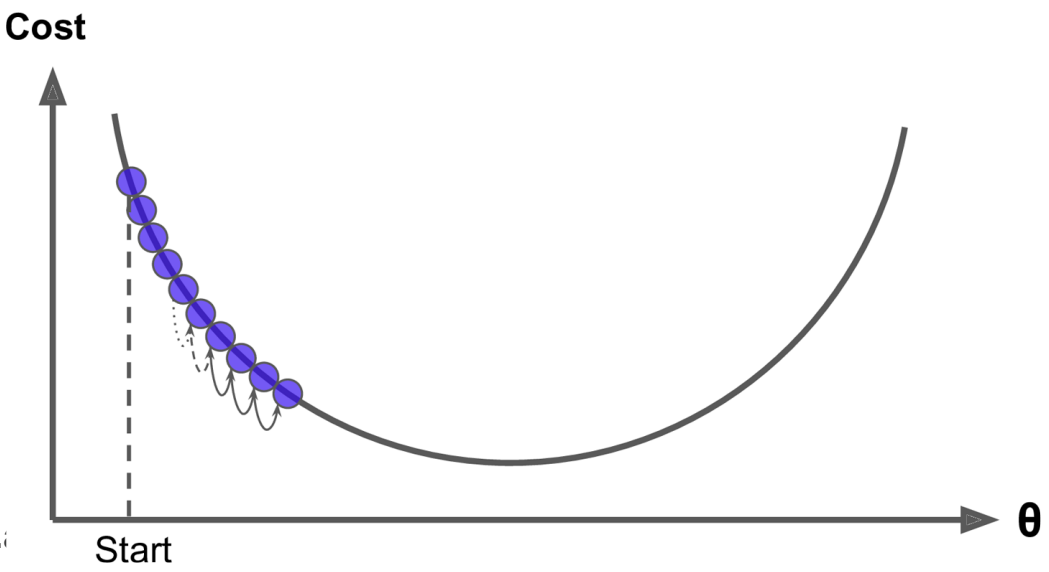
Gradient Descent (3)

- Concretely
  - you start by filling θ with random values, and then
    - this is called random initialization
  - you improve it gradually,
    - taking one baby step at a time,
    - each step attempting to decrease the cost function
      - e.g., the MSE
  - until the algorithm converges to a minimum



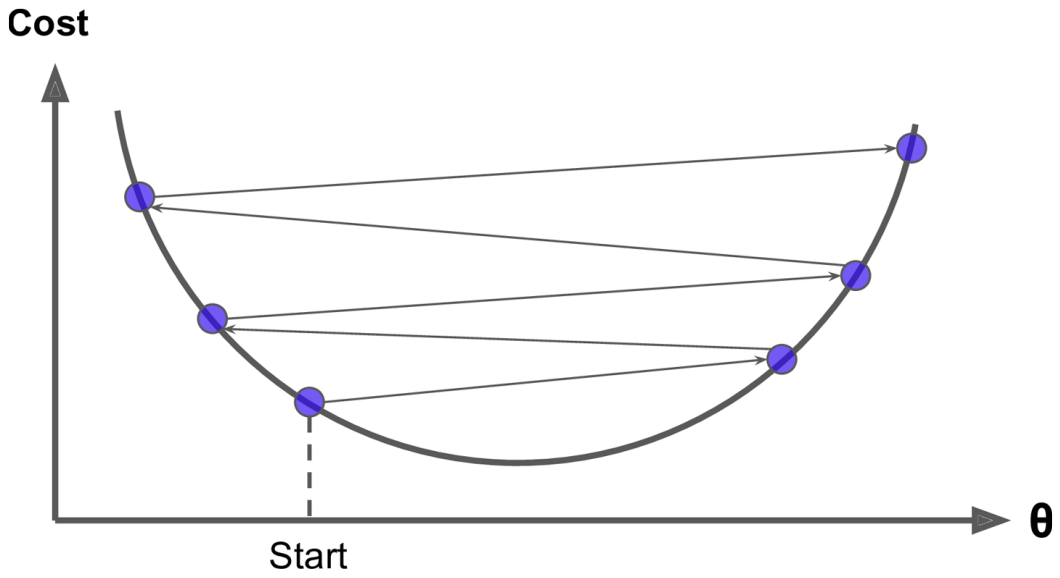
Gradient Descent (4)

- An important parameter in Gradient Descent is the size of the steps
  - determined by the *learning rate* hyperparameter
- If the learning rate is too small, then
- the algorithm will have to go through many iterations to converge
  - which will take a long time



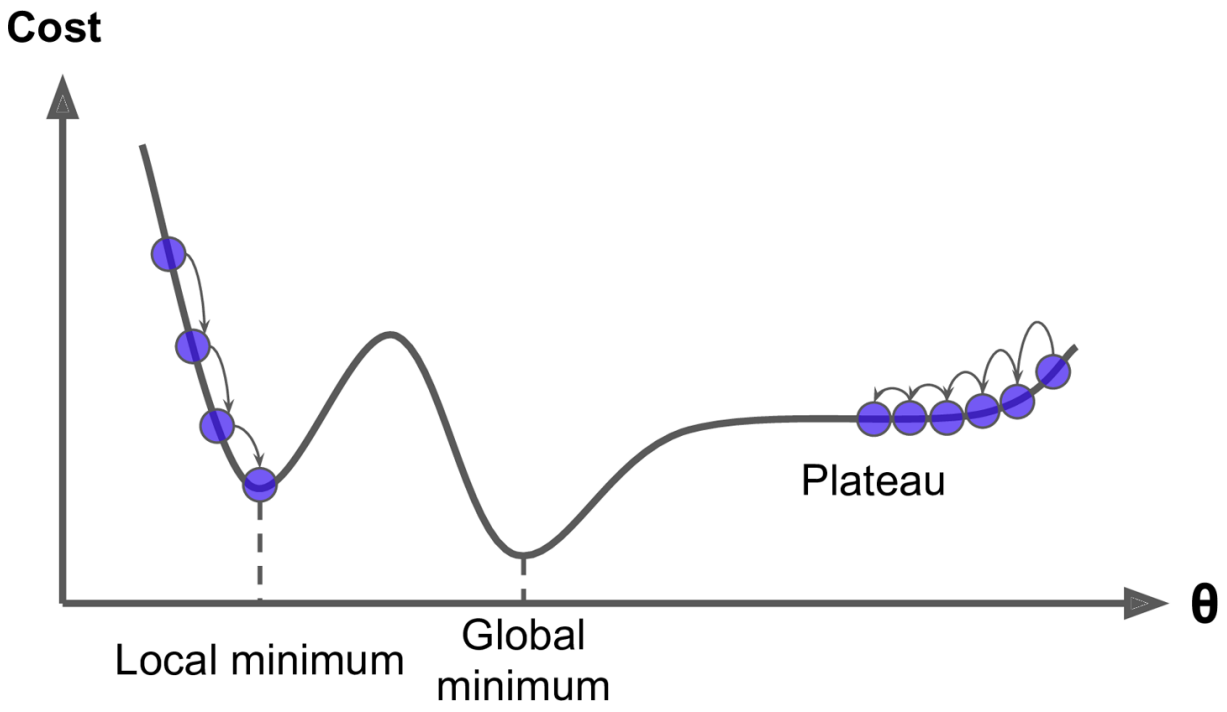
Gradient Descent (5)

- if the learning rate is too high
  - you might jump across the valley and end up on the other side, possibly even higher up than you were before
- This might make the algorithm diverge, with larger and larger values, failing to find a good solution



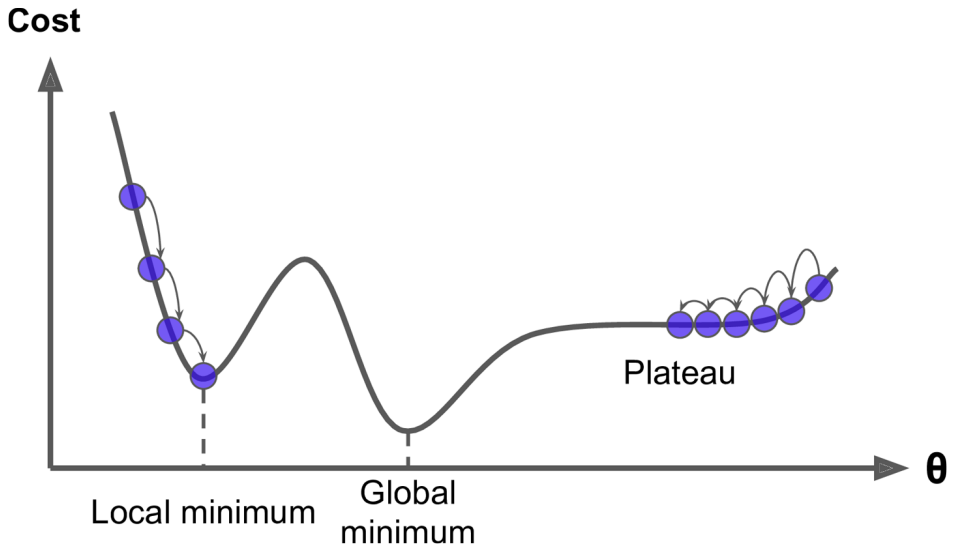
Gradient Descent (6)

- Not all cost functions look like nice regular bowls
  - There may be holes, ridges, plateaus, and irregular terrains, making convergence to the minimum very difficult



Gradient Descent (7)

- The figure shows the two main challenges with Gradient Descent



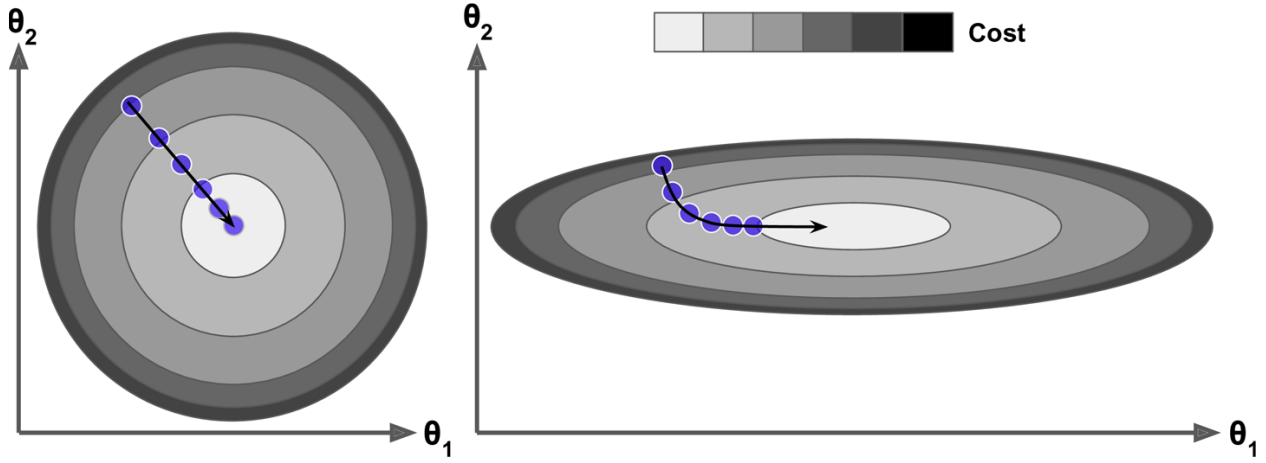
  - If the random initialization starts the algorithm on the left, then it will converge to a local minimum, which is not as good as the global minimum
  - If it starts on the right, then it will take a very long time to cross the plateau, and if you stop too early you will never reach the global minimum

Gradient Descent with MSE

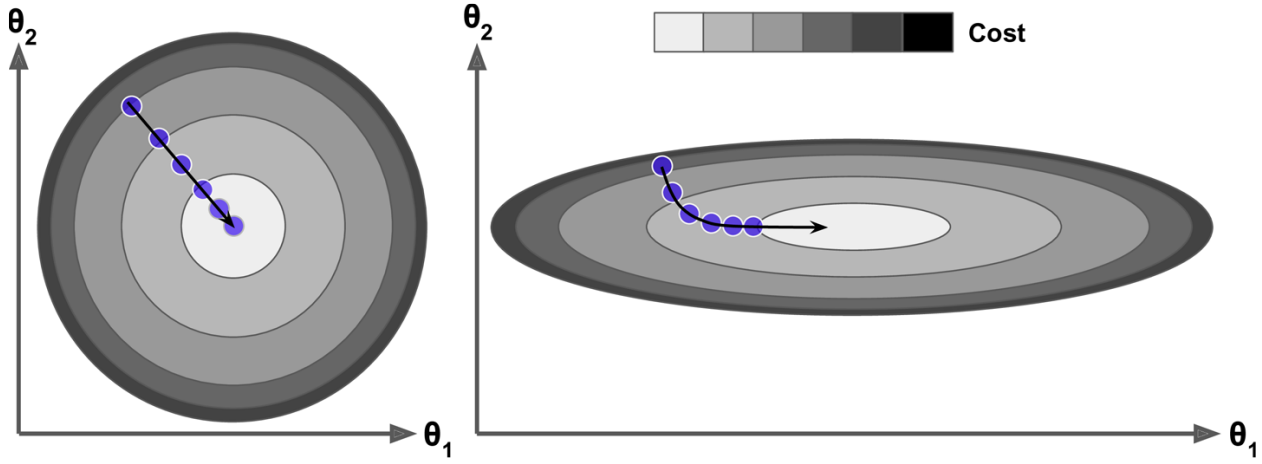
- The MSE cost function for a Linear Regression model happens to be a convex function
  - If you pick any two points on the curve, the line segment joining them never crosses the curve
- This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly
- Consequence
  - Gradient Descent is guaranteed to approach arbitrarily close the global minimum
    - if you wait long enough and if the learning rate is not too high

Feature scaling for Gradient Descent (1)

- The cost function has the shape of a bowl
  - it can be an elongated bowl if the features have different scales
- The figure shows Gradient Descent on a training set where
  - On the left: features 1 and 2 have the same scale, and
  - On the right: feature 1 has much smaller values than feature 2



Feature scaling for Gradient Descent (2)



- On the left the Gradient Descent algorithm goes straight toward the minimum
- On the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley
  - it will take a long time

Summarizing Gradient Descent

- Training a model means searching for a combination of model parameters that minimizes a cost function (over the training set)
- It is a search in the model's parameter space
  - the more parameters a model has,
  - the more dimensions this space has, and
  - the harder the search is
- The cost function is convex in the case of Linear Regression
  - the needle is simply at the bottom of the bowl

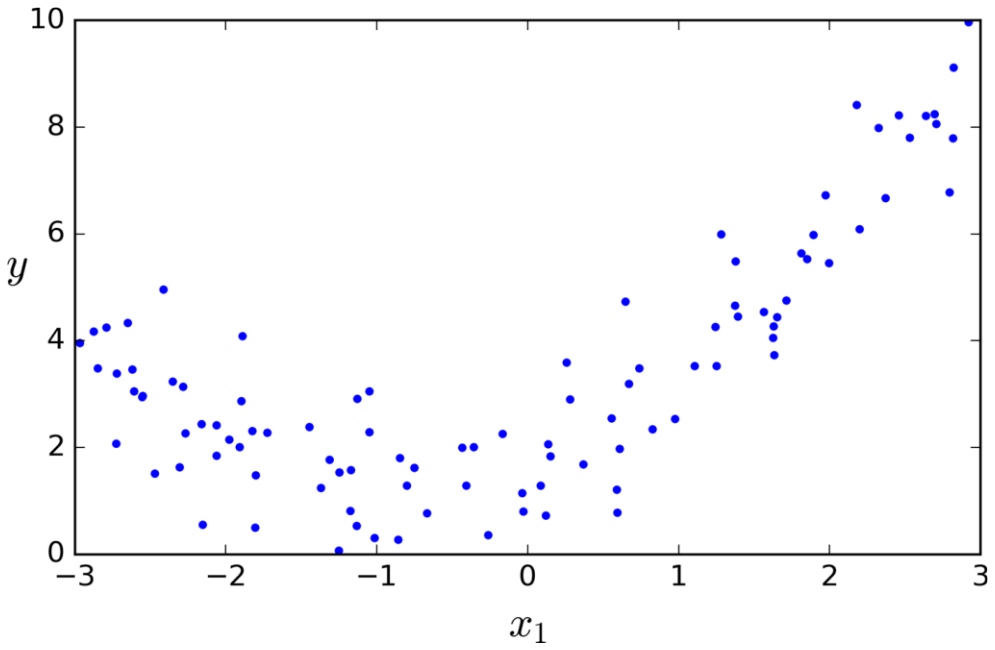
Polynomial Regression

What if your data is actually more complex than a simple straight line?

- Surprisingly, you can actually use a linear model to fit nonlinear data
  - A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features
- This technique is called Polynomial Regression

Polynomial Regression: An example

- Let's consider some nonlinear data, based on a simple *quadratic equation* (plus some noise)
  - $y = ax^2 + bx + c$




```
import numpy as np
import matplotlib.pyplot as plt
#Generate values
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.rand(m, 1)
#Define Plot
plt.plot(X, y, "b.")
plt.xlabel("x_1")
plt.ylabel("y")
plt.show()
```

Polynomial Regression in scikit-learn (1)

- A straight line will never fit this data properly
  - Scikit-Learn's `PolynomialFeatures` class to transform our training data, adding the square (2nd-degree polynomial) of each feature in the training set as new features

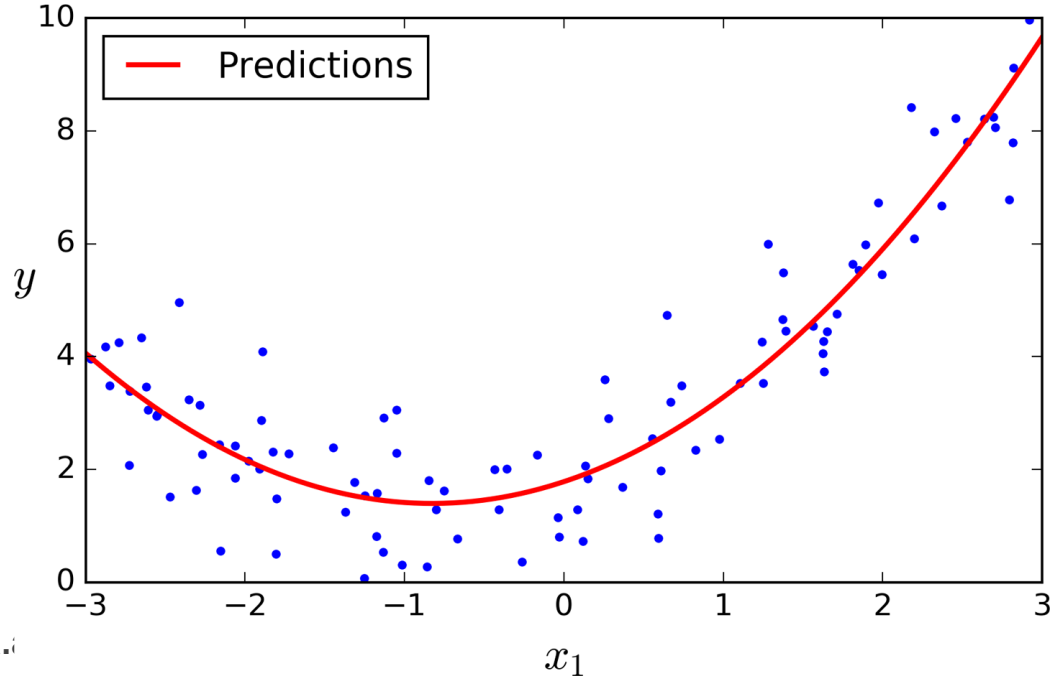
```
import numpy as np
import numpy.random as rnd
from sklearn.preprocessing import PolynomialFeatures

np.random.seed(42)
#Generate values
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.rand(m, 1)
#Add the square
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
print(X[0])
print(X_poly[0])
…
```

```
array([-0.75275929])
array([-0.75275929,  0.56664654])
```

Polynomial Regression in scikit-learn (2)

```
…
#Apply regression
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
print(lin_reg.intercept_, lin_reg.coef_)
#Define Plot
X_new=np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("x_1")
plt.ylabel("y", rotation=0)
plt.legend(loc="upper left")
plt.show()
```

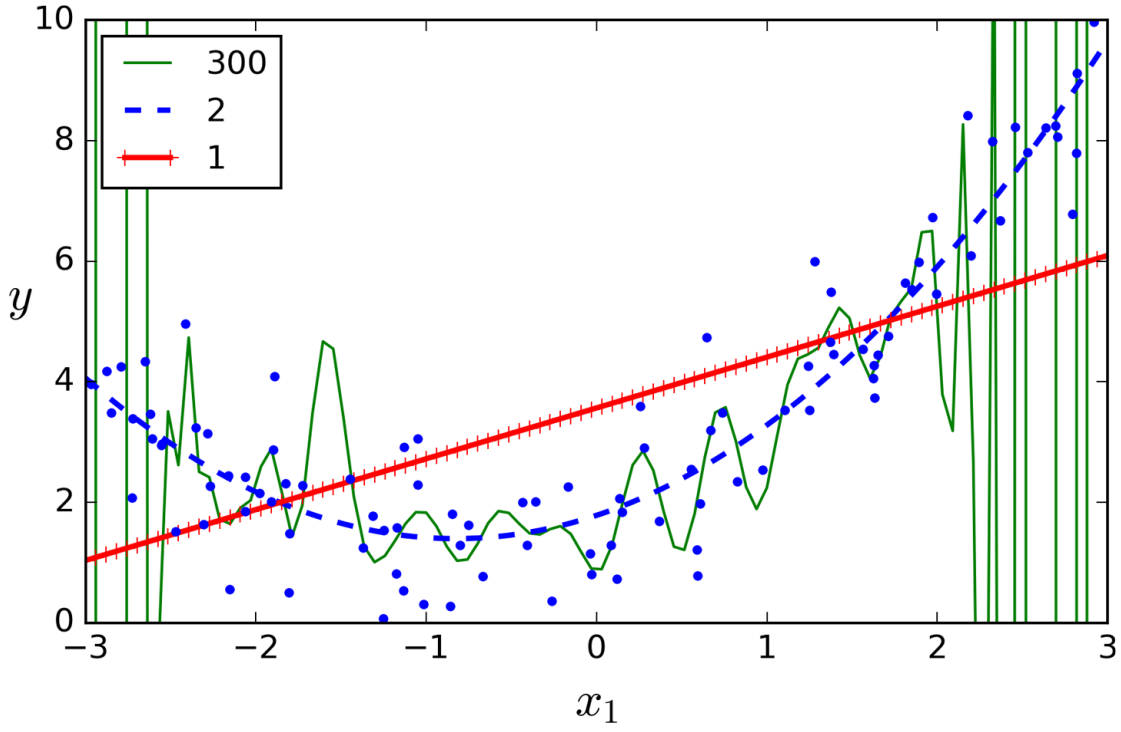
```
(array([ 1.78134581]), array([[ 0.93366893,  0.56456263]]))
```



Degrees of polynomial model (1)

- If you perform high-degree Polynomial Regression, you will likely fit training data much better than with Linear Regression

- The 300-degree polynomial model wiggles around to get as close as possible to the training instances



Degrees of polynomial model (2)

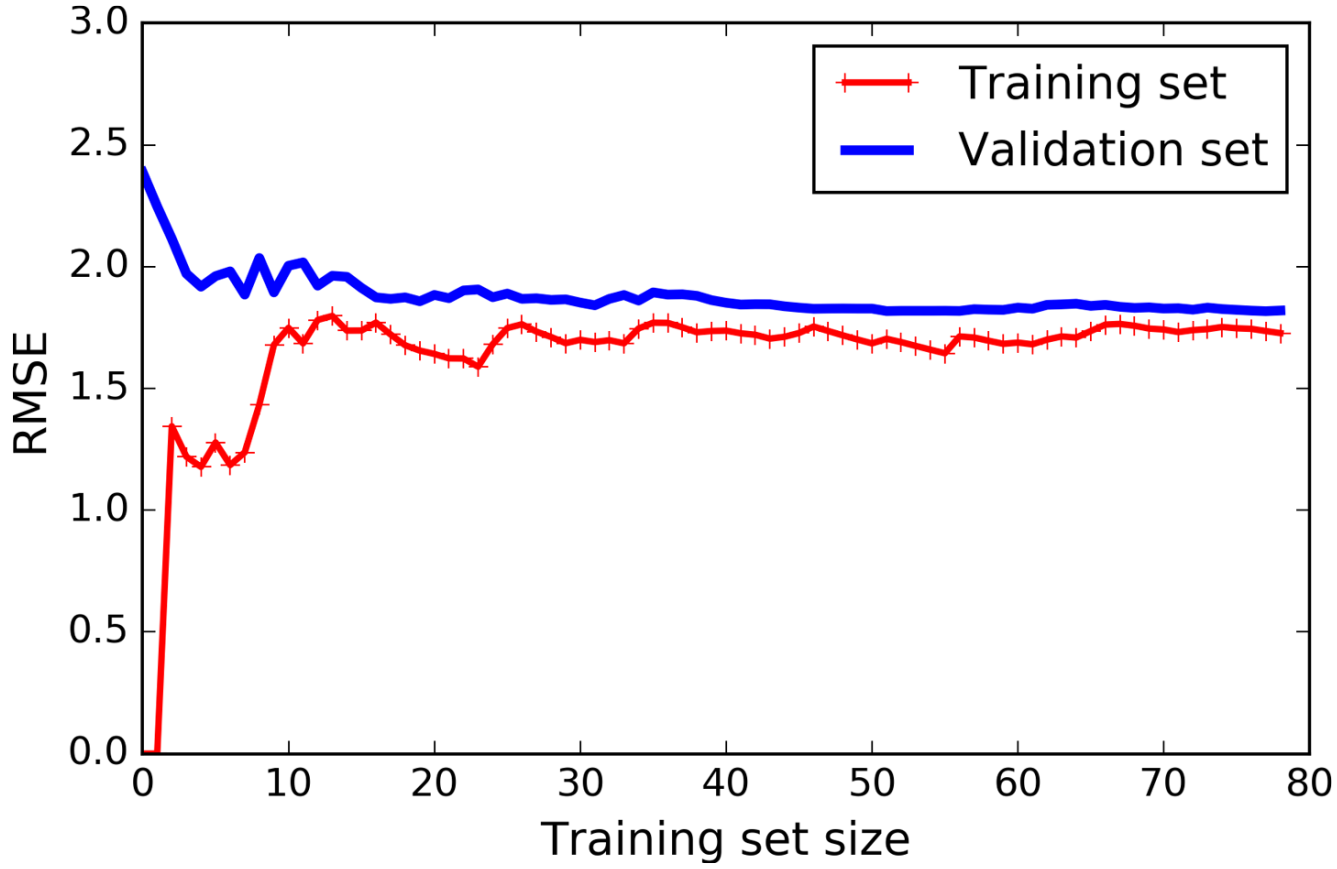
- The high-degree Polynomial Regression model is severely overfitting the training data
- The linear model is underfitting the training data
- The model that will generalize best (in this case) is the quadratic model
  - The data was generated using a quadratic model
- In general you won't know what function generated the data
  - How can you decide how complex your model should be?
  - How can you tell that your model is overfitting or underfitting the data?

Learning Curves (1)

- You previously used cross-validation to get and estimate of a model's generalization performance
- Another way is to look at the learning curves
  - Plots of the model's performance on the training set and the validation set as a function of the training set size (or the training iteration)
- To generate the plots, simply train the model several times on different sized subsets of the training set

Learning Curves (2)

- Let's look at the learning curves of the plain Linear Regression model



Performances on the training data

- In the figure, when there are just one or two instances in the training set, the model can fit them perfectly
  - which is why the curve starts at zero
- As new instances are added to the training set, it is impossible for the model to fit the training data perfectly
  - the data is noisy, and
  - it is not linear at all
- The error goes up until it reaches a plateau
  - point in which adding new instances to the training set doesn't make the average error much better or worse

Performances on the validation data

- In the figure, when the model is trained on very few training instances, it is incapable of generalizing properly
  - which is why the validation error is initially quite big
- As the model is shown more training examples, it learns and thus the validation error slowly goes down
- However, a straight line cannot do a good job modeling the data
  - the error ends up at a plateau, very close to the other curve

Summarizing Learning Curves

- The discussed learning curves are typical of an underfitting model
  - Both curves have reached a plateau
  - they are close and fairly high
- If your model is underfitting the training data, adding more training examples will not help
- You need to use
  - a more complex model, or
  - come up with better features

The Bias/Variance Tradeoff (1)

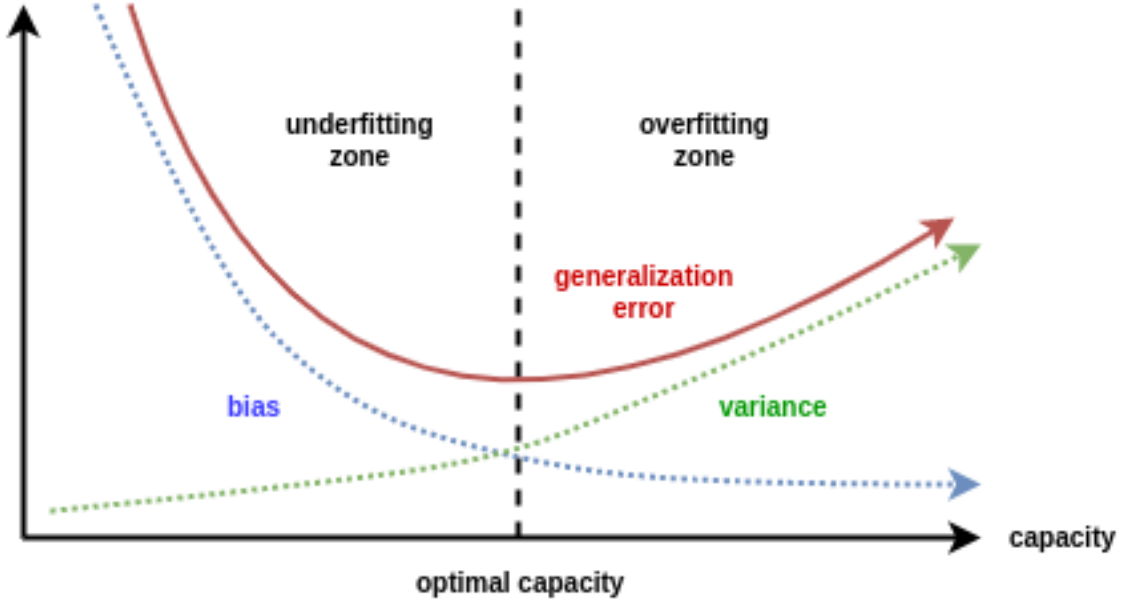
- An important theoretical result of statistics and Machine Learning is the fact that a model's generalization error can be expressed as the sum of three very different errors
- Bias
  - This part of the generalization error is due to wrong assumptions
    - such as assuming that the data is linear when it is actually quadratic
  - A high-bias model is most likely to underfit the training data

The Bias/Variance Tradeoff (2)

- Variance
  - This part is due to the model's excessive sensitivity to small variations in the training data
  - A model with many degrees of freedom is likely to have high variance, and to overfit the training data
    - such as a high-degree polynomial model
- Irreducible error
  - This part is due to the noisiness of the data itself
  - The only way to reduce this part of the error is to clean up the data
    - e.g., fix data sources: broken sensors or detect/remove outliers

The Bias/Variance Tradeoff (3)

- Increasing a model's complexity will typically increase its variance and reduce its bias
- Reducing a model's complexity increases its bias and reduces its variance



Regularized Linear Models

- A good way to reduce overfitting is to regularize the model (i.e., to constrain it)
  - the fewer degrees of freedom it has, the harder it will be for it to overfit the data
- For example, a simple way to regularize a polynomial model is to reduce the number of polynomial degrees
- For a linear model, regularization is typically achieved by constraining the weights of the model

Ridge Regression (1)

- Ridge Regression (also called Tikhonov regularization) is a regularized version of Linear Regression
  - A regularization term equal to $\alpha \sum_{i=1}^{n} \theta_i^2$ is added to the cost function
- This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible
- The regularization term should only be added to the cost function during training
- To evaluate the model's performance you should use the unregularized performance measure

Ridge Regression (2)

- It is quite common for the cost function used during training to be different from the performance measure used for testing
  - A good training cost function should have optimization-friendly derivatives
  - The performance measure used for testing should be as close as possible to the final objective
- Example: A classifier
  - trained using a cost function such as the log loss (discussed in a moment), but
  - evaluated using precision/ recall

The Ridge Regression Cost Function (1)

- The following equation presents the Ridge Regression cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

- The hyperparameter α controls how much you want to regularize the model
  - If $\alpha = 0$ then Ridge Regression is just Linear Regression
  - If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean

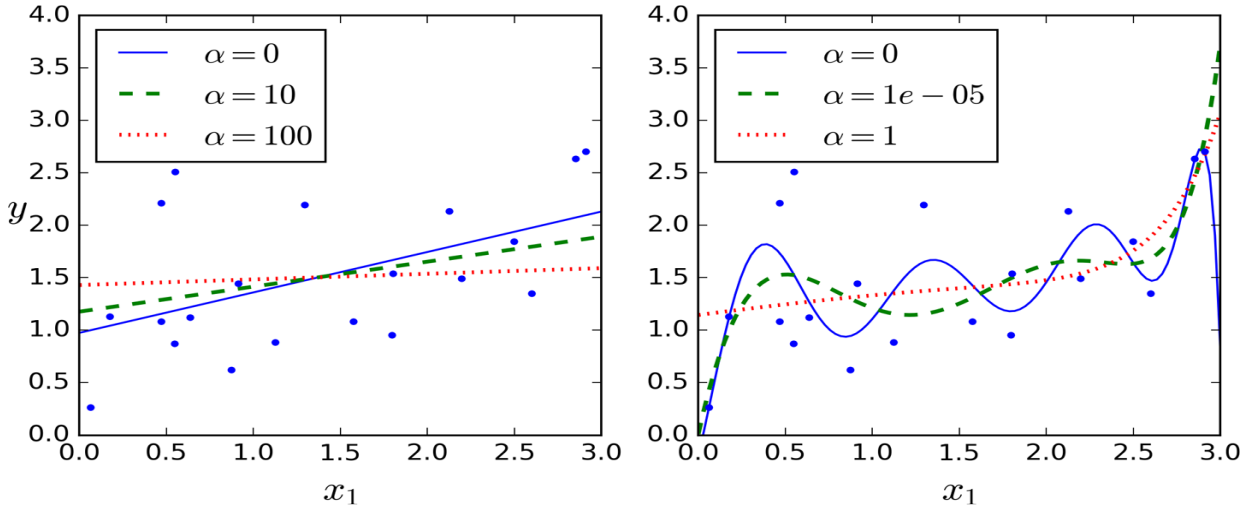
The Ridge Regression Cost Function (2)

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

- Note that the bias term θ_0 is not regularized
  - the sum starts at $i = 1$ not 0
- It is important to scale the data before performing Ridge Regression, as it is sensitive to the scale of the input features
  - e.g., using a StandardScaler
- This is true of most regularized models

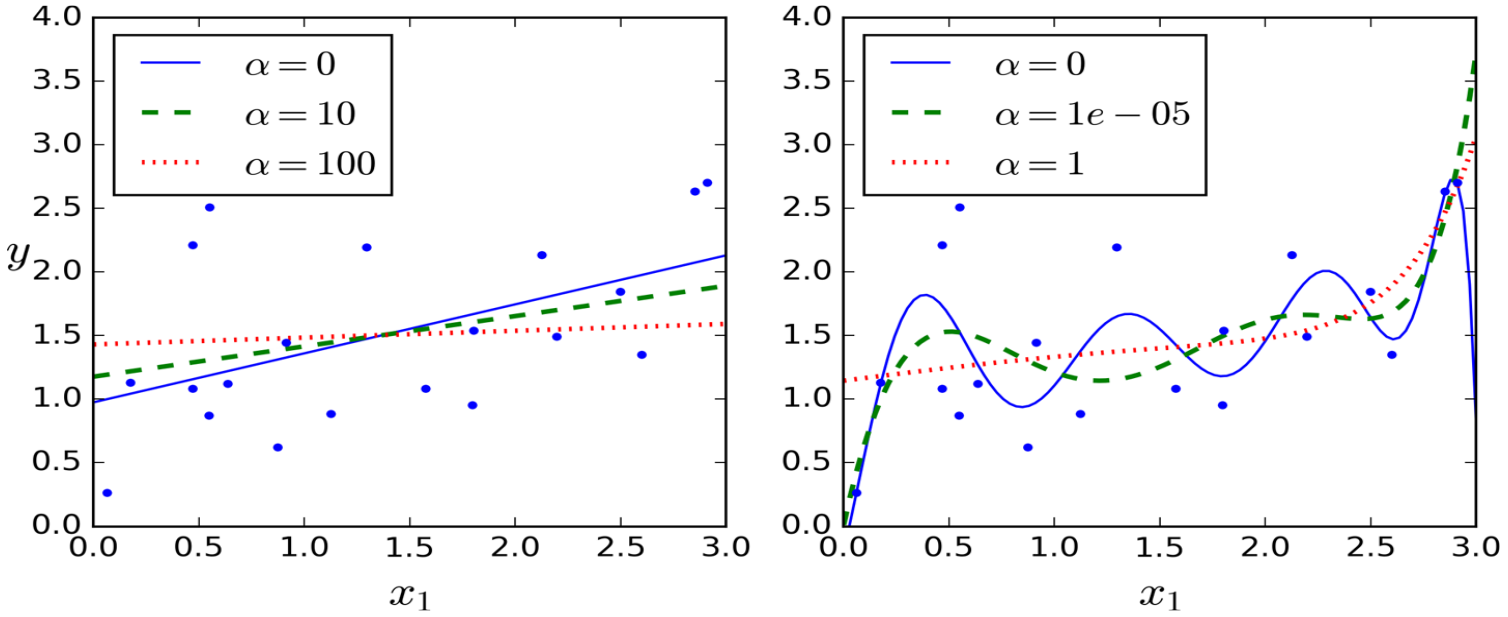
Example of Several Ridge Models (1)

- Several Ridge models trained on linear data using different α value



- On the left, plain Ridge models are used leading to linear predictions
- On the right
  - the data is expanded using PolynomialFeatures(degree=10)
  - the data is scaled using a StandardScaler, and
  - the Ridge models are applied to the resulting features
    - this is Polynomial Regression with Ridge regularization

Example of Several Ridge Models (2)



- Note how increasing α leads to flatter
  - i.e., less extreme, more reasonable predictions
- This reduces the model's variance but increases its bias

Ridge Regression Closed-form Solution

- We can perform Ridge Regression either by computing a closed-form equation or by performing Gradient Descent
  - As with Linear Regression
- The following equation shows the closed-form solution

$$\hat{\theta} = (X^T \cdot X + \alpha A)^{-1} \cdot X^T \cdot y$$

  - where A is the $n \times n$ identity matrix except with a 0 in the top-left cell, corresponding to the bias term)

Lasso Regression

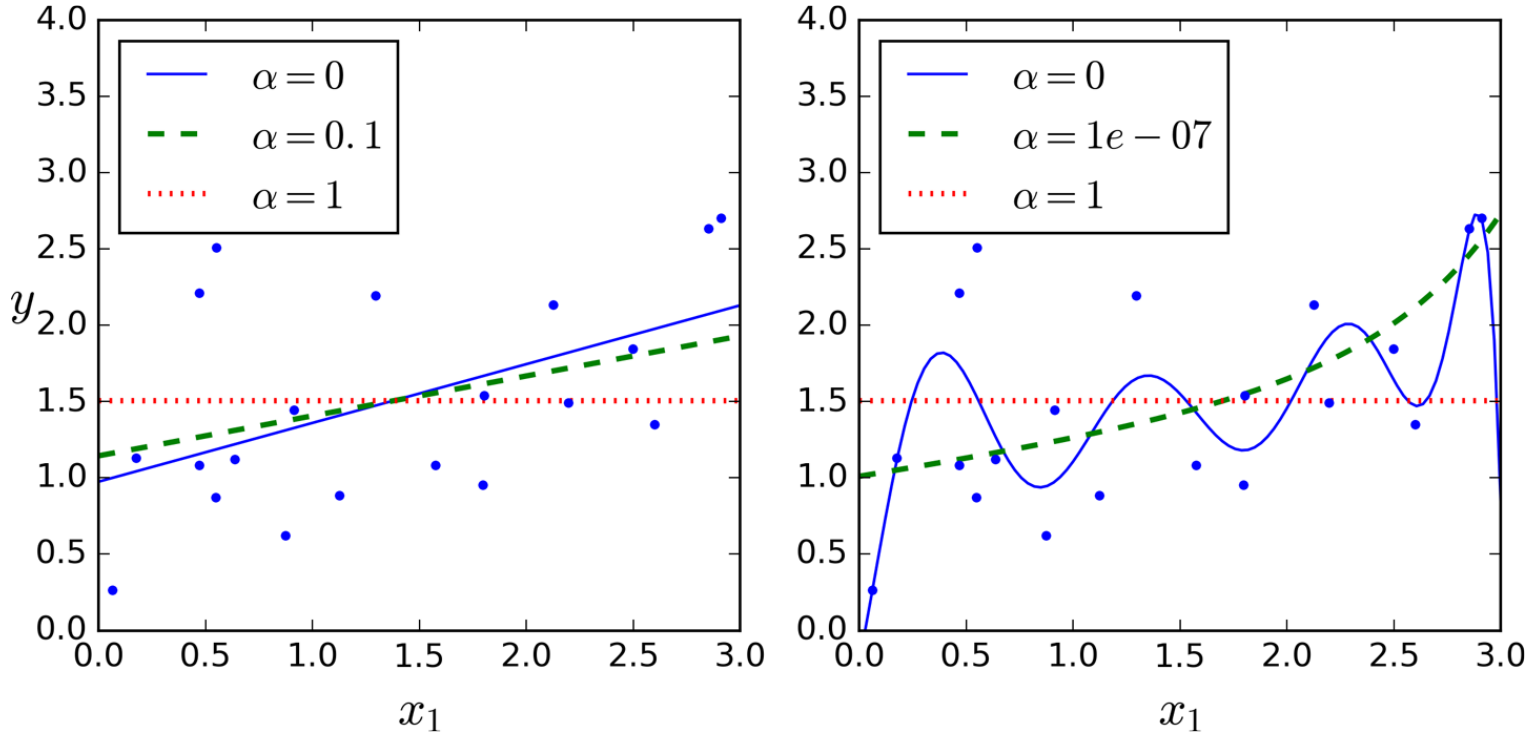
- Least Absolute Shrinkage and Selection Operator Regression (simply called Lasso Regression) is another regularized version of Linear Regression
  - It adds a regularization term to the cost function, but it uses the ℓ_1 norm of the weight vector instead of half the square of the ℓ_2 norm

$$J(\theta) = \mathrm{MSE}(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

- Lasso Regression tends to completely eliminate the weights of the least important features
  - i.e., set them to zero

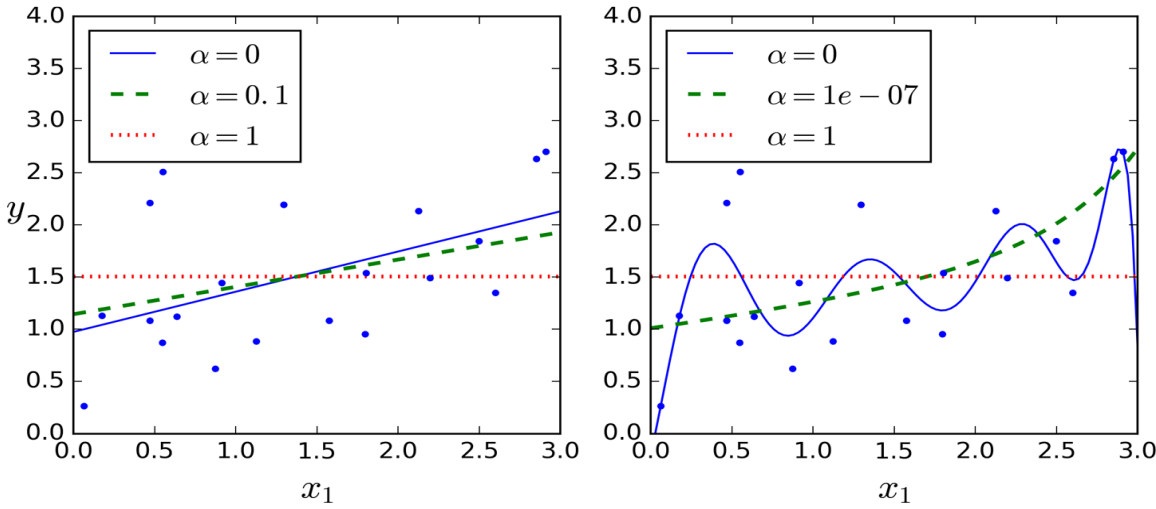
Example of Several Lasso Models (1)

- The following figure shows the Lasso on the previous example, by using smaller α values



Example of Several Lasso Models (2)

- The dashed line on the right plot with $\alpha = 10^{-7}$ looks quadratic, almost linear



  - all the weights for the high-degree polynomial features are equal to zero

- Lasso Regression automatically performs feature selection and outputs a sparse model
  - i.e., with few nonzero feature weights

Elastic Net

- Elastic Net is a middle ground between Ridge Regression and Lasso Regression
  - The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and
  - you can control the mix ratio r

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^{n} \theta_i^2$$

  - when $r = 0$, Elastic Net is equivalent to Ridge Regression, and
  - when $r = 1$, it is equivalent to Lasso Regression (see Equation 4-12)

Using Regularization

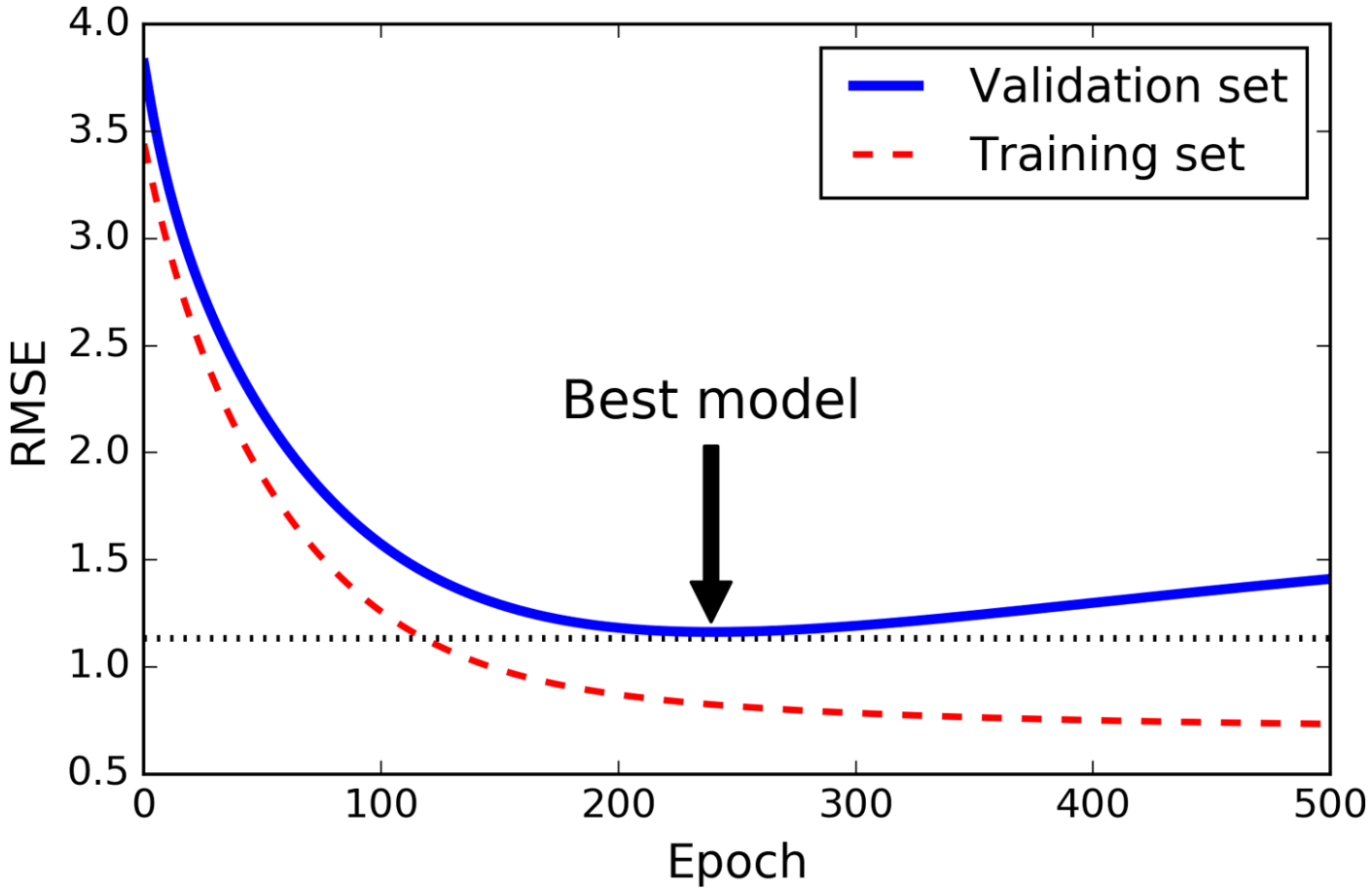
When should you use plain Linear Regression (i.e., without any regularization), Ridge, Lasso, or Elastic Net?

- Generally you should avoid plain Linear Regression
- Ridge is a good default
- Lasso or Elastic Net if you suspect that only a few features are actually useful
  - you should prefer since they tend to reduce the useless features' weights down to zero as we have discussed
- In general, Elastic Net is preferred over Lasso
  - Lasso may behave erratically when
    - the number of features is greater than the number of training instances, or
    - when several features are strongly correlated

Early Stopping (1)

- A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum
  - This is called early stopping
- The figure (next slide) shows a complex model
  - In this case a high-degree Polynomial Regression model trained using Batch Gradient Descent
- As the epochs go by
  - the algorithm learns and its prediction error (RMSE) on the training set naturally goes down, and so does its prediction error on the validation set

Early Stopping (2)



Early Stopping (3)

- After a while the validation error stops decreasing and actually starts to go back up
- This indicates that the model has started to overfit the training data
- With early stopping you just stop training as soon as the validation error reaches the minimum
- It is such a simple and efficient regularization technique that Geoffrey Hinton called it a “beautiful free lunch”

Logistic Regression

- Some regression algorithms can be used for classification as well and vice versa
- Logistic Regression is commonly used to estimate the probability that an instance belongs to a particular class
  - e.g., what is the probability that this email is spam?
- If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class
  - it belongs to the positive class – labeled “1”
- else it predicts that it does not
  - it belongs to the negative class – labeled “0”
- This makes it a binary classifier

Estimating Probabilities (1)

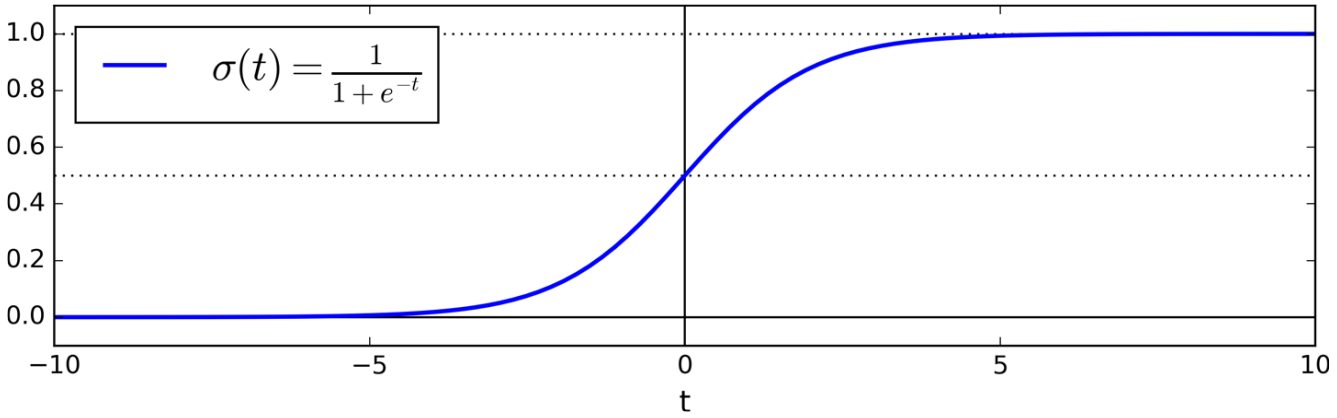
- Logistic Regression model computes a weighted sum of the input features plus a bias term
- The model outputs the logistics of this result
  - Instead of outputting the result directly
    - Like the Linear Regression model does

$$\hat{p} = h_\theta(\mathrm{x}) = \sigma(\theta^T \cdot \mathrm{x})$$

Estimating Probabilities (2)

- The logistic—also called the logit, noted $\sigma(\cdot)$— is a sigmoid function that outputs a number between 0 and 1
- It is defined as shown in the following

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$



Estimating Probabilities (3)

- Once the Logistic Regression model has estimated the probability $\hat{p} = h_\theta(\mathrm{x})$ that an instance x belongs to the positive class, it can easily make its prediction $\hat{y}$

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

- Notice that
  - $\sigma(t) < 0.5$ when $t < 0$, and
  - $\sigma(t) \geq 0.5$ when $t \geq 0$
- A logistic Regression model predicts 1 if $\theta^T \cdot \mathrm{x}$ is not negative, and 0 if it is negative

Training a logistic regression model

- The objective of training is to set the parameter vector θ so that the model can make estimations
  - high probabilities for positive instances $(y = 1)$, and
  - low probabilities for negative instances $(y = 0)$
- This idea is captured by the following cost function for a single training instance x

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

Training and Cost Function

- Such a cost function makes sense
  - – $\log(t)$ grows very large when t is close to 0
    - the cost will be large if the model estimates a probability close to 0 for a positive instance
    - it will also be very large if the model estimates a probability close to 1 for a negative instance
- On the other hand
  - – $\log(t)$ is close to 0 when t is close to 1
    - the cost will be close to 0 if the estimated probability is close to 0 for a negative instance, or
    - the cost will be close to 1 for a positive instance, which is precisely what we want

Logistic Regression Cost Function

- The cost function over the whole training set is simply the average cost over all training instances
- It can be written in a formula single called the log loss

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[\, y^{(i)}\log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - \hat{p}^{(i)}\right)\right]$$

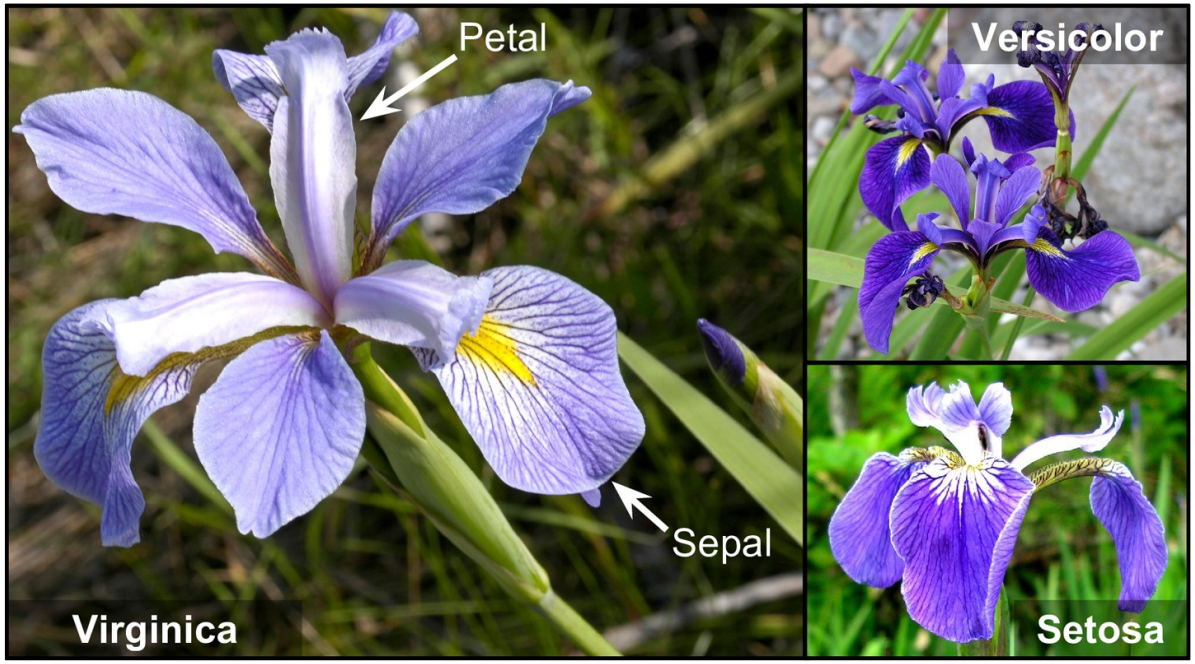
Training and Cost Function

- There is no known closed-form equation to compute the value of θ that minimizes this cost function
  - there is no equivalent of the Normal Equation
- But this cost function is convex
  - Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum
    - if the learning rate is not too large and you wait long enough
- The partial derivatives of the cost function with regards to the j[th] model parameter θ_j is given by

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\sigma\left(\theta^T \cdot \mathrm{x}^{(i)}\right) - y^{(i)} \right) x_j^{(i)}$$

An example of Logistic Regression (1)

- Let's use the iris dataset to illustrate Logistic Regression
  - The dataset contains the sepal and petal length and width of 150 iris flowers of three different species
    - Iris-Setosa
    - Iris-Versicolor
    - Iris-Virginica



An example of Logistic Regression (2)

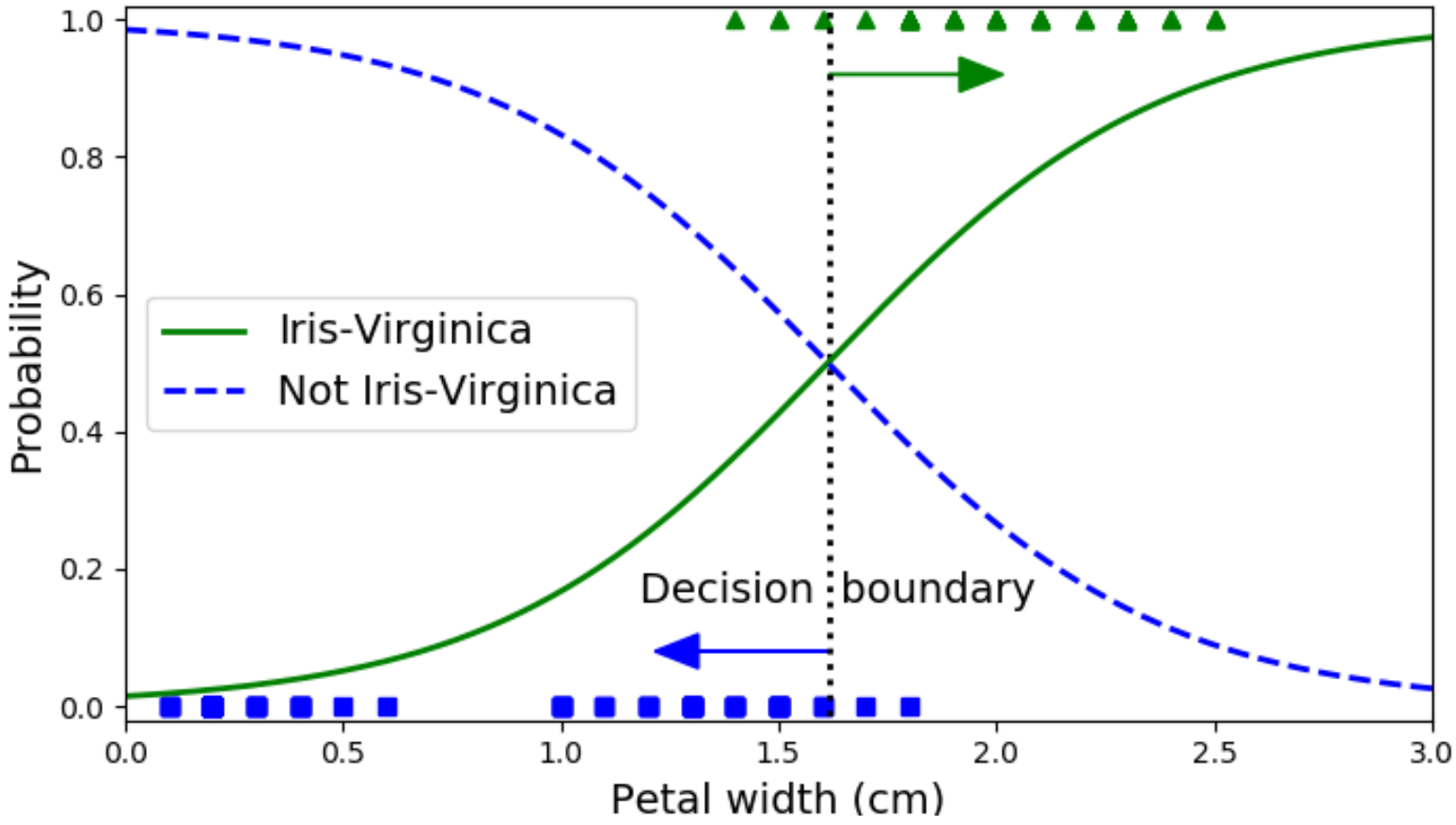
- Let's try to build a classifier to detect the Iris-Virginica type based only on the petal width feature
- After this train a Logistic Regression model
- Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 to 3 cm
- The code in the next slide implements Logistic Regression model on Iris-Virginica dataset

Logistic Regression in scikit-learn (1)

```
from sklearn import datasets
import numpy as np
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt
#Load iris dataset
iris = datasets.load_iris()
# petal width
X = iris["data"][:, 3:]
# 1 if Iris-Virginica, else 0
y = (iris["target"] == 2).astype(np.int)
#LogisticRegression method
log_reg = LogisticRegression(solver="liblinear", random_state=42)
log_reg.fit(X, y)
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0]
# Define Plot of decision boundary
plt.figure(figsize=(8, 3))
plt.plot(X[y==0], y[y==0], "bs")
plt.plot(X[y==1], y[y==1], "g^")
plt.plot([decision_boundary, decision_boundary], [-1, 2], "k:", linewidth=2)
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2, label="Not Iris-Virginica")
plt.text(decision_boundary+0.02, 0.15, "Decision  boundary", fontsize=14, color="k", ha="center")
plt.arrow(decision_boundary, 0.08, -0.3, 0, head_width=0.05, head_length=0.1, fc='b', ec='b')
plt.arrow(decision_boundary, 0.92, 0.3, 0, head_width=0.05, head_length=0.1, fc='g', ec='g')
plt.xlabel("Petal width (cm)", fontsize=14)
plt.ylabel("Probability", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 3, -0.02, 1.02])
plt.show()
```

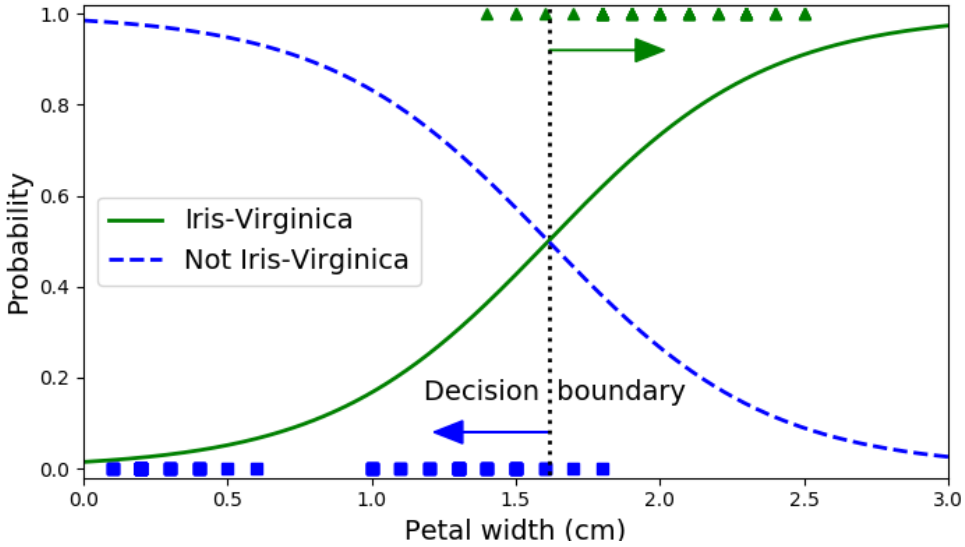
Logistic Regression in scikit-learn (2)

Output:



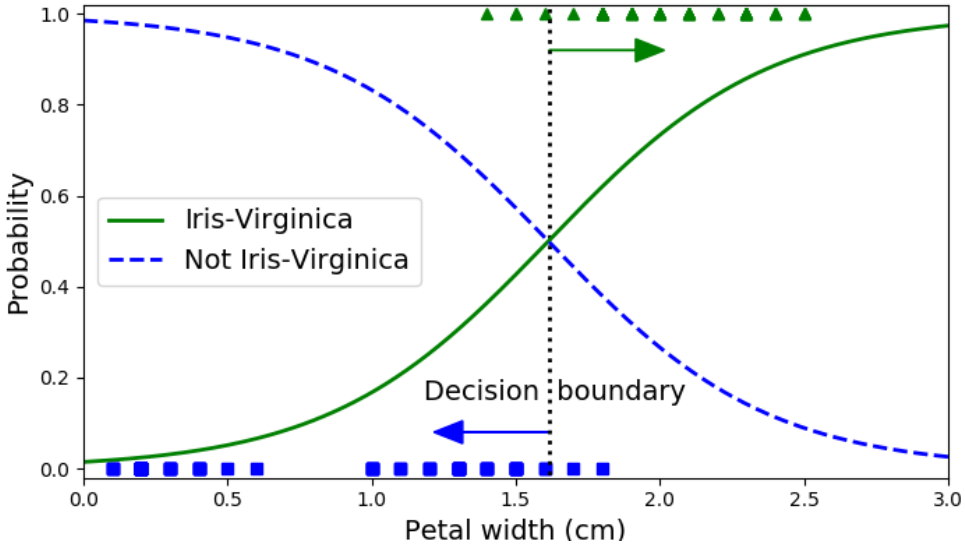
Decision Boundaries (1)

- The petal width of Iris-Virginica flowers ranges from 1.4 cm to 2.5 cm
  - represented by triangles
- The other iris flowers generally have a smaller petal width ranging from 0.1 cm to 1.8 cm
  - represented by squares



Decision Boundaries (2)

- Notice that there is a bit of overlap
  - Above about 2 cm the classifier is highly confident that the flower is an Iris- Virginica
    - it outputs a high probability to that class
  - Below 1 cm it is highly confident that it is not an Iris-Virginica
    - high probability for the "Not Iris-Virginica" class
- In between these extremes, the classifier is unsure



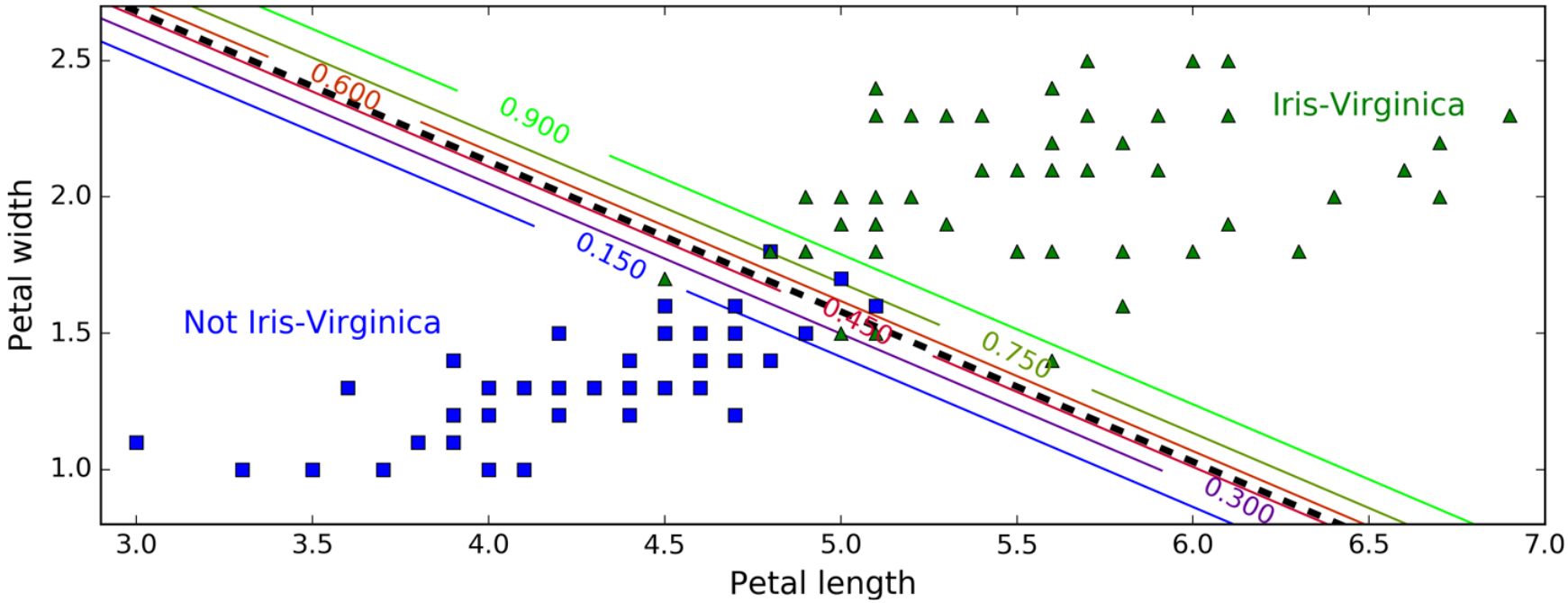
Decision Boundaries (3)

- However, if you ask it to predict the class it will return whichever class is the most likely
  - using the `predict()` method rather than the `predict_proba()` method
- Therefore, there is a *decision boundary* at around 1.6 cm where both probabilities are equal to 50%
  - if the petal width is higher than 1.6 cm, the classifier will predict that the flower is an Iris-Virginica, or
  - else it will predict that it is not
    - even if it is not very confident

Decision Boundaries (4)

- The figure in the next slide shows the same dataset but this time displaying two features: petal width and length
  - The Logistic Regression classifier can estimate the probability that a new flower is an Iris-Virginica based on these two features
- The dashed line represents the points where the model estimates a 50% probability
  - This is the model's decision boundary
- Note that it is a linear boundary

Decision Boundaries (5)



- Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right)
- All the flowers beyond the top-right line have an over 90% chance of being Iris-Virginica according to the model

Softmax Regression (1)

- The Logistic Regression model can be generalized to support multiple classes directly
  - without having to train and combine multiple binary classifiers
- This is called Softmax Regression, or Multinomial Logistic Regression
- When given an instance **x**, the Softmax Regression model first computes a score $s_k(\mathbf{x})$ for each class k, then estimates the probability of each class by applying the softmax function (also called the normalized exponential) to the scores
- The equation to compute $s_k(\mathbf{x})$ is:

$$s_k(\mathrm{x}) = \left(\theta^{(k)}\right)^T \cdot \mathrm{x}$$

Softmax Regression (2)

- Once computed the score of every class for the instance **x**, you can estimate the probability $\hat{p}_k$ that the instance belongs to class *k* by running the scores through the softmax function
  - it computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials)

$$\hat{p}_k = \sigma\big(s(\mathrm{x})\big)_k = \frac{\exp\big(s_k(\mathrm{x})\big)}{\sum_{j=1}^{K} \exp\Big(s_j(\mathrm{x})\Big)}$$

  - K is the number of classes
  - $s(\mathrm{x})$ is a vector containing the scores of each class for the instance x
  - $\sigma\big(s(\mathrm{x})\big)_k$ is the estimated probability that the instance x belongs to class k given the scores of each class for the instance

Softmax Regression (3)

- The Softmax Regression classifier predicts the class with the highest estimated probability
  - It is simply the class with the highest score as shown in the following

$$\hat{y} = \underset{k}{\operatorname{argmax}}\, \sigma\big(s(\mathrm{x})\big)_k = \underset{k}{\operatorname{argmax}}\, s_k(\mathrm{x}) = \underset{k}{\operatorname{argmax}} \left(\left(\theta^{(k)}\right)^T \cdot \mathrm{x} \right)$$

  - The argmax operator returns the value of a variable that maximizes a function
  - In this equation it returns the value of k that maximizes the estimated probability $\sigma\big(s(\mathrm{x})\big)_k$

Training Softmax Regression Models

- The Softmax Regression classifier predicts only one class at a time
  - it is multiclass, not multioutput
- It should be used only with mutually exclusive classes such as different types of plants
  - You cannot use it to recognize multiple people in one picture
- Let's take a look at training of softmax regression models
  - The objective is to have a model that estimates a high probability for the target class, and consequently a low probability for the other classes

Softmax Regression Cost Function

- The cross entropy cost function penalizes the model when it estimates a low probability for a target class

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\, log\left(\hat{p}_k^{(i)}\right)$$

  - $y_k^{(i)}$ is equal to 1 if the target class for the i^{th} instance is k
  - otherwise, it is equal to 0

- Cross entropy is frequently used to measure how well a set of estimated class probabilities match the target classes