CAP4770 – Intro to Data Science

Training Models

Prof. Ye Xia

Linear Regression

- m instances; n features.
- We observe m training data $(x_1, y_1), \ldots, (x_m, y_m)$.
- Each x_i is a vector of n feature values for instance i, $x_i \in \mathbb{R}^n$;
- Each output y_i is a scalar output (target value), where $y_i \in \mathbb{R}$.

- Suppose we have scalar variables $z_1, \ldots, z_n$ and scalar parameters $\theta_0, \theta_1, \ldots, \theta_n$.

  For convenience, we define $z = (z_1, \ldots, z_n)^T$, $\bar{z} = (1, z_1, \ldots, z_n)^T$ and $\theta = (\theta_0, \theta_1, \ldots, \theta_n)^T$.
- A function of the form $\bar{z}^T\theta = \theta_0 + \sum_{j=1}^{n} z_j\theta_j$ is affine in z and linear in θ. We will see the linearity in θ is more important.
- In linear regression, we restrict ourselves to the family of functions:

$g_\theta(z) = \bar{z}^T\theta$, where $\theta \in \mathbb{R}^{n+1}$ are the model parameters.

- The goal of training is to learn the parameters θ from the training data.
- Once the parameters θ are learned, given a new input vector $u = (u_1, \ldots, u_n)$, the predicted output $\hat{y}$ is

$$\hat{y} = \theta_0 + \sum_{j=1}^{n} u_j\theta_j.$$

- θ_0 is called the **bias** term or the **intercept**.
- $\theta_1, \ldots, \theta_n$ are called the **feature weights**. In Scikit-Learn's `LinearRegression`, they are accessed through the `coef_` instance variable.
- Note that the predicted output is a weighted sum of the features/attributes, plus the intercept.

- In general, a **linear model** is weighted sum of the transformed features, plus an intercept, i.e.,

$$\hat{y} = \theta_0 + \sum_{j=1}^{n} h_j(u_j)\theta_j.$$

Method of Least Squares

- Recall the method of **least squares**, where we find θ by solving the following problem:

$$\min_{\theta} \sum_{i=1}^{m} (y_i - (\theta_0 + \sum_{j=1}^{n} x_{ij}\theta_j))^2. \tag{1}$$

  The objective function of the minimization problem is m times the mean square error (MSE). The minimization problem is equivalent to minimizing the MSE.

- There are various ways to solve (1).

Matrix Inversion

- We will write (1) in the matrix form.
- Let $\mathbf{y} = (y_1, \ldots, y_m)^T$.
- Let $\mathbf{X}$ be the $m \times (n+1)$ matrix based on the input vectors:

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \cdots & x_{mn} \end{bmatrix}$$

- The function to be minimize in (1) can be written as

$$RSS(\theta) \triangleq (\mathbf{y} - \mathbf{X}\theta)^T(\mathbf{y} - \mathbf{X}\theta),$$

where RSS stands for residual sum of squares.

- $RSS(\theta)$ is a quadratic function. Its gradient and Hessian are:

$$\nabla RSS(\theta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta)$$

$$\nabla^2 RSS(\theta) = 2\mathbf{X}^T\mathbf{X}.$$

- By setting $\nabla RSS(\theta) = 0$, we get the *normal equation*:

$$\mathbf{X}^T\mathbf{X}\theta = \mathbf{X}^T\mathbf{y} \tag{2}$$

- The matrix $\mathbf{X}^T\mathbf{X}$ is positive semidefinite, and therefore, $RSS(\theta)$ is convex. Any solution to the normal equation minimizes $RSS(\theta)$.

  A real symmetric matrix M is positive definite if $z^T M z > 0$ for every nonzero real vector z. A real symmetric matrix is positive definite if and only if all of its eigenvalues are positive.

  For positive semidefinite, replace '> 0' and 'positive' with '≥ 0' and 'nonnegative' from the above.

- Let us assume $\mathbf{X}$ has full column rank and therefore the matrix $\mathbf{X}^T\mathbf{X}$

is positive definite. Then, $RSS(\theta)$ is strictly convex. We get the unique minimum:

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{3}$$

- $\mathbf{X}$ is an $m \times (n+1)$ matrix, where m is the number of instances in the dataset and n is the number of features. In many linear regression problems, $m \gg n$ and the data are noisy, and therefore, it is highly likely that $\mathbf{X}$ has a full (column) rank.
- What about the general situation where $\mathbf{X}$ does not necessarily have full column rank? You need to first find the pseudo inverse of $\mathbf{X}$ using singular value decomposition (SVD), which is denoted by $\mathbf{X}^\dagger$.
- The solution that minimizes $RSS(\theta)$ is

$$\hat{\theta} = \mathbf{X}^\dagger\mathbf{y}.$$

SVD and Pseudo Inverse

- Any real $m \times n$ matrix A can be decomposed as

$$A = USV^T,$$

where

  - U is a $m \times m$ orthonormal matrix ($U^TU = UU^T = I_m$);
  - V is a $n \times n$ orthonormal matrix ($V^TV = VV^T = I_n$);
  - S is a $m \times n$ matrix with $l = \min(m, n)$ entries $\sigma_i \geq 0$ on the main diagonal, 0 elsewhere. The σ_i are called **singular values** of A.

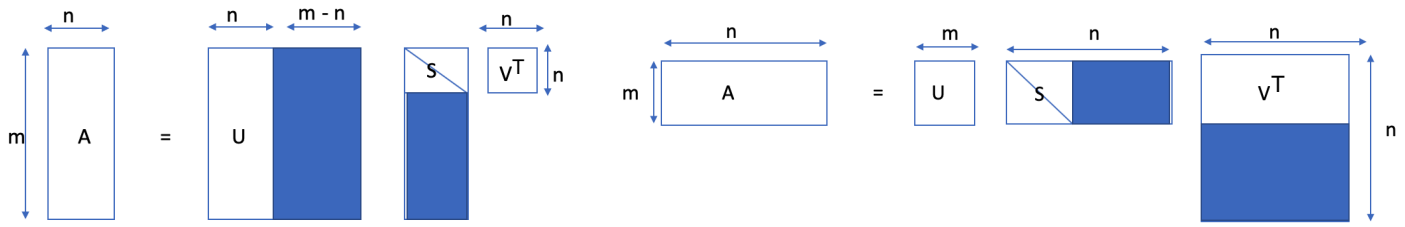


Figure 1: SVD. Left: $m \geq n$; Right: $m < n$. The shaded $m - n$ columns of U on the left figure or the shaded rows of V^T on the right figure need not to be computed since they will be multiplied with 0.

- The cost of computing SVD is $O(\min(mn^2, m^2n))$, which is great because practical problems usually have only one of m and n becoming very large.
- The (Moore-Penrose) **pseudo inverse** of a matrix $A \in \mathbb{R}^{m \times n}$, denoted by $A^\dagger$, is the unique $n \times m$ matrix that satisfies the following 4 properties:

$$AA^\dagger A = A, A^\dagger AA^\dagger = A^\dagger, (A^\dagger A)^T = A^\dagger A, (AA^\dagger)^T = AA^\dagger.$$

In that case, A is also the pseudo inverse of $A^\dagger$.

- If A is a square and non-singular matrix, then $A^\dagger = A^{-1}$.
- When A has rank n, then $A^T A$ is invertible. Then, $A^\dagger = (A^T A)^{-1} A^T$.

  $(A^T A)^{-1} A^T$ is called the *left inverse* of A.
- When A has rank m (full row rank), then AA^T is invertible. Then, $A^\dagger = A^T (AA^T)^{-1}$.

  $A^T (AA^T)^{-1}$ is called the *right inverse* of A.
- In the general case, suppose rank$(A) = r$ and suppose the SVD of A is $A = USV^T$. It can be checked that $S^\dagger$ is the $n \times m$ matrix with $1/\sigma_i$ on the diagonal for $i = 1, \ldots, r$, 0 elsewhere.

  Then, $A^\dagger = VS^\dagger U^T$. This can be verified using the definition.

Solve $Ax = u$ Again

- When a solution exists, $x = A^\dagger u$ is the minimum-norm solution.

  Proof: First, we show $x = A^\dagger u$ is a solution. By assumption that a solution exists, $u \in \text{range}(A)$. There exist z such that $Az = u$. Now,

$$Ax = A(A^\dagger u) = A(A^\dagger(Az)) = (AA^\dagger A)z = Az = u.$$

  Hence, $x = A^\dagger u$ is a solution.

Suppose y is also a solution, i.e., $Ay = u$. Then,

$$\begin{aligned}(y-x)^T x &= (y - A^\dagger u)^T (A^\dagger u) \\ &= (y - A^\dagger u)^T (A^\dagger A A^\dagger u) \\ &= (y - A^\dagger u)^T (A^\dagger A) A^\dagger u \\ &= (y - A^\dagger u)^T (A^\dagger A)^T A^\dagger u \\ &= (A^\dagger A y - A^\dagger A A^\dagger u)^T A^\dagger u \\ &= (A^\dagger u - A^\dagger u)^T A^\dagger u \\ &= 0.\end{aligned}$$

That is, $y - x$ and x are orthogonal. Then,

$$\|y\|^2 = \|y - x + x\|^2 = \|y - x\|^2 + \|x\|^2 \geq \|x\|^2 .$$

- When a solution does not exist, $x = A^\dagger u$ minimizes the least squares, i.e., $f(x) = \frac{1}{2} \|Ax - u\|^2$.

  Proof: The first-order necessary condition for optimality is that the

normal equations are satisfied by x:

$$\nabla f(x) = A^T(Ax - u) = A^T Ax - A^T u = 0.$$

We will show $x = A^\dagger u$ satisfies the normal equations.

$$\begin{aligned} A^T Ax - A^T u &= A^T AA^\dagger u - A^T u \\ &= A^T (AA^\dagger)^T u - A^T u \\ &= (AA^\dagger A)^T u - A^T u \\ &= A^T u - A^T u \\ &= 0. \end{aligned}$$

We know the Hessian of f is

$$\nabla^2 f(x) = A^T A.$$

Since $A^T A$ is positive semidefinite, the function $f(x)$ is convex. Hence, $x = A^\dagger u$ is a global minimum. It is unique iff rank$(A) = m$.

Minimizing $RSS(\theta)$ by SVD

- In our case, we want to minimize

$$RSS(\theta) = (\mathbf{y} - \mathbf{X}\theta)^T(\mathbf{y} - \mathbf{X}\theta) = \|\mathbf{y} - \mathbf{X}\theta\|^2 .$$

- We compute $\mathbf{X}^\dagger$ by SVD and set $\hat{\theta} = \mathbf{X}^\dagger\mathbf{y}$. We see that $\hat{\theta}$ always minimizes $RSS(\theta)$. There are two cases:
  - When a solution for the problem $\mathbf{X}\theta = \mathbf{y}$ exists, $\hat{\theta}$ is a minimum-norm solution. Moreover, since $\mathbf{X}\hat{\theta} = \mathbf{y}$ and hence $RSS(\hat{\theta}) = 0$, which is minimized.
  - When a solution for the problem $\mathbf{X}\theta = \mathbf{y}$ doesn't exist, $\hat{\theta}$ still minimizes $RSS(\theta)$.

Linear Regression in Scikit-Learn

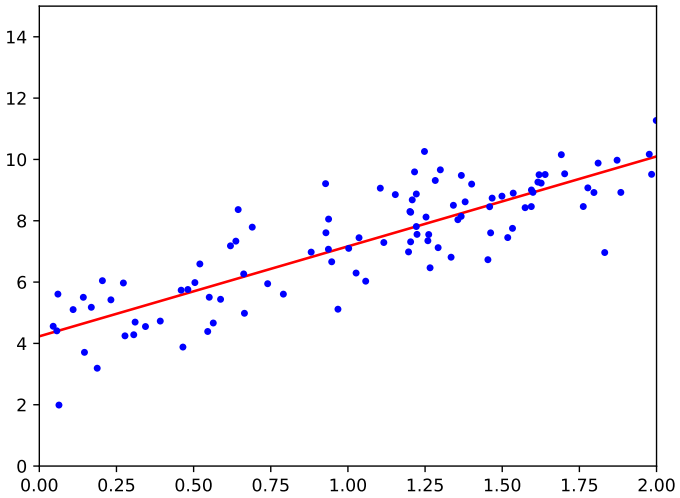
- Option 1: directly by matrix inverse: $\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$.

```
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)
```

```
theta_best
```

```
array([[4.23139434],
       [2.9334466 ]])
```

Here is the result:



- Option 2: By pseudo inverse, which relies on SVD: $\hat{\theta} = \mathbf{X}^{\dagger}\mathbf{y}$.

```
theta2=np.linalg.pinv(X_b).dot(y)
```

- Option 3: Use the `LinearRegression` class.

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y) # no need to add x0
lin_reg.intercept_, lin_reg.coef_
```

```
(array([4.23139434]), array([[2.9334466]]))
```

```
lin_reg.predict(X_new)  # X_new doesn't contain x0
```

```
array([[ 4.23139434],
       [10.09828754]])
```

The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (least squares), which you could call directly:

```
import scipy
theta_best_svd, residuals, rank, s = scipy.linalg.lstsq(X_b, y)
theta_best_svd
```

```
array([[4.23139434],
       [2.9334466 ]])
```

There is also a Numpy version of the function:
`numpy.linalg.lstsq()`.

Both functions use SVD to compute the needed pseudo inverse.

- Computation complexity:
  - Matrix inversion: $\mathbf{X}^T\mathbf{X}$ is a $(n+1) \times (n+1)$ matrix. The matrix inversion of $\mathbf{X}^T\mathbf{X}$ typically takes about $O(n^{2.4})$ to $O(n^3)$ operations, depending on the algorithm. The matrix multiplication of $\mathbf{X}^T\mathbf{X}$ takes $m^2 \times (n+1)$ operations.
  - Pseudo inverse by SVD: The cost of computing SVD of $\mathbf{X}$ is $O(\min(m(n+1)^2, (n+1)m^2))$. Since we expect $m > n$, the computation complexity is $O(mn^2)$
  - Both can be slow when the number of features grows large (e.g., 100,000).
  - We will need to consider other optimization algorithms, such as gradient descent.

A Bit of Math

Gradient

- Suppose the function $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable. The gradient of f at a point $x \in \mathbb{R}^n$, denoted by $\nabla f(x)$, is the vector of the first derivatives:

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(x) \\ \frac{\partial f}{\partial x_2}(x) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x) \end{pmatrix}$$

  Or, we write $\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right)^T$

  It is sometimes convenient to write the gradient as $\frac{\partial f}{\partial x}$.

- The gradient at each point x is a n-dimensional vector.

- At every point x, there is a gradient (vector). This gives a so-called vector field.

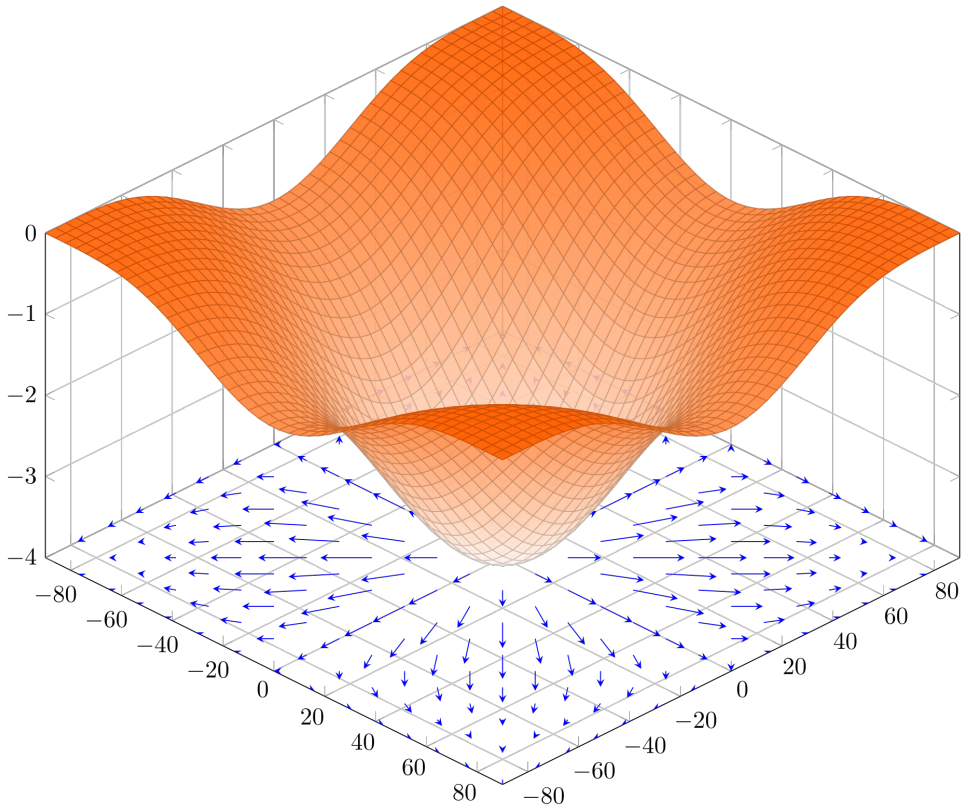


Figure 2: The gradient of the function $f(x) = -(\cos^2 x_1 + \cos^2 x_2)^2$ depicted as a vector field on the bottom plane (Wikipedia).

- Given a scalar c, the set $\{x|f(x) = c\}$ is called the **level set**.

  When $n = 2$ (i.e., $x \in \mathbb{R}^2$), a level set is also called a **level curve**.

- Example: For $f(x) = x_1^2 + x_2^2$. The gradient at the point $x = (x_1, x_2)$ is $\nabla f(x) = (2x_1, 2x_2)^T$.

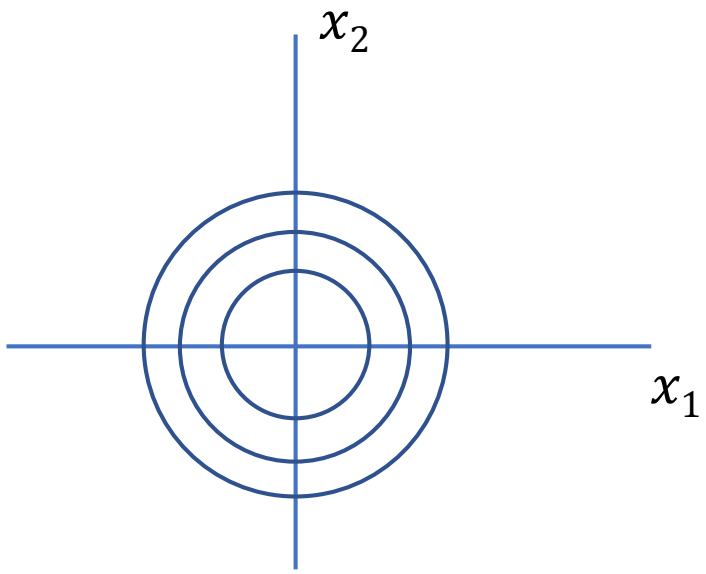


Figure 3: Three level sets for the function $f(x) = x_1^2 + x_2^2$ (for three different values of c.

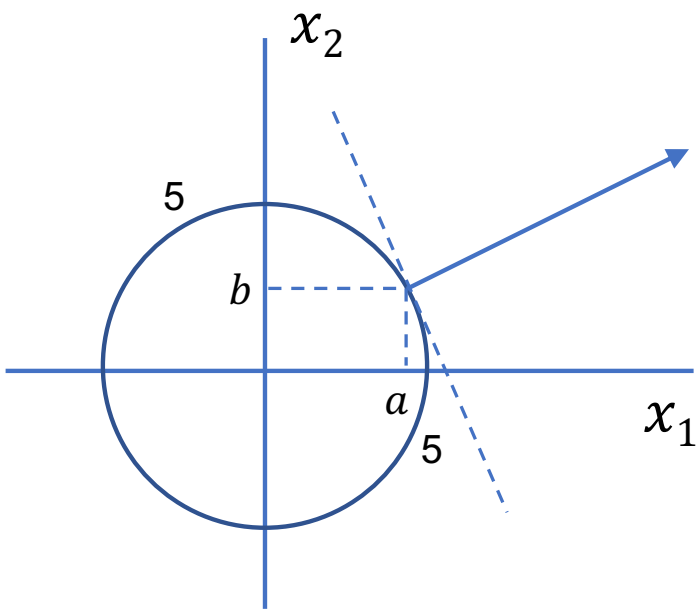


Figure 4: Level set corresponding to $f(x) = 25$ for the function $f(x) = x_1^2 + x_2^2$. The gradient at the point $x = (a, b)$ is shown as a (translated) vector, which is the vector $(2a, 2b)^T$. The gradient is perpendicular to the tangent line at (a, b).

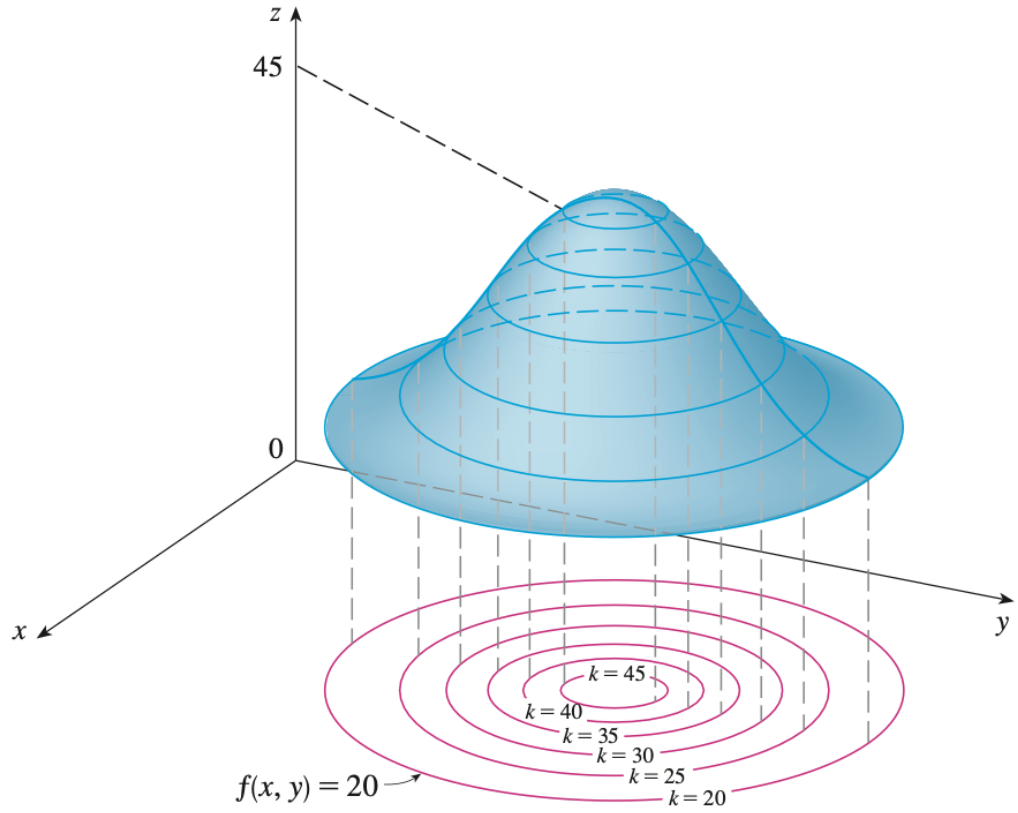


Figure 5: Level curves shown together with the function values on the level curves. Here, we are using different notations; the function is written as $z = f(x, y)$, where $x, y, z \in \mathbb{R}$ (From James Steward's Calculus).

- For a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, the "best" linear approximation of the function around a point $x_o \in \mathbb{R}^n$ is $f(x_o) + \nabla f(x_o)^T (x - x_o)$. That is

$$f(x) \approx f(x_o) + \nabla f(x_o)^T (x - x_o),$$

  for x around x_o. It is the best linear approximation because it is the first two terms of the Taylor series.

- The gradient $\nabla f(x)$ points to the direction where the function f at the point x has the greatest increase in value (in the sense of rate of change, i.e., directional derivative).

  In other words, for the function f at the point x, the largest directional derivative is in the direction of $\nabla f(x)$.

  Due to linearity of derivatives, $-\nabla f(x)$ points to the direction where the function f at the point x has the greatest decrease in value. (Think about the $-f$ function.)

Some Properties of Gradient

- Product Rule: If $f, g : \mathbb{R}^n \to \mathbb{R}$ are differentiable functions, then

$$\nabla(fg)(x) = \nabla f(x)g(x) + f(x)\nabla g(x). \tag{4}$$

- **Chain Rule - Version 1:** If $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}^n$ are differential functions, then

$$(f \circ g)'(x) = \nabla f(g(x)) \cdot g'(x) = \nabla f(g(x))^T g'(x), \tag{5}$$

  where $\cdot$ is the dot product of two vectors. This is the normal chain rule in multivariable calculus and it is easily provable by the definition of derivative.

  Note $g'(x)$ is a vector of derivatives w.r.t the scalar variable x.

  In anticipation of later generalization, note that $g'(x)^T = \nabla g(x)$; the

latter is the gradient matrix. Then, (5) can also be written as

$$(f \circ g)'(x) = g'(x)^T \nabla f(g(x)) = \nabla g(x) \nabla f(g(x)).$$

Version 2: If $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ are differential functions, then

$$\nabla(g \circ f)(x) = g'(f(x)) \nabla f(x) = \nabla f(x) g'(f(x)). \qquad (6)$$

Gradient Matrix and Jacobian Matrix

- Consider a function $f : \mathbb{R}^n \to \mathbb{R}^m$, which can be viewed as m functions, each from $\mathbb{R}^n$ to $\mathbb{R}$. In other words, we can write $f = (f_1, f_2, \ldots, f_m)^T$, where each $f_i : \mathbb{R}^n \to \mathbb{R}$.
- Suppose all the f_i are differentiable at $x \in \mathbb{R}^n$. Then, each f_i has a gradient at x, $\nabla f_i(x)$.
- We define the **gradient matrix** for f as

$$\nabla f(x) = (\nabla f_1(x), \nabla f_2(x), \ldots, \nabla f_m(x)).$$

  $\nabla f(x)$ is a $n \times m$ matrix.
- The gradient matrix generalizes the notion of gradient.
- The **Jacobian matrix** of the function $f : \mathbb{R}^n \to \mathbb{R}^m$, denoted by $\mathbf{J}$ (or $\mathbf{J}_f$ or Df), is a $m \times n$ matrix whose (i, j)th entry is $\frac{\partial f_i}{\partial x_j}$. In other

words,

$$\mathbf{J} = \left(\frac{\partial f_i}{\partial x_j}\right)_{ij} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla f_1(x)^T \\ \nabla f_2(x)^T \\ \vdots \\ \nabla f_m(x)^T \end{bmatrix}$$

- We see that the Jacobian and gradient matrices are matrix transpose of each other:

$$\mathbf{J}^T = \nabla f \qquad \text{or} \qquad \mathbf{J} = (\nabla f)^T.$$

- **Chain Rule (generalized):** If $f : \mathbb{R}^n \to \mathbb{R}^k$ and $g : \mathbb{R}^k \to \mathbb{R}^m$ are differential functions, then $g \circ f$ is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$.

  Chain rule expressed in the gradient matrices:

$$\nabla(g \circ f)(x) = \nabla f(x) \nabla g(f(x)). \tag{7}$$

(Need to flip the order of f and g.)

The above is the same as

$$\mathbf{J}^T_{(g\circ f)}(x) = \mathbf{J}^T_f(x)\mathbf{J}^T_g(f(x)).$$

Taking the transpose, the chain rule expressed in the Jacobian matrices is:

$$\mathbf{J}_{(g\circ f)}(x) = \mathbf{J}_g(f(x))\mathbf{J}_f(x).$$

(No need to flip the order of f and g.)

More Identities in Vector and Matrix Calculus

- Suppose $a, b \in \mathbb{R}^n$ are vectors and A is a matrix. Suppose $x \in \mathbb{R}^n$.
- Then, $a^T x$ is a function from $\mathbb{R}^n$ to $\mathbb{R}$. Its gradient is:

$$\nabla(a^T x) = a. \tag{8}$$

Implication: If A is a matrix, then the gradient matrix and the Jacobian satisfy

$$\nabla(Ax) = A^T, \text{ or } \mathbf{J} = A. \tag{9}$$

- Dot product: Suppose $f, g : \mathbb{R}^n \to \mathbb{R}^m$. Then,

$$\nabla(f \cdot g)(x) = \nabla f(x) g(x) + \nabla g(x) f(x) = \mathbf{J}_f^T(x) g(x) + \mathbf{J}_g^T(x) f(x). \tag{10}$$

Note that $\nabla(f \cdot g)$ is a gradient (vector), and ∇f and ∇g are $n \times m$ gradient matrices. How to remember the first formula: It has to be matrix $\times$ a column vector to yield a column vector.

By taking the transpose, (10) can also be written as:

$$\mathbf{J}_{f\cdot g}(x) = g(x)^T \mathbf{J}_f(x) + f(x)^T \mathbf{J}_g(x). \tag{11}$$

Proof: Use the product rule (4) for real-valued functions, we get

$$\begin{aligned}
\nabla(f\cdot g) &= \nabla\left(\sum_{j=1}^{m} f_j g_j\right) = \sum_{j=1}^{m} \nabla(f_j g_j) \\
&= \sum_{j=1}^{m} (g_j \nabla f_j + f_j \nabla g_j) = \sum_{j=1}^{m} g_j \nabla f_j + \sum_{j=1}^{m} f_j \nabla g_j \\
&= (\nabla f_1, \ldots, \nabla f_m) \begin{pmatrix} g_1 \\ \vdots \\ g_m \end{pmatrix} + (\nabla g_1, \ldots, \nabla g_m) \begin{pmatrix} f_1 \\ \vdots \\ f_m \end{pmatrix} \\
&= (\nabla f) g + (\nabla g) f.
\end{aligned}$$

- $x^T Ax$ is a function from $\mathbb{R}^n$ to $\mathbb{R}$. We have

$$\nabla(x^T Ax) = (A + A^T)x. \tag{12}$$

  Why? Note that $\nabla x = I$, the identity matrix.

$$\begin{aligned}\nabla(x^T Ax) &= \nabla(x \cdot Ax) = \nabla(x)Ax + \nabla(Ax)x \\ &= IAx + A^T x = (A + A^T)x.\end{aligned}$$

- Example:

$$RSS(\theta) = (\mathbf{y} - \mathbf{X}\theta)^T(\mathbf{y} - \mathbf{X}\theta) = (\mathbf{y} - \mathbf{X}\theta) \cdot (\mathbf{y} - \mathbf{X}\theta).$$

  One method:

$$\begin{aligned}RSS(\theta) &= \mathbf{y}^T\mathbf{y} - \theta^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\theta + \theta^T\mathbf{X}^T\mathbf{X}\theta \\ &= \mathbf{y}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\theta - \mathbf{y}^T\mathbf{X}\theta + \theta^T\mathbf{X}^T\mathbf{X}\theta \\ &= \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\theta + \theta^T\mathbf{X}^T\mathbf{X}\theta.\end{aligned}$$

Using (9) and (12), we get

$$\nabla RSS(\theta) = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\theta = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta).$$

Another method is to start with (10).

$$\begin{aligned}\nabla RSS(\theta) &= 2\left(\nabla(\mathbf{y} - \mathbf{X}\theta)\right)(\mathbf{y} - \mathbf{X}\theta) \\ &= -2\left(\nabla(\mathbf{X}\theta)\right)(\mathbf{y} - \mathbf{X}\theta) \\ &= -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta),\end{aligned}$$

where in the last step we used (9).

Hessian

- Suppose the function $f : \mathbb{R}^n \to \mathbb{R}$ has all the second-order partial derivatives. The **Hessian** of f, denoted by $\mathbf{H}_f$, is the (symmetric) $n \times n$ matrix of all the second derivatives.

$$\mathbf{H}_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ & & \vdots & \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{13}$$

- In short, $(\mathbf{H}_f)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$.

- Note that the first row is the transpose of the gradient of $\frac{\partial f}{\partial x_1}$, i.e., $(\nabla \frac{\partial f}{\partial x_1})^T$. The first column is the gradient of $\frac{\partial f}{\partial x_1}$, i.e., $\nabla \frac{\partial f}{\partial x_1}$.

Hence, the Hessian of f is also the Jacobian of the gradient of f. Since the Hessian is a symmetric matrix, it is also equal to the gradient matrix of the gradient of f. That is,

$$\mathbf{H}_f = \mathbf{J}_{\nabla f} = \nabla(\nabla f).$$

By convention, $\nabla(\nabla f)$ is written as $\nabla^2 f$, and hence, we can write

$$\mathbf{H}_f = \nabla^2 f.$$

- Example: we have shown

$$\nabla RSS(\theta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta).$$

Then, using (9), we have

$$\begin{aligned}\nabla^2 RSS(\theta) &= -2\nabla\left(\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta)\right) \\ &= 2\nabla\left(\mathbf{X}^T\mathbf{X}\theta\right) = 2\left(\mathbf{X}^T\mathbf{X}\right)^T \\ &= 2\mathbf{X}^T\mathbf{X}.\end{aligned}$$

- Similarly, the Hessian of the quadratic function $x^T Ax$ is

  $$\nabla^2(x^T Ax) = \nabla\nabla(x^T Ax) = \nabla((A+A^T)x) = (A+A^T)^T = A+A^T.$$

  For a quadratic function $x^T Ax$, one can always choose A to be a symmetric matrix. Then,

  $$\nabla^2(x^T Ax) = 2A.$$

Gradient Descent

- Gradient descent is a very different way to train a linear regression model, which is better suited for cases where there are a large number of features or too many training instances to fit in memory.
- Gradient descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems, not just the least squares problem.
- Suppose we are minimizing the cost function $f(\theta)$.

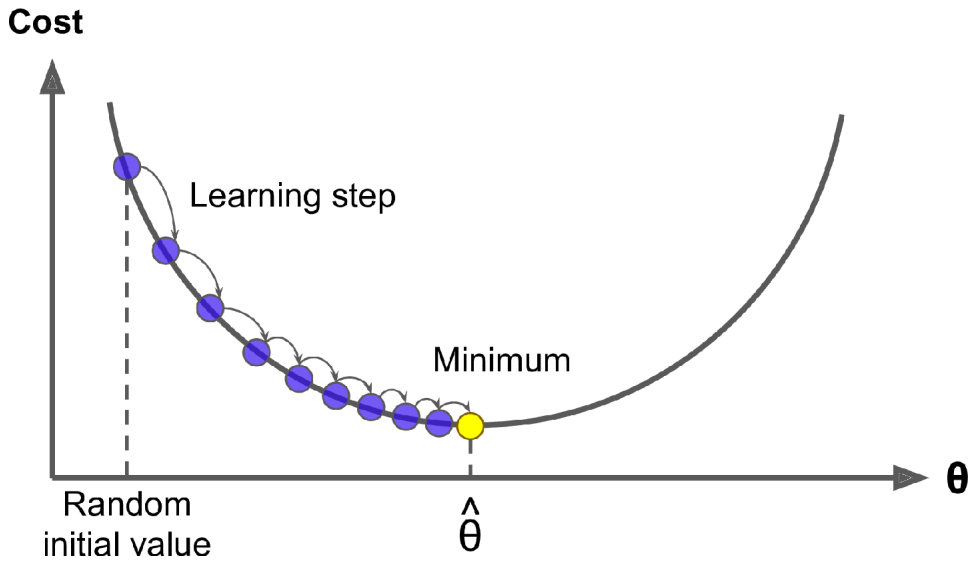


Figure 6: The negative gradient direction at each iteration step is to the right.

- **Gradient Descent (aka Steepest Descent:)** At each step of the iteration, say kth iteration, we are at $\theta^{(k)}$. Then, $\theta^{(k+1)}$ should be at

$$\theta^{(k+1)} = \theta^{(k)} - \alpha^{(k)} \nabla f(\theta^{(k)}). \tag{14}$$

  - $\alpha^{(k)} > 0$ is the step size (aka learning rate) at iteration k. It may be taken as a constant α.

– $\{\alpha^{(k)}\}$ cannot be too large or too small. In the former case, the gradient descent algorithm may not converge; in the latter case, the convergence speed is too slow.

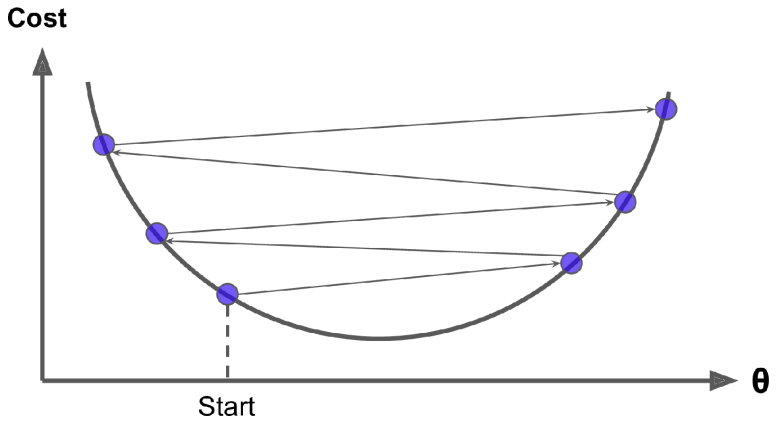


Figure 7: The step size (learning rate) is too large.

– Note that since we are minimizing the cost, we are moving into the $-\nabla f(\theta^{(k)})$ direction, which is the direction of the steepest descent.

– Under a suitable step size rule, the gradient descent algorithm

converges to a (local) minimum. For a convex problem, this will be a global minimum.

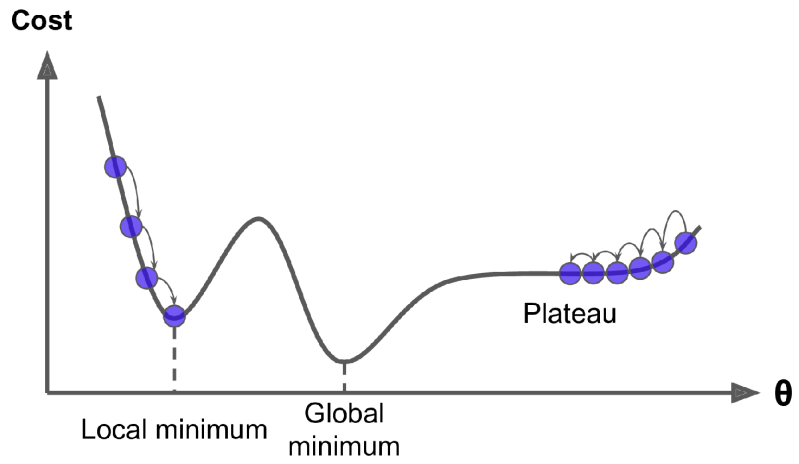


Figure 8: Possible issues: Starting from the left leads to convergence to a local minimum. Starting from the right leads to slow convergence due to the flat area.

- Why is feature scaling needed? Recall that the gradient of f is normal to the level curve.

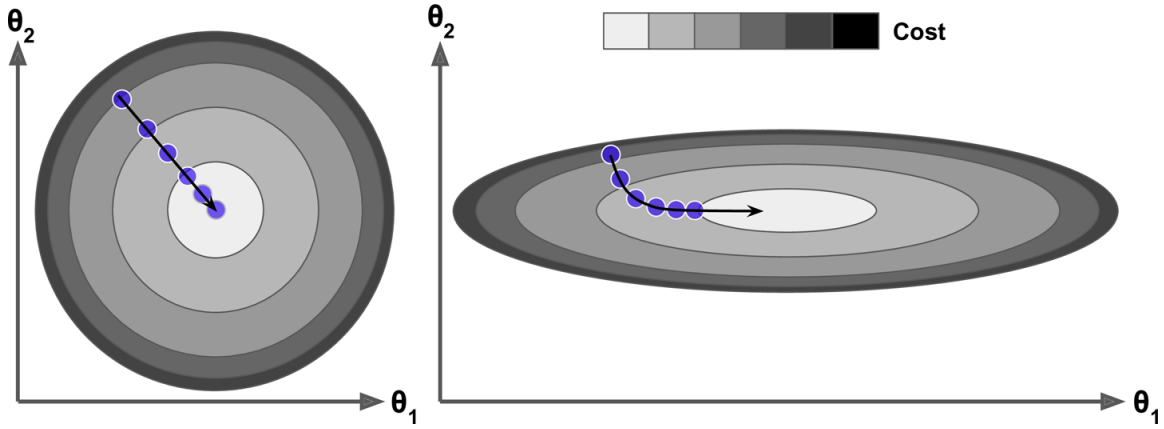


Figure 9: Left: Fast convergence. Right: Slow convergence.

$$\text{Left: } f(x) = \theta_1^2 + \theta_2^2 = (\theta_1, \theta_2) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix}$$

$$\text{Right: } f(x) = \frac{1}{4}\theta_1^2 + 4\theta_2^2 = (\theta_1, \theta_2) \begin{pmatrix} 1/4 & 0 \\ 0 & 4 \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix}$$

Convergence Speed of Gradient Descent

- Fact: Let A be $n \times n$ real symmetric matrix. Suppose its eigenvalues are ordered such that $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. Then, for all $y \in \mathbb{R}^n$,

$$\lambda_1 \|y\|^2 \leq y^T A y \leq \lambda_n \|y\|^2 .$$

- Consider a quadratic function $f(x) = \frac{1}{2} x^T A x$. Assume A is positive definite. Then, the minimum is at $x^* = 0$. Also, at any x,

$$\nabla f(x) = \frac{1}{2}(A + A^T)x = Ax, \nabla^2 f(x) = A^T = A.$$

  The gradient descent algorithm is:

$$x^{(k+1)} = x^{(k)} - \alpha^{(k)} \nabla f(x^{(k)}) = x^{(k)} - \alpha^{(k)} A x^{(k)} = (I - \alpha^{(k)} A) x^{(k)}.$$

- Suppose the smallest and the largest eigenvalues of A are m and M.

- The best convergence rate bound under a constant step size is:

$$\frac{\left\|x^{(k+1)}\right\|}{\left\|x^{(k)}\right\|} \leq \frac{M-m}{M+m}.$$

- Under the step size $\frac{2}{M+m}$, the sequence $\left\|x^{(k)}\right\|$ converges to 0 (this is equivalent to $x^{(k)} \to 0$) no slower than $\left(\frac{M-m}{M+m}\right)^k$.

  Why no slower? Here and in the following, we make statements by using the tightest upper bound. It turns out this upper bound is sharp in the following sense: For any A, there is a starting point $x^{(0)}$ such that the inequality holds as equality for all k.

- M/m is called the **condition number** of A. If M/m is not very large, the convergence speed is very fast.

- If M/m is very large, the matrix A is said to be **ill-conditioned**. Such problems are characterized by a very elongated elliptical level sets. The gradient descent algorithm converges slowly in such cases.

- It can also be shown

$$\frac{f(x^{(k+1)})}{f(x^{(k)})} \leq \left(\frac{M-m}{M+m}\right)^2.$$

For the quadratic function, the minimum function value is $f^* = f(0) = 0$. We see that

$$f(x^{(k)}) - f^* = f(x^{(k)}) \leq \left(\frac{M-m}{M+m}\right)^{2k} = \left(\frac{1-m/M}{1+m/M}\right)^{2k}.$$

If we want $f(x^{(k)}) \leq \left(\frac{1-m/M}{1+m/M}\right)^{2k} \leq \epsilon$ for some small positive ϵ, we need

$$k \geq \frac{\log \epsilon}{2 \log\left(\frac{1-m/M}{1+m/M}\right)}.$$

For very large M/m, we have

$$k \geq \frac{\log \epsilon}{2 \log \left(\frac{1-m/M}{1+m/M} \right)} \approx \frac{\log \epsilon}{2 \log(1 - m/M)} \approx \frac{-\log \epsilon}{2m/M}$$
$$= \frac{\log(1/\epsilon)}{2m/M} = \log(1/\epsilon) \frac{M}{2m}.$$

(In the above, $\log$ is with base e.)

Even for a very small ϵ, $\log(1/\epsilon)$ is not a big number. However, if $M \gg m$, k can be very large.

- When the matrix A is singular, then $m = 0$. This corresponds to the situation the condition number is ∞. The gradient descent algorithm may converge very slowly.

  Under some technical conditions, it can be shown
  $f(x^{(k)}) - f^* = f(x^{(k)}) \leq c/k$, where c is a constant.

  This means to reach $f(x^{(k)}) - f^* \leq \epsilon$, it takes $O(1/\epsilon)$ iterations.

Scaled Gradient Descent

- Consider a more general scaled gradient descent algorithm:

$$x^{(k+1)} = x^{(k)} - \alpha^{(k)} D^{(k)} \nabla f(x^{(k)}),$$

  where $D^{(k)}$ is a positive definite matrix.

- Let

$$S = (D^{(k)})^{1/2}.$$

- Consider the transformation $x = Sy$. Let

$$h(y) = f(Sy) = \frac{1}{2}(Sy)^T ASy = \frac{1}{2} y^T S^T ASy = \frac{1}{2} y^T SASy.$$

- The gradient descent algorithm for h is:

$$y^{(k+1)} = y^{(k)} - \alpha^{(k)} \nabla h(y^{(k)}).$$

Multiplying by S on both side, we get

$$Sy^{(k+1)} = Sy^{(k)} - \alpha^{(k)} S\nabla h(y^{(k)}).$$

Noting

$$x^{(k+1)} = Sy^{(k+1)}, x^{(k)} = Sy^{(k)}, \nabla h(y^{(k)}) = S\nabla f(x^{(k)}), S^2 = D^{(k)},$$

we get back

$$x^{(k+1)} = x^{(k)} - \alpha^{(k)} D^{(k)} \nabla f(x^{(k)})$$

- In summary,

  scaled gradient algorithm on $x \Longleftrightarrow$

  linear change of variables via $x = Sy$**, and steepest descent on** y**.**

- We now have

$$\frac{\left\|y^{(k+1)}\right\|}{\left\|y^{(k)}\right\|} \leq \frac{M-m}{M+m},$$

where M and m are the largest and smallest eigenvalues of the Hessian of $h(y)$, which is

$$\nabla^2 h(y) = SAS.$$

- If we choose $D^{(k)} \approx A^{-1}$ and therefore $S = (D^{(k)})^{1/2} \approx A^{-1/2}$, then $SAS = SA^{1/2}A^{1/2}S \approx I$ and $M \approx m$.

  Here, $A^{-1/2}$ denotes the inverse of $A^{1/2}$.

  Then,

  $$\frac{\left\|y^{(k+1)}\right\|}{\left\|y^{(k)}\right\|} \approx 0.$$

  The convergence rate is fast.

- We can see this in the x-space. Since $y = S^{-1}x$,

  $$\|y\|^2 = (S^{-1}x)^T S^{-1}x = x^T S^{-2}x = x^T (D^{(k)})^{-1}x.$$

Then,

$$\frac{\left\|y^{(k+1)}\right\|^2}{\left\|y^{(k)}\right\|^2} = \frac{(x^{(k+1)})^T (D^{(k)})^{-1} x^{(k+1)}}{(x^{(k)})^T (D^{(k)})^{-1} x^{(k)}} \leq \left(\frac{M-m}{M+m}\right)^2.$$

When $D^{(k)} = A^{-1}$, $M = m = 1$, and

$$\frac{(x^{(k+1)})^T (D^{(k)})^{-1} x^{(k+1)}}{(x^{(k)})^T (D^{(k)})^{-1} x^{(k)}} = \frac{(x^{(k+1)})^T A x^{(k+1)}}{(x^{(k)})^T A x^{(k)}} = \frac{f(x^{(k+1)})}{f(x^{(k)})} = 0.$$

Newton's Algorithm

- **Newton's algorithm:** For the quadratic function $f(x) = \frac{1}{2}x^T Ax$, the Hessian of f is A, i.e., $\nabla^2 f = A$. When we choose $D^{(k)} = A^{-1}$, it is the same as choosing $D^{(k)} = \left(\nabla^2 f(x^{(k)})\right)^{-1}$.
- When f is not a quadratic function, we can choose $D^{(k)} = \left(\nabla^2 f(x^{(k)})\right)^{-1}$. Under suitable technical condition, the algorithm converges very fast.

Diagonal Scaling

- **Diagonal scaling:** It is often difficult to compute the inverse of the Hessian. However, one can choose $D^{(k)}$ that approximates $\left(\nabla^2 f(x^{(k)})\right)^{-1}$. In particular, $D^{(k)}$ can be the diagonal matrix

$$D^{(k)} = \begin{pmatrix} d_1^{(k)} & & & \\ & d_2^{(k)} & & \\ & & \ddots & \\ & & & d_n^{(k)}. \end{pmatrix}$$

  where $d_i^{(k)} = \left(\frac{\partial^2 f(x^{(k)})}{(\partial x_i)^2}\right)^{-1}$.

- Recall $S = (D^{(k)})^{1/2}$ and $y = S^{-1}x$. The diagonal entries of

$S^{-1} = (D^{(k)})^{-1/2}$ are

$$(d_i^{(k)})^{-1/2} = \left(\frac{\partial^2 f(x^{(k)})}{(\partial x_i)^2} \right)^{1/2}.$$

Then, $y_i = (d_i^{(k)})^{-1/2} x_i$. We see that the diagonal scaling corresponds to variable scaling.

- For the quadratic function $f(x) = \frac{1}{2} x^T A x$ where $A = (a_{ij})$ is a symmetric matrix,

$$(d_i^{(k)})^{-1/2} = a_{ii}^{1/2}.$$

Then, $y_i = a_{ii}^{1/2} x_i$.

Back to Linear Regression Problem

Diagonal Scaling and Feature Scaling

- For the linear regression problem, for convenience, we add the factor $1/2$ to it. Minimizing $RSS(\theta)$ is the same as minimizing $\frac{1}{2}RSS(\theta)$.

$$\begin{aligned}\frac{1}{2}RSS(\theta) &= \frac{1}{2}\left\|\mathbf{y}-\mathbf{X}\theta\right\|^2 = \frac{1}{2}(\mathbf{y}-\mathbf{X}\theta)^T(\mathbf{y}-\mathbf{X}\theta) \\ &= \frac{1}{2}(\theta^T\mathbf{X}^T\mathbf{X}\theta - 2\mathbf{y}^T\mathbf{X}\theta + \mathbf{y}^T\mathbf{y})\end{aligned}$$

- For the quadratic form $\frac{1}{2}\theta^T\mathbf{X}^T\mathbf{X}\theta$, we let $A = \mathbf{X}^T\mathbf{X}$. Suppose the matrix A has entries a_{jl}, where $0 \le j, l \le n$, and the matrix $\mathbf{X}$ has entries x_{ij}, where $1 \le i \le m$ and $0 \le j \le n$.

  The diagonal entries of A are

$$a_{jj} = \sum_{l=1}^{m} x_{lj}^2.$$

Note: A is a $(n+1) \times (n+1)$ dimensional matrix. In a_{jj}, the subscript j is an index for feature. Since we index the features as $0, 1, \ldots, n$, where feature 0 corresponds to the intercept, we let the row and column indices for the matrix A to be $0, 1, \ldots, n$.

For feature j, a_{jj} is the result of summing x_{lj}^2 over all the instances. Therefore, $\sqrt{a_{jj}}$ is the l_2 norm of the vector for feature j.

- For diagonal scaling, the diagonal matrix is $D^{(k)} = \text{diag}((a_{00})^{-1}, \ldots, (a_{n,n})^{-1})$. Then,

$$S = \text{diag}((a_{00})^{-1/2}, \ldots, (a_{n,n})^{-1/2})$$
$$S^{-1} = \text{diag}((a_{00})^{1/2}, \ldots, (a_{n,n})^{1/2}).$$

The transformation of the variables is $\nu = S^{-1}\theta$. In the component form, it is

$$\nu_j = (a_{jj})^{1/2}\theta_j = \left(\sum_{l=1}^{m} x_{lj}^2\right)^{1/2} \theta_j.$$

- Note that $S = S^T$. Then,

$$RSS(\theta) = RSS(S\nu) = (S\nu)^T \mathbf{X}^T \mathbf{X}(S\nu) - 2\mathbf{y}^T \mathbf{X}(S\nu) + \mathbf{y}^T \mathbf{y}$$
$$= (\nu)^T (\mathbf{X}S)^T (\mathbf{X}S)\nu - 2\mathbf{y}^T \mathbf{X}S\nu + \mathbf{y}^T \mathbf{y}.$$

- Suppose we write $\mathbf{X}$ in the column vectors $\mathbf{X} = (w_0, w_1, \ldots, w_n)$ where each m-dimensional vector w_j corresponds to feature j. Then,

$$\mathbf{X}S = \left(\frac{w_0}{(a_{00})^{1/2}}, \frac{w_1}{(a_{11})^{1/2}}, \ldots, \frac{w_n}{(a_{n,n})^{1/2}} \right).$$

  We see that $\mathbf{X}S$ corresponds to feature scaling.

- **Why we do feature scaling in ML:** If we do feature scaling first and then apply the steepest descent algorithm, we are minimizing the objective function:

$$\nu^T (\mathbf{X}S)^T (\mathbf{X}S)\nu - 2\mathbf{y}^T \mathbf{X}S\nu + \mathbf{y}^T \mathbf{y}$$

in the ν-space using the steepest descent algorithm.

That is equivalent to minimizing $RSS(\theta)$ over θ using **diagonal scaling**, which gives faster convergence.

Min-Max Scaling and Standard Scaling

- In both methods of scaling, the data is first shifted. We will first show: For any j, if we shift all the feature-j values by a common amount for all instances, the optimal solution for linear regression does not change except the intercept.
- We write $RSS(\theta)$ in the scalar form.

$$RSS(\theta) = \sum_{i=1}^{m} \big(y_i - (\theta_0 + \sum_{j=1}^{n} x_{ij}\theta_j)\big)^2.$$

- For each feature $j = 1, \ldots, n$, suppose u_j is some constant. Let us make the substitution $z_{ij} = x_{ij} - u_j$ for all instance i.
- Now, suppose we do linear regression with $\{z_{ij}\}$ as the input data.

Suppose $\theta^* = (\theta_0^*, \theta_1^*, \ldots, \theta_n^*)$ minimizes

$$Z(\theta) \triangleq \sum_{i=1}^{m} \big(y_i - (\theta_0 + \sum_{j=1}^{n} z_{ij}\theta_j)\big)^2.$$

- Define the vector $\hat{\theta}$ as

$$\hat{\theta} = (\theta_0^* - \sum_{j=1}^{n} u_j\theta_j^*, \theta_1^*, \theta_2^*, \ldots, \theta_n^*).$$

- We claim $\hat{\theta}$ is the optimal solution to the original linear regression problem with $\{x_{ij}\}$ as the input data. That is, $\hat{\theta}$ minimizes

$$RSS(\theta) = \sum_{i=1}^{m} \big(y_i - (\theta_0 + \sum_{j=1}^{n} x_{ij}\theta_j)\big)^2.$$

Proof: Suppose claim is not true. Then, there exists $\theta = (\theta_0, \theta_1, \ldots, \theta_n)$ such that

$$\sum_{i=1}^{m}\big(y_i - (\theta_0 + \sum_{j=1}^{n} x_{ij}\theta_j)\big)^2 < \sum_{i=1}^{m}\big(y_i - (\hat{\theta}_0 + \sum_{j=1}^{n} x_{ij}\hat{\theta}_j)\big)^2.$$

This can be written as

$$\sum_{i=1}^{m}\big(y_i - (\theta_0 + \sum_{j=1}^{n}(z_{ij} + u_j)\theta_j)\big)^2 < \sum_{i=1}^{m}\big(y_i - (\hat{\theta}_0 + \sum_{j=1}^{n}(z_{ij} + u_j)\hat{\theta}_j)\big)^2.$$

Then,

$$\begin{aligned}
&\sum_{i=1}^{m}\big(y_i - (\theta_0 + \sum_{j=1}^{n} u_j\theta_j + \sum_{j=1}^{n} z_{ij}\theta_j)\big)^2 \\
<&\sum_{i=1}^{m}\big(y_i - (\hat{\theta}_0 + \sum_{j=1}^{n} u_j\hat{\theta}_j + \sum_{j=1}^{n} z_{ij}\hat{\theta}_j)\big)^2 \\
=&\sum_{i=1}^{m}\big(y_i - (\hat{\theta}_0 + \sum_{j=1}^{n} u_j\theta_j^* + \sum_{j=1}^{n} z_{ij}\theta_j^*)\big)^2 \\
=&\sum_{i=1}^{m}\big(y_i - (\theta_0^* + \sum_{j=1}^{n} z_{ij}\theta_j^*)\big)^2
\end{aligned}$$

We see that the vector $(\theta_0 + \sum_{j=1}^{n} u_j\theta_j, \theta_1, \theta_2, \ldots, \theta_n)$ leads to a lower value for

$Z()$ than $Z(\theta^*)$, which contradicts the optimality of θ^* for $Z()$. □

- For min-max scaling, $u_j = \min_i x_{ij}$ for each $j = 1, \ldots, n$. Each feature-j values is first shifted down by u_j for $j \geq 1$. Then, all the values of feature j is divided by the maximum value of feature j, i.e., the max-norm of the feature-j vector (after the shifting).

  Min-max scaling is not the same as diagonal scaling with the factor $1/(a_{jj})^{1/2}$ for each feature j. We see earlier each $(a_{jj})^{1/2}$ is the l_2 norm of the feature-j vector.

- For standard scaling, $\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_{ij}$. After each feature-$j$ values is first shifted down by u_j (for $j \geq 1$), the new feature-j vector has a mean 0, i.e., $\frac{1}{m} \sum_{i=1}^{m} z_{ij} = \frac{1}{m} \sum_{i=1}^{m} (x_{ij} - u_j) = 0$.

  For scaling, each feature-j value, z_{ij}, is divided by σ_j, which is the

standard deviation of the feature-j vector:

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(z_{ij})^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{ij} - u_j)^2.$$

If a_{jj} is computed after the data has been shifted, then

$$a_{jj} = m\sigma_j^2.$$

In standard scaling, when we divide the values of feature j by σ_j for each $j \geq 1$, it is the same as multiplication by a scaling factor of $\sqrt{m}/(a_{jj})^{1/2}$.

In diagonal scaling, the scaling factor is $1/(a_{jj})^{1/2}$. Therefore, standard scaling and diagonal scaling are equivalent. The extra factor $\sqrt{m}$ is the same for all the features, and it can be absorbed by the step size.

Batch Gradient Descent

- Return to the gradient of $RSS(\theta)$ (also with the original indexing of the features).

$$\nabla RSS(\theta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta).$$

- We will also write the component version, which is useful:

$$RSS(\theta) = \sum_{i=1}^{m}(y_i - (\theta_0 + \sum_{j=1}^{n} x_{ij}\theta_j))^2.$$

  Hence, for $k = 1, \ldots, n$,

$$\frac{\partial RSS(\theta)}{\partial \theta_k} = -2\sum_{i=1}^{m}(y_i - (\theta_0 + \sum_{j=1}^{n} x_{ij}\theta_j))x_{ik}.$$

- Equivalently, we can minimize the the mean square error $MSE(\theta)$,

which is related to $RSS(\theta)$ by

$$MSE(\theta) = \frac{1}{m} RSS(\theta).$$

Then,

$$\frac{\partial MSE(\theta)}{\partial \theta_k} = \frac{-2}{m} \sum_{i=1}^{m} (y_i - (\theta_0 + \sum_{j=1}^{n} x_{ij}\theta_j)) x_{ik}. \qquad (15)$$

- At each iteration step of the gradient descent, the gradient is calculated based on the full training set $\mathbf{X}$. This is why the algorithm is also called **Batch Gradient Descent**.
- As a result, it is very slow on very large training sets.
- However, gradient descent scales well with the number of features. Training a linear regression model with hundreds of thousands of features is much faster using gradient descent than using the normal equation or SVD decomposition.

- Consider a regression example where the input is a scalar.

```
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 a column of 1s
```

```
eta = 0.1 # learning rate
n_iterations = 1000
m=100

theta = np.random.randn(2,1) # random initialization
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

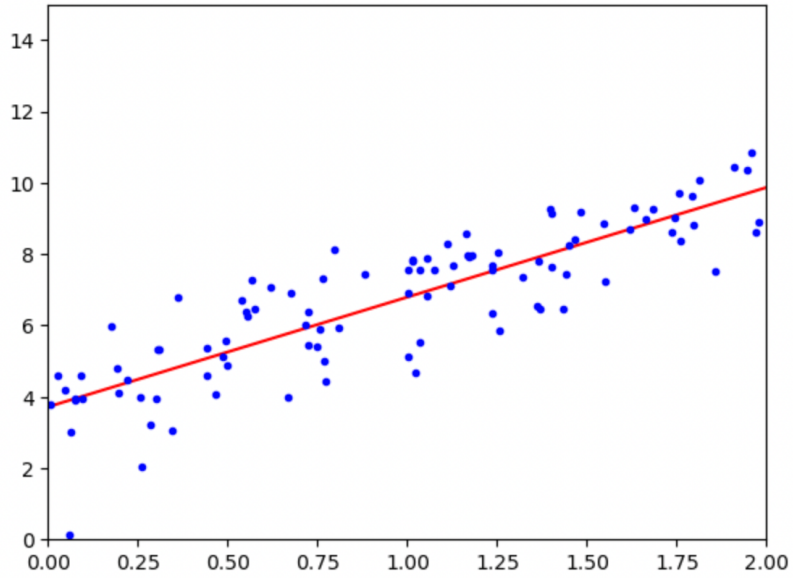
```
theta
```

```
array([[3.71906927],
       [3.0748614 ]])
```

- Here, the algorithm iterates 1000 times; in each iteration, the whole training dataset is used, which has $m = 100$ instances.

- In the code, `X_b` is our $\mathbf{X}$. `X_b.dot(theta)` is matrix multiplication $\mathbf{X}\theta$. `X_b.T.dot(v)` is the matrix multiplication $\mathbf{X}^T v$.

  Also, in the code, the objective function is the MSE, which has the additional factor $1/m$ for the gradient.



Using Batch Gradient Descent

- You may need to adjust learning rate α (step size):

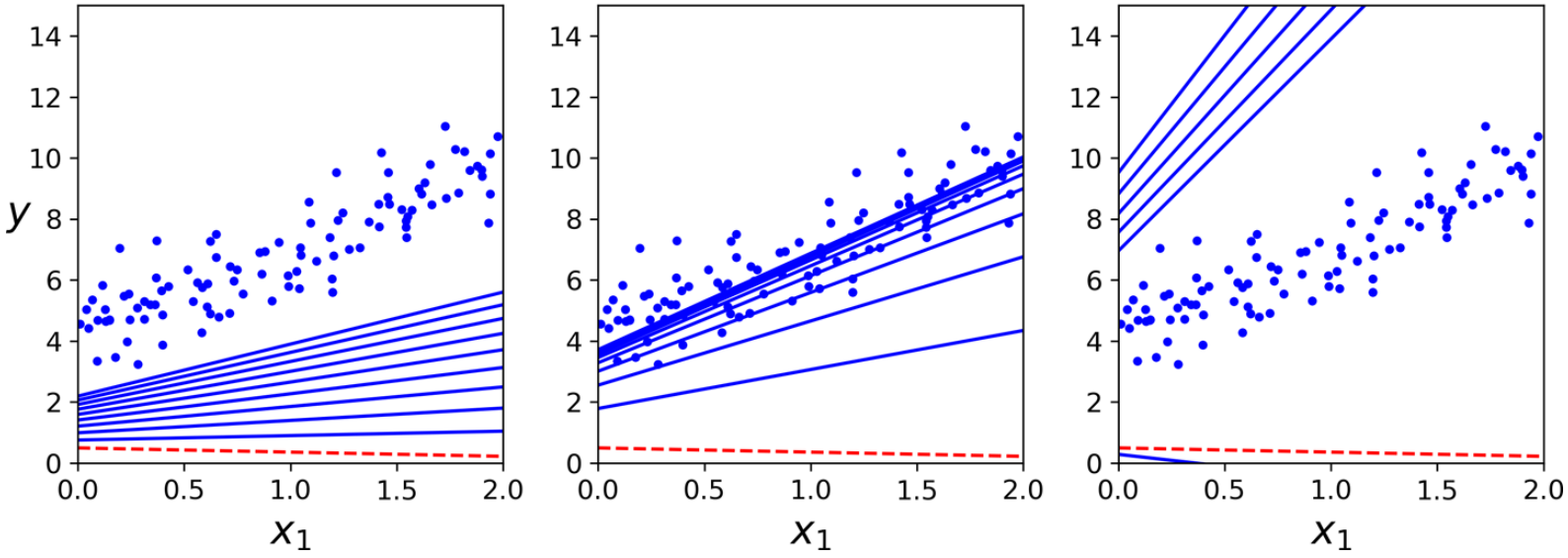


Figure 10: First 10 steps of batch gradient descent. Left: $\alpha = 0.02$. Middle: $\alpha = 0.1$. Right: $\alpha = 0.5$.

- How to set the number of iterations? Recall that the minimum x^*

must satisfy $\nabla f(x^*) = 0$. A simple solution is to set a very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny - that is, when its norm $\|\nabla f(x)\| \leq \epsilon$, where ϵ (called the tolerance) is a chosen small positive number.

Stochastic Gradient Descent

- Stochastic gradient descent picks a random instance in the training set at every step and computes an approximation of the gradient based only on that single instance.

  If we use only one instance in (15), say instance i, we get approximations of the partial derivatives:

$$\frac{\partial MSE(\theta)}{\partial \theta_k} \approx -2(y_i - (\theta_0 + \sum_{j=1}^{n} x_{ij}\theta_j))x_{ik}$$
$$= 2(\theta^T x_i - y_i)x_{ik},$$

  where x_i denotes the feature vector of instance i,
  $x_i = (x_{i0}, x_{i1}, \ldots, x_{in})^T$.

  (Recall $x_{i0} = 1$. The above formula applies for $k = 0, 1, \ldots, n$.)

The approximation of the gradient is

$$\nabla MSE(\theta) \approx 2(\theta^T x_i - y_i)x_i.$$

- Stochastic gradient descent makes it possible to train on huge training sets.
- The cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum. Once it gets there it will continue to bounce around, never settling down.
- Randomness is good for escaping from local optima (which does not happen in linear regression but may happen for other classes of objective functions), but bad because it means that the algorithm can never settle at the minimum.
- One solution to this dilemma is to gradually reduce the learning rate. The step sizes start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the

algorithm to settle at the global minimum.

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0/(t+t1)

theta = np.random.randn(2,1) # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

- We iterate by rounds of m iterations (here $m = 100$); each round is called an epoch.
- In each epoch and in each iteration during that epoch, we randomly draw an instance from the training set, and use that instance to

compute the approximate gradient.

This randomization is important!

Note that some instances may be picked several times per epoch, while others may not be picked at all. This is usually fine.

- The learning schedule gives a sequence of learning rates (step sizes) that decay according to $\frac{5}{k+50}$, where k is the index of the iteration.
- While the batch gradient descent code iterated $1,000$ times through the whole training set of $m = 100$ instances, this code goes through the training set only 50 times and reaches a good solution.

Using Scikit-Learn's Stochastic Gradient Descent

- You can use the `SGDRegressor` class, which defaults to optimizing the squared error cost function.

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

```
▸ SGDRegressor
```

```
sgd_reg.intercept_, sgd_reg.coef_
```

```
(array([3.67012906]), array([3.05335106]))
```

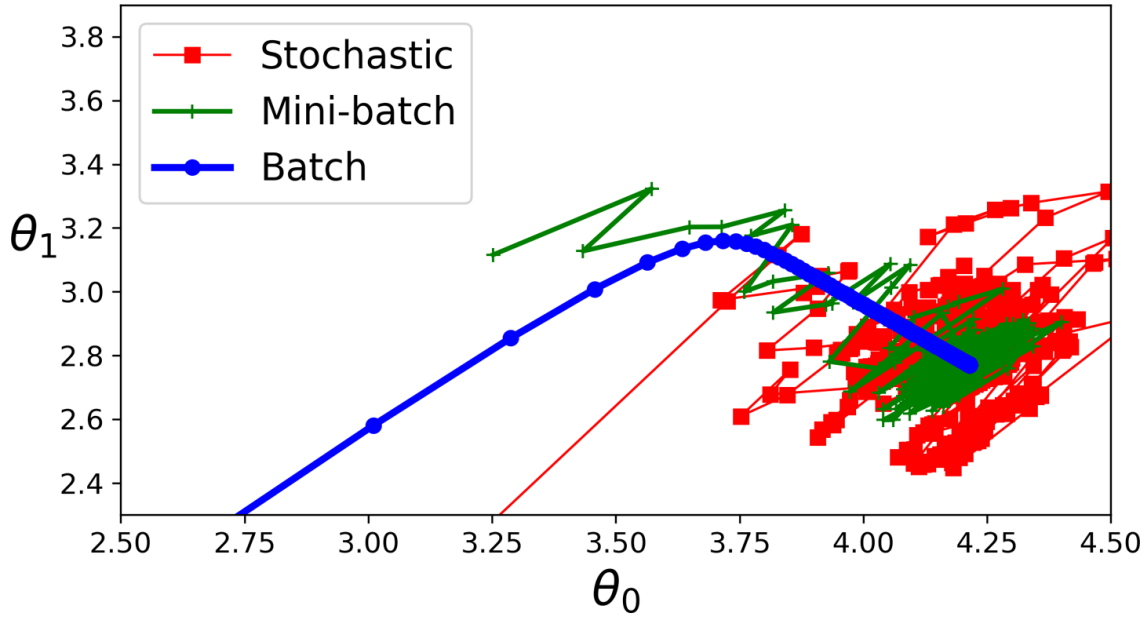
- `y.ravel()`: convert the 2-d array `y` with shape $(100, 1)$ to a flattened 1-d array with shape $(100,)$.
- The code runs for maximum $1,000$ epochs or until the loss drops by less than 0.001 during one epoch (`max_iter=1000, tol=1e-3`).

`tol`$= 1e-3$ means stop when `loss > best_loss - tol`.

- It starts with a learning rate of 0.1 (`eta0=0.1`), using the default learning schedule.
- `learning_ratestr`: the learning rate schedule; default=`'invscaling'`.
  - Options are: `'constant'`, `'optimal'`, `'invscaling'`, `'adaptive'`
  - `'optimal'` is similar to the learning schedule in our sample code.
  - `'invscaling'`: `eta = eta0 / pow(t, power_t)`, where
  - `power_t`: another hyperparameter; default is 0.5 (probably). Therefore, `eta = eta0`$/\sqrt{t}$.
- Lastly, it does not use any regularization (`penalty=None`; more details on this shortly).

Mini-Batch Gradient Descent

- At each step, compute the gradients on a small random subset of the instances called a mini-batch.
- The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.



- The algorithm's progress in the parameter space is less erratic than with Stochastic GD, especially with fairly large mini-batches.

  As a result, Mini-batch GD will end up walking around a bit closer to the minimum than Stochastic GD - but it may be harder for it to escape from local minima (in the case that the objective function has local minima).

Comparison of Algorithms for Linear Regression

- m: number of instances. n: number of features.

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

- Normal Equation: This refers to solving the normal equation by directing inverting the matrix.
- Sciket-Learn's `LinearRegression` also solves the normal equation, but does so through by SVD.
- Out-of-core algorithms are algorithms that are designed to process data that are too large to fit into a computer's main memory at once.

Polynomial Regression

- Surprisingly, you can 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.**
- Generate 100 training data points.

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

- Add the square of x-value as a new feature.

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X) # add the square of each feature
```

```
X[0]
```

```
array([-2.62549046])
```

```
X_poly[0]
```

```
array([-2.62549046,  6.89320014])
```

- Train a linear regression model.

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

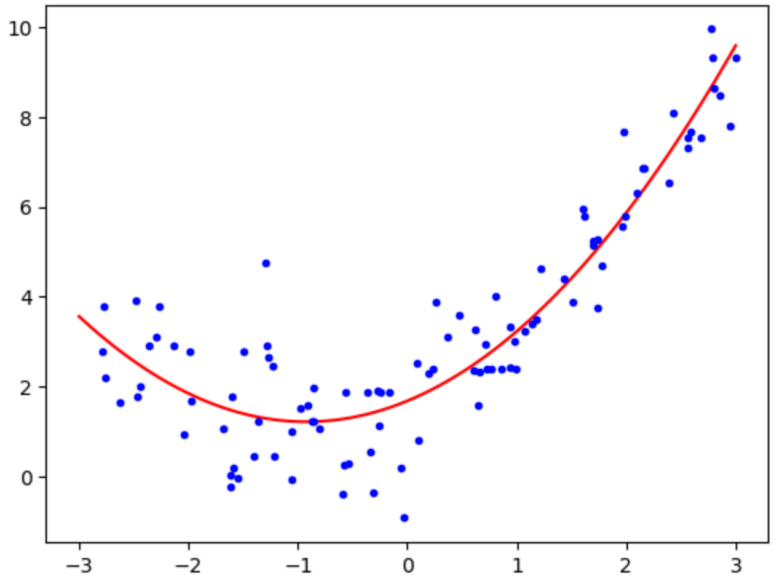
```
(array([1.68940759]), array([[1.00244928, 0.54274841]]))
```

- See the result.

```
X_newlin=np.linspace(-3,3,100)
X_newsq=np.square(X_newlin)
X_new_combined=np.transpose([X_newlin, X_newsq])
```

```
y_pred=lin_reg.predict(X_new_combined)
```

```
plt.plot(X_newlin, y_pred, "r-")
plt.plot(X, y, "b.")
plt.show()
```

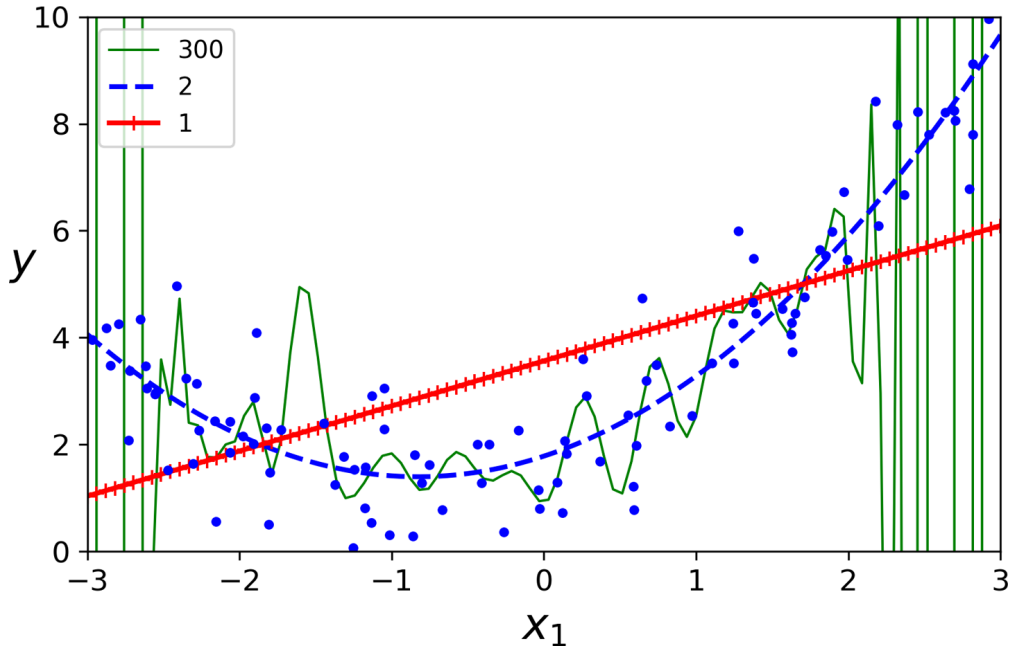


- When there are multiple features, polynomial regression is capable of finding relationships between features (which is something a plain linear regression model cannot do).

  This is possible because `PolynomialFeatures` also adds all combinations of features up to the given degree. For example, if there were two features a and b, `PolynomialFeatures` with degree=3 would not only add the features a^2, a^3, b^2, and b^3, but also the combinations ab, a^2b, and ab^2.

Learning Curves

- Compare polynomial models with degree 300, 2 and 1.



- The linear model (with degree 1) is underfitting. The degree-300 polynomial is overfitting.

For the degree-300 polynomial, see at the boundary of the x-interval how the output value becomes drastically different if the input value changes just a bit. This will show up as high variance in prediction.

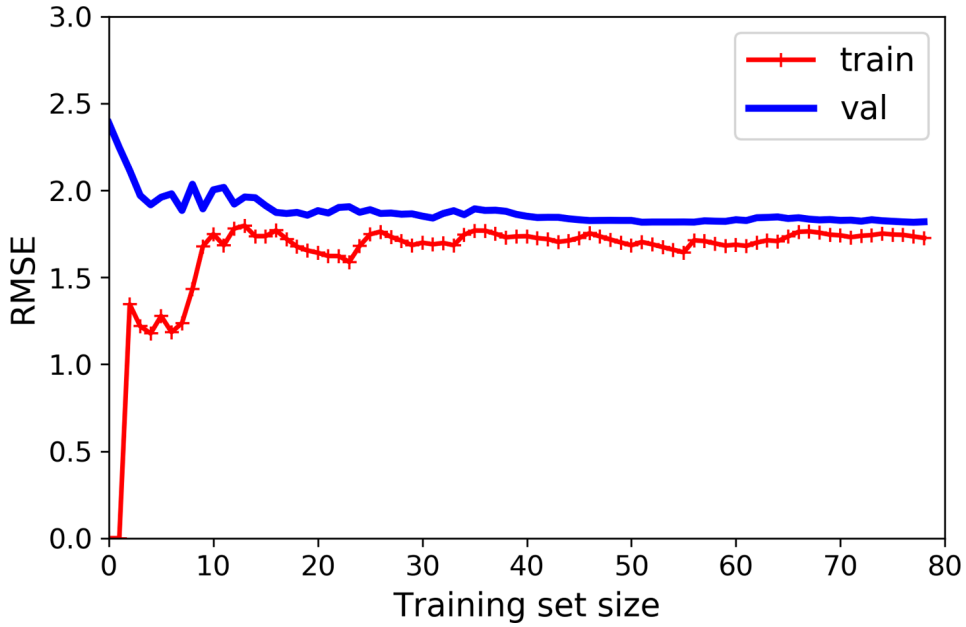
- We know our data are generated by a degree-2 polynomial.
- Earlier, we used cross-validation to get an estimate of a model's generalization performance.
  - If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then the model is overfitting.
  - If it performs poorly on both, then it is underfitting.
- Another way to tell is to look at the **learning curves**: these are 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 learning curves, train the model several times on different sized subsets of the training set.

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

- Learning curves for the linear regression model.

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```



– Characteristics of underfitting: Both errors flatten at high levels, which are close in values. At that point, adding new instances to the training set doesn't make the average error much better or worse.

  The function values are in the range $[-2, 10]$ for all the instances.

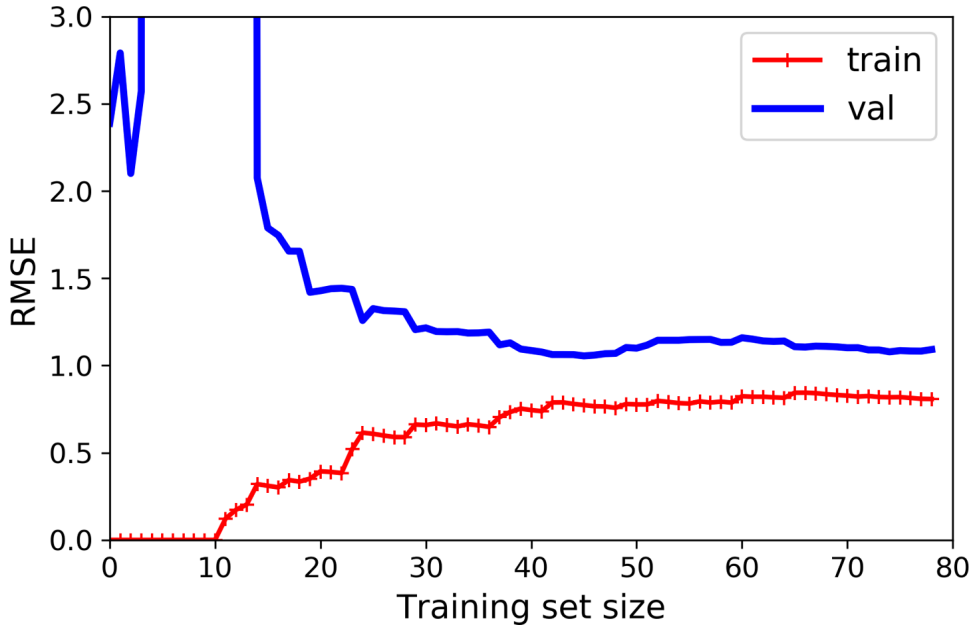
RMSE approaches 1.8, which is high.

- Next, a 10th-degree polynomial model.

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
```



- The error on the training data is much lower than with the linear regression model.
- There is a gap between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of overfitting. If you used a much larger training set, however, the two curves would continue to get closer.

  Why? As more and more training data are used, even a complex model eventually will underfit. We should see the training error and validation error come close. However, the magnitude of the errors is smaller than a less complex model.
- One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

Bias-Variance Tradeoff

- Special Case: $Y = f(X) + \varepsilon$, where ε is an independent noise with 0 mean and variance σ^2.
- Let $\hat{f}$ denote the estimator (i.e., trained model).
- For $X = x_0$, let the true output be denoted by Y_0, which is random. In the current case, $Y_0 = f(x_0) + \varepsilon$. The predicted output is $\hat{f}(x_0)$, which is also random because the training data are drawn randomly.
- The bias is: $E[Y_0 - \hat{f}(x_0)|X = x_0] = f(x_0) - E_{\mathcal{T}}[\hat{f}(x_0)]$, where $\mathcal{T}$ is the distribution from which all the training data are drawn.
- We showed: The prediction error is the sum of an irreducible component σ^2, the squared bias and the variance of the estimate.

Specifically,

$$\begin{aligned} Err(x_0) =& E[(Y_0 - \hat{f}(x_0))^2 | X = x_0] \\ =& \sigma^2 + (f(x_0) - E\hat{f}(x_0))^2 + E[(E\hat{f}(x_0) - \hat{f}(x_0))^2] \\ =& \sigma^2 + \text{Bias}^2(\hat{f}(x_0)) + \text{Var}(\hat{f}(x_0)). \end{aligned}$$

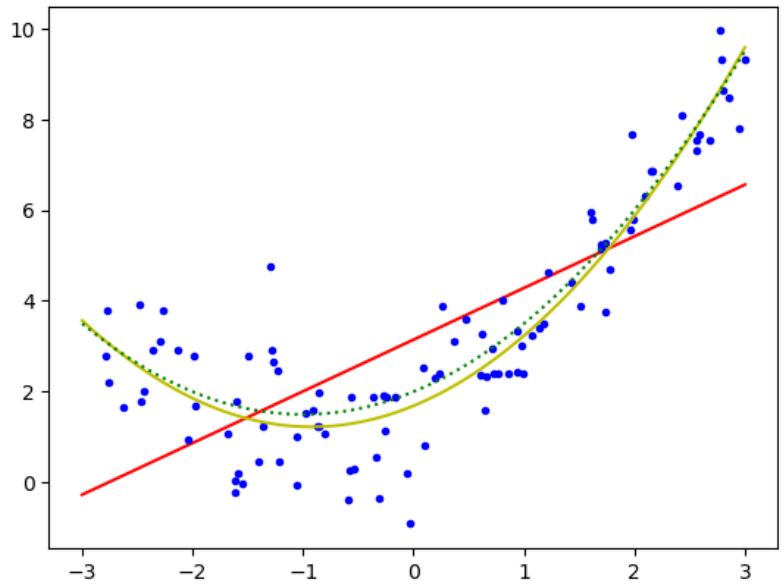


Figure 11: Linear versus quadratic models. True $f(x)$ is $f(x) = 0.5x^2 + x + 2$ (dotted line).

- The above expression of the prediction error is for a single fixed input x_0. We may get lucky that the bias is small for a particular x_0 even if the model underfits. See the linear model in the figure and imagine x_0 is near where the linear and quadratic functions intercept.

However, if x_0 is elsewhere, the bias of the linear model can be large.

- In computing the prediction error, we should let x_0 vary. Let X_0 be a random input. The expected prediction error is

$$E[Err(X_0)] = \sigma^2 + E[\text{Bias}^2(\hat{f}(X_0))] + E[\text{Var}(\hat{f}(X_0))].$$

- There is often a trade-off between the bias and the variance in different estimators/models.
- *Bias:* This part of the generalization error is due to wrong assumptions, such as assuming the input-output relationship is linear when it is actually quadratic (f is quadratic; $\hat{f}$ is linear). A high-bias model is likely to be associated with underfitting the training data.

  When a model underfits (e.g., the linear model in the figure), the expected bias is relatively large. However, such a model is relatively stable for different draws of the training data, which implies low variance.

- *Variance:* This part measures how sensitive the model is with respect to different draws of the training data.

  A model with many degrees of freedom (such as a high-degree polynomial model) adapts well to a given set of training data. When another set of training data is drawn, the learned parameters of the model may change by a lot to adapt to the new data. This means overfitting tends to be associated with high variance, and more complex models tend to have high variance.

- *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 the data sources, such as broken sensors, or detect and remove outliers).

- Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, 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 so as to reduce the degrees of freedom).
- For a linear model, regularization is typically achieved by constraining the weights of the model.

  Recall that a linear model can be complex too if it has a large number of features or you add higher-degree terms for the features.

Ridge Regression (Tikhonov Regularization)

- Recall that our original objective is to minimize $MSE(\theta)$ (which is equivalent to minimizing $RSS(\theta)$; we will now use $MSE(\theta)$ as the objective function because of the code).
- Modify the cost function to:

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

- The bias term θ_0 is usually not regularized.
- The hyperparameter $\alpha \geq 0$ controls how much you want to regularize the model. When α is large, each $|\theta_i|$ must be small.
- Let the vector $\mathbf{w} = (\theta_1, \ldots, \theta_n)^T$. The regularization term is $\alpha/2 \left\|\mathbf{w}\right\|^2$ (the norm is the 2-norm or l_2 norm).

- Gradient descent can be used to solve $\min_\theta J(\theta)$. The gradient is $\nabla J(\theta) = -\frac{2}{N}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta) + \frac{\alpha}{2}\begin{pmatrix} 0 \\ \mathbf{w} \end{pmatrix}$.
- It is important to scale the data (e.g., using a `StandardScaler`) before performing ridge regression.
- Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to use the unregularized performance measure to evaluate the model's performance.
- Ridge regularization can also be combined with polynomial regression. The data is first expanded using `PolynomialFeatures`, then it is scaled using a `StandardScaler`, and finally the (linear) Ridge regression is applied to the resulting features.

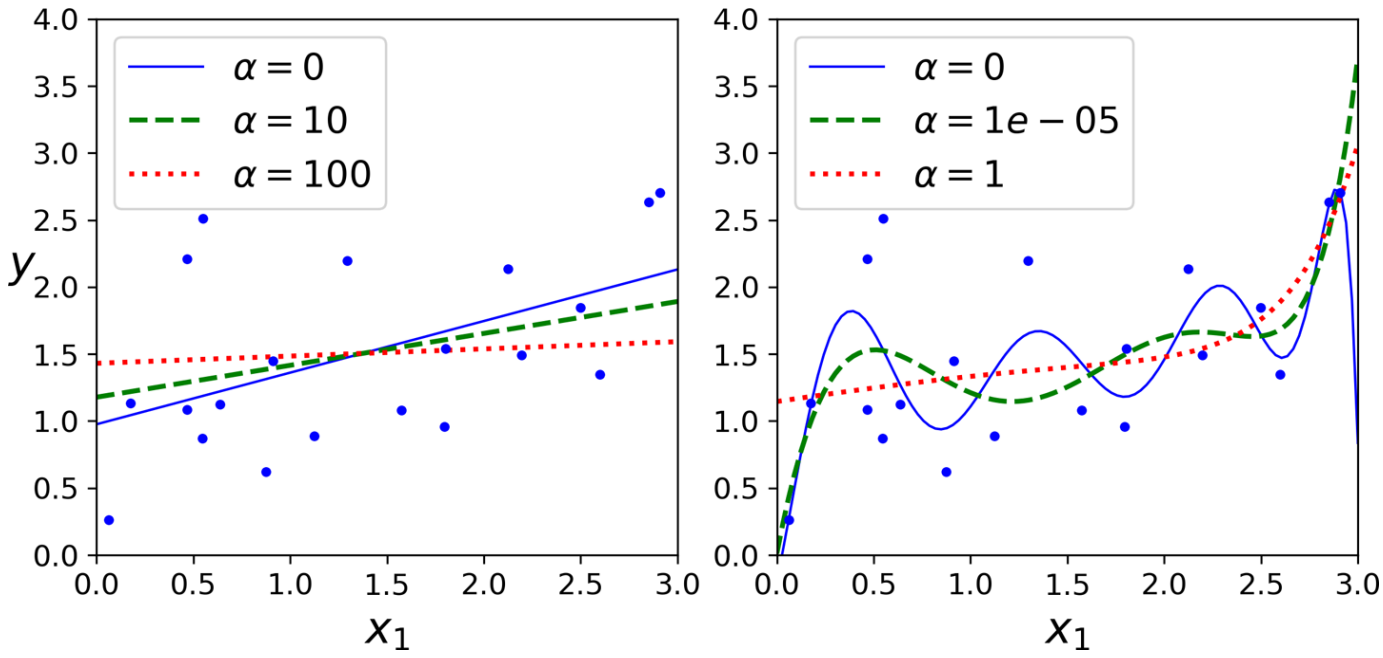


Figure 12: Ridge regression. Left: linear model; right: polynomial of degree 10.

- Note how increasing α leads to flatter predictions with less wiggles, thus reducing the model's variance but increasing its bias.

  Why? The regularization in Ridge regression constrains the

magnitude of each θ_i. A small change in the values of a feature won't lead to drastic change in the output. Also, the effects of different features on the output are made comparable. For a high-degree polynomial model, this means that the high-degree terms (features) don't have much more influence on the output than the low-degree terms.

- How to use it? Option 1: Close-form solution by solving linear equations (based on Cholesky decomposition).

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky")
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

```
array([[4.84621005]])
```

- Using stochastic gradient descent:

```
sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])
```

```
array([4.82246691])
```

The `penalty` hyperparameter sets the type of regularization term to use. Specifying `l2` indicates that you want SGD to add a regularization term to the cost function equal to half the square of the l_2 norm of the weight vector: this is simply Ridge regression.

Lasso Regression

- Lasso stands for *Least Absolute Shrinkage and Selection Operator Regression*.
- Modify the cost function to:

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

  In other words, the regularization term uses the l_1 norm instead of the l_2 norm.

- An important characteristic of Lasso Regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero).

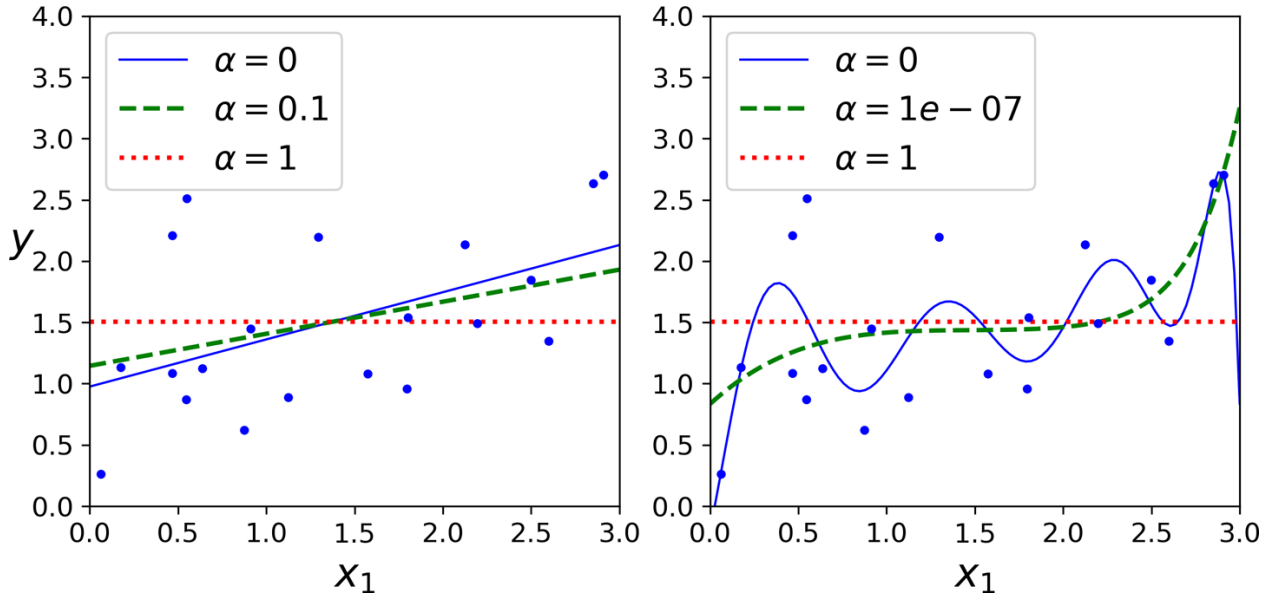


Figure 13: Lasso regression. Left: linear model; right: polynomial of degree 10.

- For example, the dashed line in the righthand plot in Fig. 13 (with $\alpha = 10^{-7}$) doesn't wiggle much, almost linear: all the weights for the high-degree polynomial features are equal to zero.

- In other words, Lasso Regression automatically performs feature selection and outputs a sparse model (i.e., with few nonzero feature weights).
- Reason: When α is substantial, the regularization term matters. Since it is a piecewise linear function, when $\theta_i \neq 0$, the partial derivative wrt to θ_i is a constant, equal to $\pm\alpha$. The gradient descent tends to quickly bring some non-zero θ_i to zero.
- Reason by geometry: The minimization problem in Lasso regression is the Lagrangian relaxation of a constrained minimization problem, which was the original formulation.

$$\min MSE(\theta) \quad \text{subject to } \sum_{i=1}^{n} |\theta_i| \leq t.$$

Note: The intercept θ_0 is not in the constraint.

For 2-D input features, the constraint is $|\theta_1| + |\theta_2| \leq t$; figure (a) below shows the constraint set.

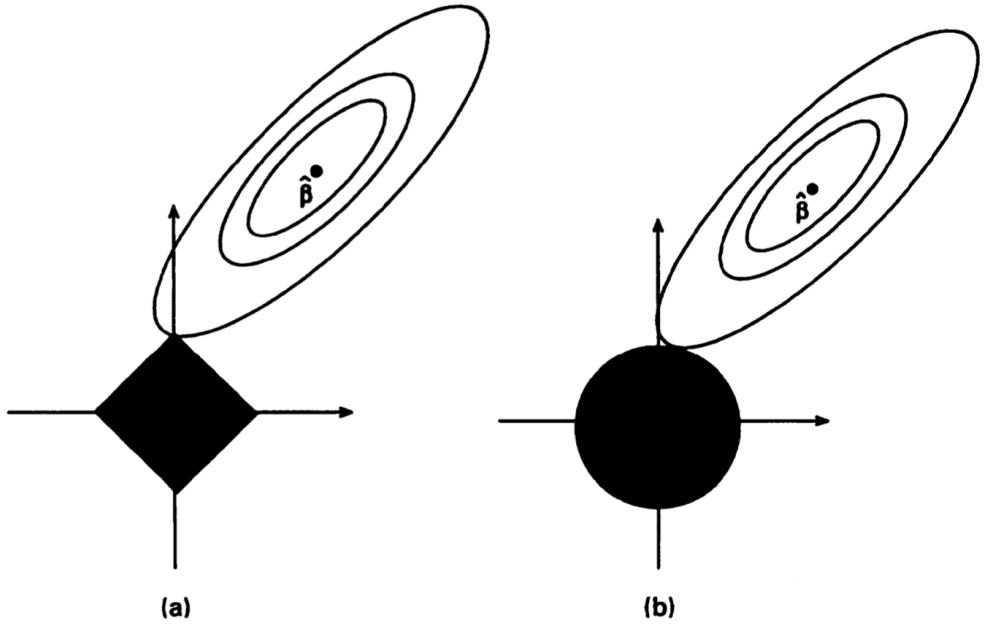


Figure 14: Geometry of Lasso and ridge Regression (Tibshirani 96). (a) Lasso; (b) Ridge. The dark regions are the constraint sets. $\hat{\beta}$ is the unconstrained minimum.

We see that since the corners extend out, they are likely to be the minimum solutions for Lasso. The corner points have the property that some components of θ are zero.

For Ridge, the constraint set is round. It is unlikely the level set

touches a point in the constraint region for which many components are zero.

Imagine you start at the unconstrained minimum $\hat{\beta}$ for $MSE(\theta)$, and consider the level sets for $MSE(\theta)$ in increasing order of their values c. When the value c is just large enough so that it is the first time any level set touches the constraint region, the point where they touch is the constrained minimum.

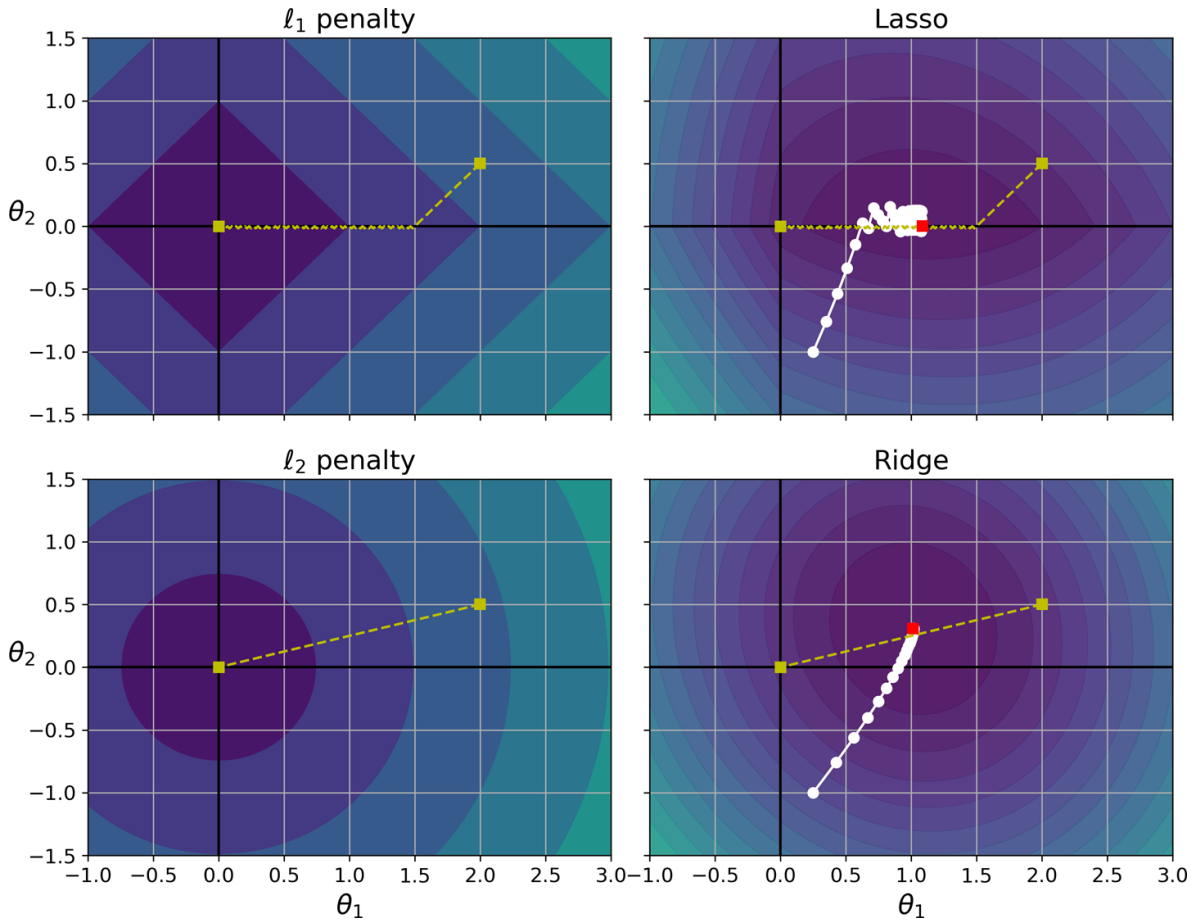


Figure 15: Comparing Lasso and Ridge regression. Left: minimizing the regulation term only; right: minimizing $J(\theta)$. See how the trajectory hits $\theta_2 = 0$ in Lasso regression.

- Disadvantage of Lasso: It can happen that the learned parameters bounce around the optimum at the end. Why? Suppose the regulation term is substantial and the minimum for $J(\theta)$ is at a point θ^* where $\theta_i^* = 0$ for some i. Not all the partial derivatives of $J(\theta)$ exist at such a point (hence, we do not have $\nabla J(\theta^*) = 0$ at the minimum).

- See the 1-d example in Fig. 16, where

$$J(\theta) = (\theta - 1)^2 + 3|\theta|.$$

  Here, $3|\theta|$ corresponds to the regulation term and $\alpha = 3$. It is easy to show the minimum is $\theta^* = 0$ and the minimum function value is 1.

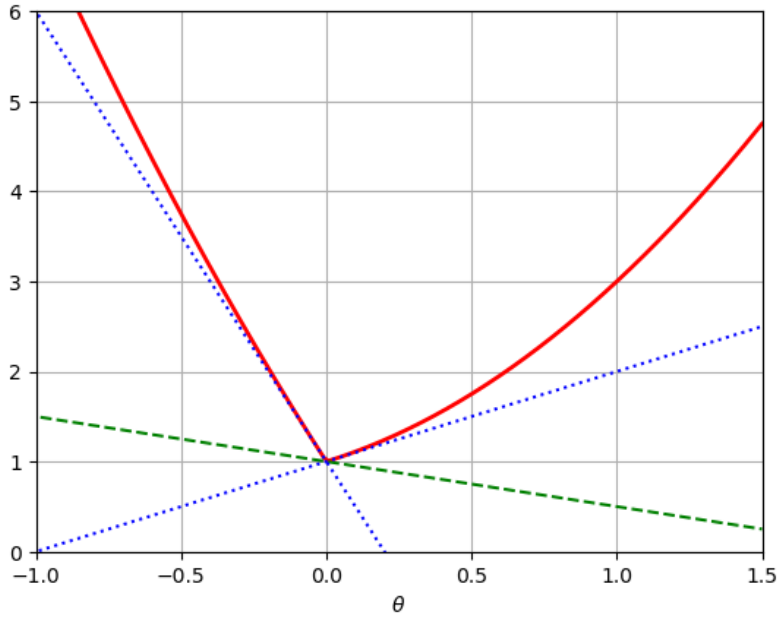


Figure 16: The solid line is $J(\theta)$. The slopes of the two blue dotted lines are the left and right derivatives at $\theta = 0$, which are -5 and 1, respectively. Any value in $[-5, 1]$ is a subgradient at 0, such as the slope of the green dashed line.

- At $\theta = 0$, the left and right derivatives are not equal, which means the function is not differentiable at 0 (no gradient at 0).
- Both the left and right derivatives are substantial in magnitude (5

and 1, respectively): they are nowhere near zero. This implies that when we use gradient descent and approach 0 from either the left or the right, the gradient has non-trivial magnitude. As a result, the algorithm tends to jump across $\theta = 0$ back and forth.

In contrast, for a function that is continuously differentiable at the minimum θ^* (e.g., as in Ridge regression), the derivative is equal to 0 at θ^*; furthermore, the magnitude of the gradient at θ is near zero when θ is close to θ^*. This is essential for the gradient descent algorithm to converge to θ^*.

- While a function may not have a gradient at some θ, a convex function always has a subgradient everywhere.

  Given a convex function $f : \mathbb{R}^n \to \mathbb{R}$, we say a vector $d \in \mathbb{R}^n$ is a **subgradient** of f at a point $x_0 \in \mathbb{R}^n$ if

  $$f(x) \geq f(x_0) + d^T(x - x_0), \quad \forall x \in \mathbb{R}^n.$$

  The right hand side is a linear function in x, and it is equal to $f(x_0)$

at $x = x_0$. The expression means that this linear function is a global under-estimator of $f(x)$, and it intercepts function graph of f at $x = x_0$.

- Recall that if a convex function f is differentiable at x_0, the gradient of $\nabla f(x_0)$ satisfies

$$f(x) \geq f(x_0) + \nabla f(x_0)^T (x - x_0), \quad \forall x \in \mathbb{R}^n.$$

  In this case, $\nabla f(x_0)$ is the only subgradient of f at x_0.

- We see that, in the non-differentiable case, the subgradient d takes the role of the gradient.

- Due to the regularization term, our optimization objective $J(\theta)$ is not everywhere differentiable. The gradient descent algorithm needs to be modified suitably, using a subgradient in place of the gradient when it is at a non-differentiable point. This is known as a **subgradient algorithm.**

At any $\tilde{\theta} \in \mathbb{R}^n$ with $\tilde{\theta}_i = 0$ for some i, the left and right partial derivatives w.r.t. θ_i are not equal, which implies that the partial derivative w.r.t. θ_i does not exist at $\tilde{\theta}$, and hence, the gradient does not exist at $\tilde{\theta}$.

- At such a $\tilde{\theta}$, one can use the following subgradient of J, denoted by g_J, in place of the role of the gradient in the algorithm.

$$g_J(\tilde{\theta}) = \nabla MSE(\tilde{\theta}) + \alpha \begin{pmatrix} \text{sign}(\tilde{\theta}_1) \\ \vdots \\ \text{sign}(\tilde{\theta}_n) \end{pmatrix},$$

where

$$\text{sign}(\tilde{\theta}_i) = \begin{cases} -1 & \text{if } \tilde{\theta}_i < 0 \\ 0 & \text{if } \tilde{\theta}_i = 0 \\ 1 & \text{if } \tilde{\theta}_i > 0. \end{cases}$$

- To help Lasso regression to converge, you need to gradually reduce the learning rate during training.
- Here is a small Scikit-Learn example using the Lasso class:

```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
lasso_reg.predict([[1.5]])
```

- Or, you could instead use `SGDRegressor(penalty="l1")`.

Elastic Net Regularization

- The Elastic Net cost function is:

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

- Elastic Net is a middle ground between Ridge regression and Lasso regression. You control the mix ratio through r.
- It is almost always preferable to have at least a little bit of regularization, so generally you should avoid plain linear regression.
- Ridge is a good default, but if you suspect that only a few features are useful, you should prefer Lasso or Elastic Net because they tend to reduce the useless features' weights down to zero.
- In general, Elastic Net is preferred over Lasso because Lasso may behave erratically when the number of features is greater than the

number of training instances or when several features are strongly correlated.

- Here is a short example that uses Scikit-Learn's `ElasticNet`.

```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X, y)
elastic_net.predict([[1.5]])
```

`l1_ratio` corresponds to the mix ratio r.

Early Stopping

- Here is a very different way to regularize iterative learning algorithms such as gradient descent: Stop training as soon as the validation error reaches a minimum.

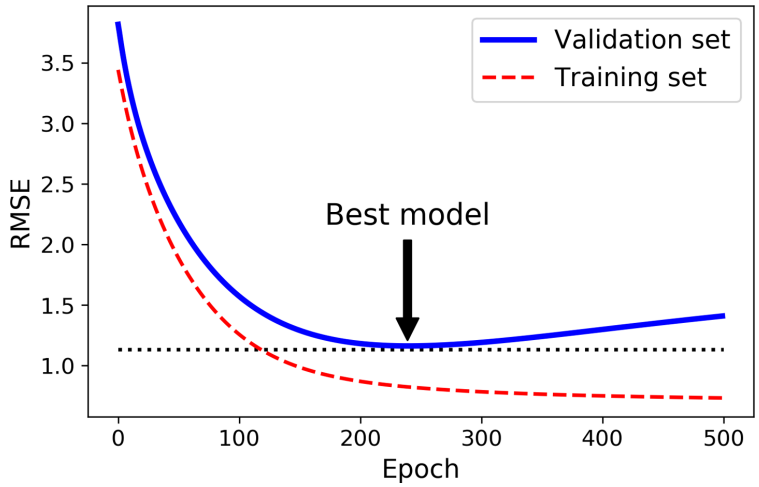


Figure 17: Batch gradient descent for a high-degree polynomial regression model

- After some iterations, if the validation error stops decreasing and starts to go back up, this indicates that the model has started to overfit the training data.
- With stochastic and mini-batch gradient descent, the curves for errors are not so smooth, and it may be hard to know whether you have reached the minimum or not.

  One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum.
- Example:

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X, y.ravel(), test_size=0.5)
```

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from copy import deepcopy

poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler())
    ])

X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005, random_state=42)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = deepcopy(sgd_reg) # use deepcopy; clone doesn't copy trained data
```

With `warm_start=True`, when the `fit()` method is called, it continues training where it left off, instead of restarting from scratch.

`tol = -np.infty`: Since `loss > best_loss - tol` is always false, this stopping criteria never activates. But, it is useful for suppressing many warning messages.

Logistic Regression (aka Logit Regression)

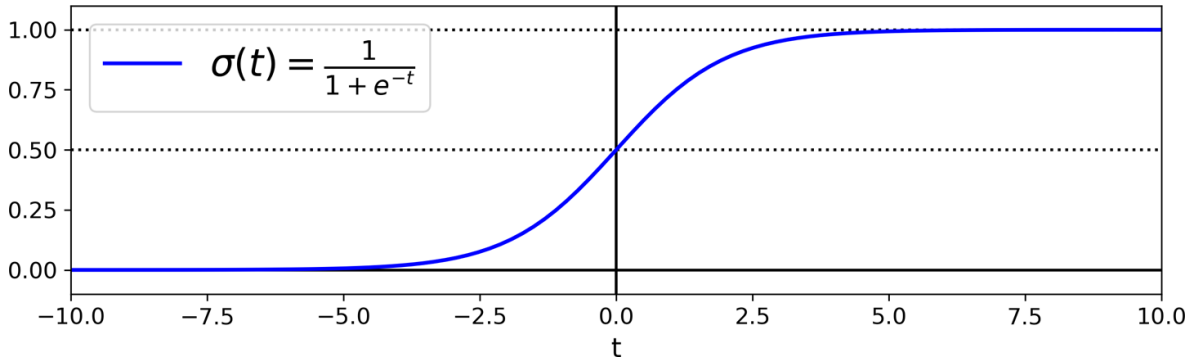
Logistic Regression

- Commonly used to estimate the probability that an instance belongs to a particular class.
- If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled "1"); otherwise, the model predicts that the instance belongs to the negative class, labeled "0". This makes it a binary classifier.
- Just like a linear regression model, a logistic regression model computes a weighted sum of the input features x (which also has a bias term), but instead of outputting the result directly like the linear regression model does, it outputs the logistic of this result:

$$p = h_\theta(x) = \sigma(x^T\theta),$$

where the logistic function $\sigma : \mathbb{R} \to (0, 1)$ is a sigmoid function:

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



- By assumption, the value $h_\theta(x) = \sigma(x^T\theta)$ is understood as the probability that an instance with the feature vector x **belongs to the positive class**.
- Once the logistic regression model has estimated (θ computed based on the training data), the predicted probability that an instance with the feature vector x belongs to the positive class is $\hat{p} = h_{\hat{\theta}}(x)$, where

$\hat{\theta}$ is the vector containing the estimated parameters. The classification of x or the predicted class for x, denoted by $\hat{y}$, is

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

- Since $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, a logistic regression model predicts:

$$\hat{y} = \begin{cases} 1 & \text{if } x^T \hat{\theta} \geq 0 \\ 0 & \text{if } x^T \hat{\theta} < 0. \end{cases}$$

The decision boundary is linear, i.e., the hyperplane $x^T \hat{\theta} = 0$.

- From $p = \sigma(t)$, where p is a probability, we get

$$t = \log \frac{p}{1-p}.$$

t is the log of the ratio of the probability for the positive class and the probability for the negative class, known as the **log odds**.

Training and Cost Function

- We are given the training data $(x_1, y_1), \ldots, (x_m, y_m)$, where each x_i is a feature vector and each y_i is a binary value.
- We assume the $\{(x_i, y_i)\}$ are drawn in an IID fashion from the underlying distribution.
- The probability of observing $(y_1, \ldots, y_m)$ given the inputs are $x_1, \ldots, x_m$ is

$$
\begin{aligned}
L(\theta) \triangleq & P(Y_1 = y_1, \ldots, Y_m = y_m | X_1 = x_1, \ldots, X_m = x_m; \theta) \\
&= \prod_{i=1}^{m} P(Y_i = y_i | X_i = x_i; \theta) \\
&= \prod_{i=1}^{m} \sigma(x_i^T \theta)^{y_i} \left(1 - \sigma(x_i^T \theta)\right)^{1-y_i} \\
&= \prod_{i=1}^{m} \sigma(x_i^T \theta)^{y_i} \prod_{i=1}^{N} \left(1 - \sigma(x_i^T \theta)\right)^{1-y_i}
\end{aligned}
$$

Often-used trick: For each i, the two cases $y_i = 1$ and $y_i = 0$ are combined into a single expression $\sigma(x_i^T\theta)^{y_i}\left(1-\sigma(x_i^T\theta)\right)^{1-y_i}$. When $y_i = 1$, the second factor is equal to 1; when $y_i = 0$, the first factor is equal to 1.

- $L(\theta)$ is known as the **likelihood function**. The training objective is to maximize $L(\theta)$, which is the same as maximizing the **log-likelihood** function $\log L(\theta)$, which is the same as minimizing the cost function $J(\theta) \triangleq -\frac{1}{m}\log L(\theta)$.

$$\begin{aligned} J(\theta) &= -\frac{1}{m}\log L(\theta) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\Big(y_i \log\sigma(x_i^T\theta) + (1-y_i)\log(1-\sigma(x_i^T\theta))\Big). \end{aligned} \tag{16}$$

  - Let

$$t_i = \begin{cases} 1 & \text{if } y_i = 1 \\ -1 & \text{if } y_i = 0 \end{cases}$$

    That is, t_i is a re-encoding of y_i.

– When $y_i = 1$, $(1 - y_i) \log(1 - \sigma(x_i^T \theta)) = 0$ and

$$y_i \log \sigma(x_i^T \theta) = \log \frac{1}{1 + \exp(-x_i^T \theta)} = -\log(1 + \exp(-x_i^T \theta))$$
$$= -\log(1 + \exp(-t_i x_i^T \theta)).$$

When $y_i = 0$, $y_i \log \sigma(x_i^T \theta) = 0$ and

$$(1 - y_i) \log(1 - \sigma(x_i^T \theta)) = \log(1 - \frac{1}{1 + \exp(-x_i^T \theta)})$$
$$= \log(\frac{\exp(-x_i^T \theta)}{1 + \exp(-x_i^T \theta)})$$
$$= \log(\frac{1}{1 + \exp(x_i^T \theta)})$$
$$= -\log(1 + \exp(-t_i x_i^T \theta)).$$

- Therefore, $J(\theta)$ can be written as

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \log(1 + \exp(-t_i x_i^T \theta)).$$

- In Scikit-Learn's `SGDClassifier`, when we set the hyperparemeter `loss="log"`, we mean the above loss function and in that case the `SGDClassifier` will be trained with logistic regression.

- There is no known closed-form equation to compute the value of θ that minimizes $J(\theta)$. However, $J(\theta)$ is convex, so gradient descent (or other iterative optimization algorithms) can be applied to find the global minimum.

- The partial derivative of $J(\theta)$ w.r.t. θ_j is

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_j} &= -\frac{1}{m}\sum_{i=1}^{m}\left(y_i\frac{x_{ij}\sigma'(x_i^T\theta)}{\sigma(x_i^T\theta)} - (1-y_i)\frac{x_{ij}\sigma'(x_i^T\theta)}{1-\sigma(x_i^T\theta)}\right) \\
&= -\frac{1}{m}\sum_{i=1}^{m} x_{ij}\sigma'(x_i^T\theta)\left(\frac{y_i}{\sigma(x_i^T\theta)} - \frac{(1-y_i)}{1-\sigma(x_i^T\theta)}\right) \\
&= -\frac{1}{m}\sum_{i=1}^{m} x_{ij}\sigma'(x_i^T\theta)\left(\frac{y_i-\sigma(x_i^T\theta)}{\sigma(x_i^T\theta)(1-\sigma(x_i^T\theta))}\right) \\
&= -\frac{1}{m}\sum_{i=1}^{m} x_{ij}\frac{\exp(-(x_i^T\theta))}{(1+\exp(-(x_i^T\theta)))^2}\left(\frac{y_i-\sigma(x_i^T\theta)}{\sigma(x_i^T\theta)(1-\sigma(x_i^T\theta))}\right) \\
&= -\frac{1}{m}\sum_{i=1}^{m} x_{ij}\exp(-(x_i^T\theta))\sigma^2(x_i^T\theta)\left(\frac{y_i-\sigma(x_i^T\theta)}{\sigma(x_i^T\theta)(1-\sigma(x_i^T\theta))}\right) \\
&= -\frac{1}{m}\sum_{i=1}^{m} x_{ij}\exp(-(x_i^T\theta))\frac{\sigma(x_i^T\theta)}{1-\sigma(x_i^T\theta)}\Big(y_i-\sigma(x_i^T\theta)\Big) \\
&= \frac{1}{m}\sum_{i=1}^{m}\Big(\sigma(x_i^T\theta)-y_i\Big)x_{ij}. \qquad (17)
\end{aligned}$$

In the above, we used the following facts, which are easy to check:

$$\sigma'(t) = \frac{\exp(-t)}{(1+\exp(-t))^2},$$

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

The meaning of (17): $\left(\sigma(x_i^T\theta) - y_i\right) x_{ij}$ is the prediction error for instance i multiplied by the jth feature value of instance i. Then, such quantities are averaged over all the instances.

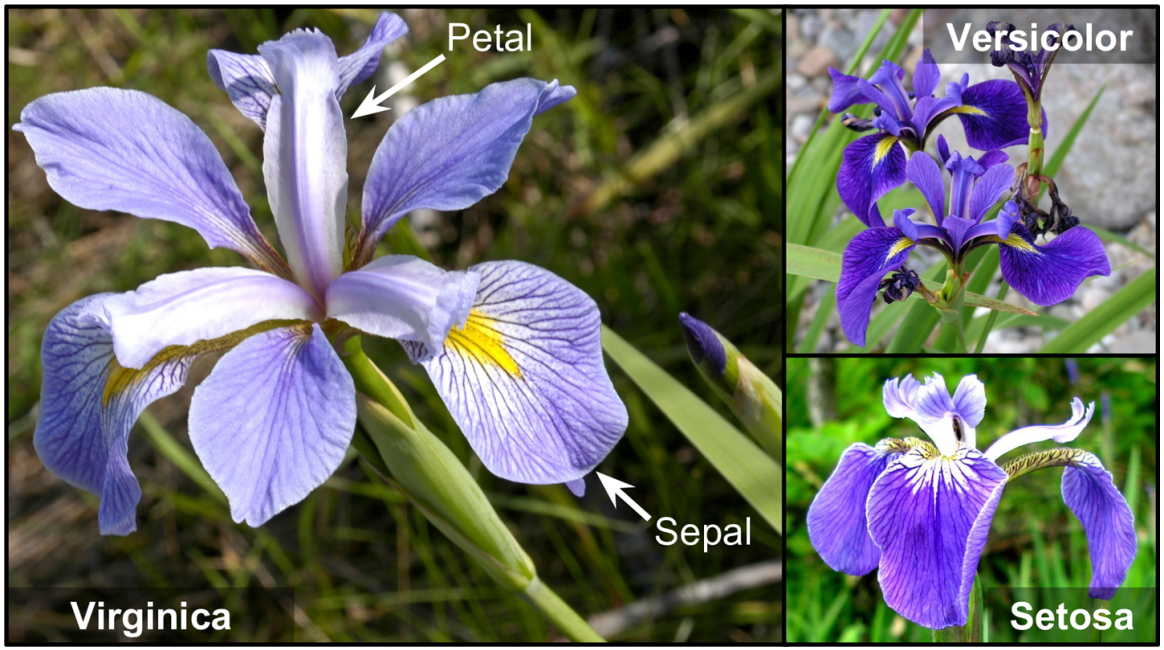
- The gradient of $J(\theta)$ is:

$$\nabla J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma(x_i^T\theta) - y_i\right) x_i.$$

- Once you have the gradient vector containing all the partial derivatives, you can use it in the batch gradient descent algorithm. You can also have stochastic GD by taking one instance at a time, or mini-batch GD by using a mini-batch of instances at a time.

Decision Boundaries

- We will use a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: Iris Setosa, Iris Versicolor, and Iris Virginica.



- Sepal is the outer part of the flower that provides protection for the

flower during its bud stage. They are green in most flowers (not for iris).

- Petals are the inner part, usually brightly colored, that surrounds the reproductive units of flowers.
- Load iris data, which is dictionary like

```
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
```

```
['data',
 'target',
 'frame',
 'target_names',
 'DESCR',
 'feature_names',
 'filename',
 'data_module']
```

- Input: 4 features per instance.
- Output value: from $\{0, 1, 2\}$ representing three classes

```
print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica
```

- We will first try 1-d (one input feature), binary logistic regression.

```
X = iris["data"][:, 3:] # petal width
y = (iris["target"] == 2).astype(int) # 1 if Iris virginica, else 0
```

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

`X` needs to be a 2D array to work with `fit()`. In the code, `[:,3:]` returns a 2D array. If you use `[:,3]`, you will get a 1D array, which won't work with `fit()`.

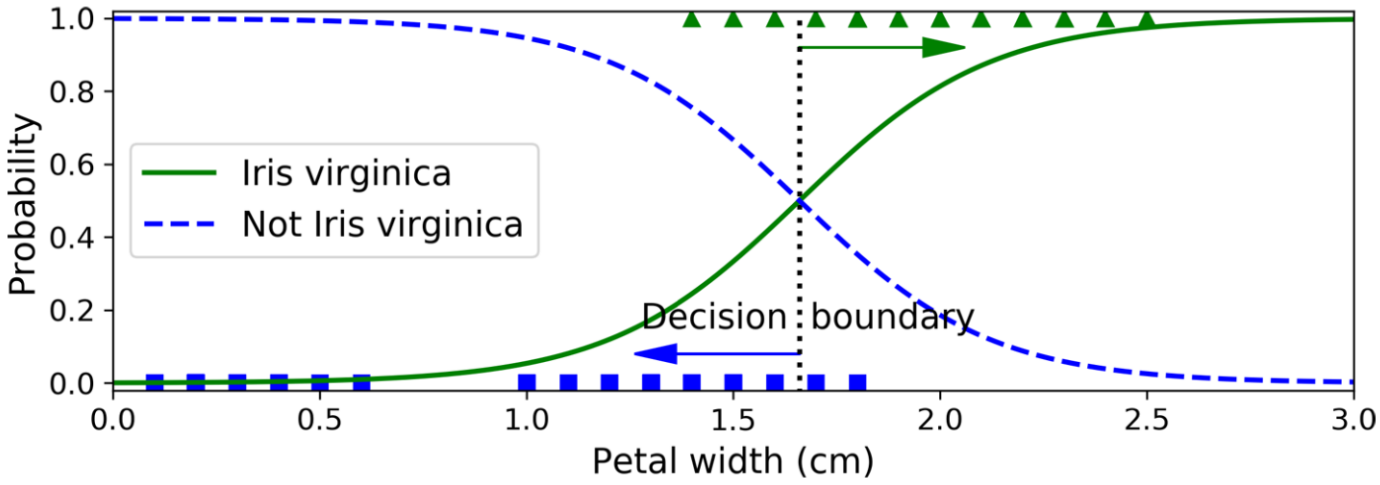
- Plot probabilities:

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris virginica")
plt.legend(loc="center left", fontsize=14)
```

`reshape(-1, 1)`: reshape the array into 1 column and as many rows as necessary

to accommodate the data. It results in a 2D array with a shape `(n, 1)`.

More matplotlib code to add more info to the figure:



The squares and triangles are actual instances in the dataset.

- The decision boundary is around 1.6 cm where the probability of being Iris Virginia is 0.5.
- For prediction:

```
log_reg.predict([[1.7], [1.5]])
```

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

Iris: 2-D Decision Boundaries

- Now, let us use two features: petal width and length. The output is still binary: 1 for Iris Virginica; 0 otherwise.

```
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(int)
```

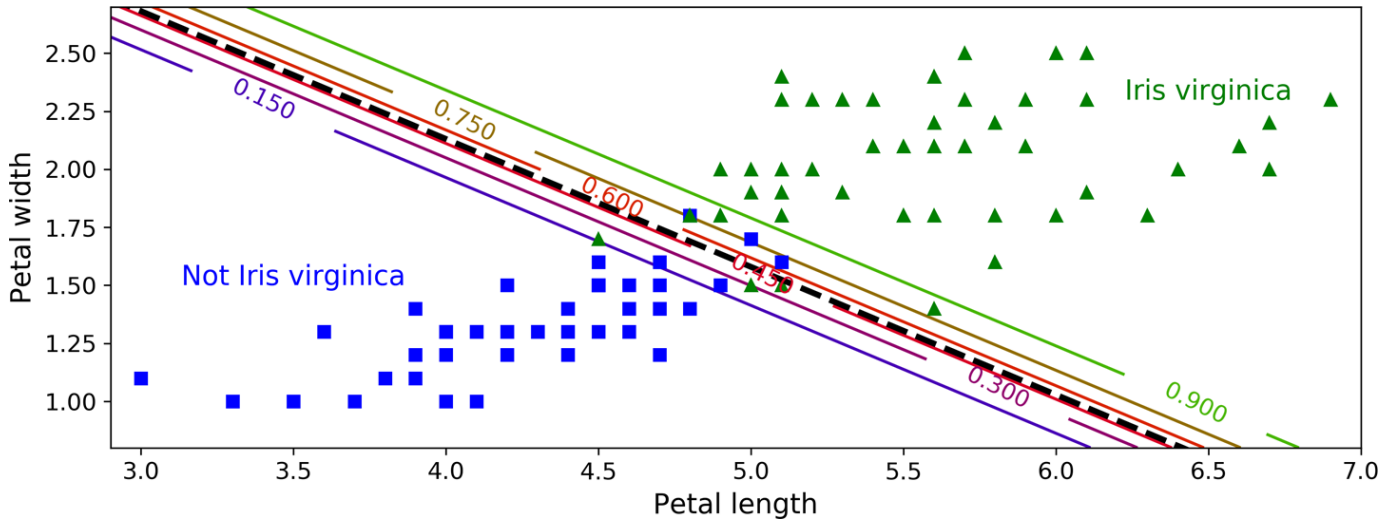
```
log_reg = LogisticRegression(solver="lbfgs", C=10**10, random_state=42)
log_reg.fit(X, y)
```

```
▸ LogisticRegression
```

```
log_reg.predict([[1.4, 0.2], [5.5, 2.5]])
```

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

- Show 2-d decision boundary using matplotlib; either plot the boundary line $\theta_0 + \theta_1 l + \theta_2 w = 0$ directly or use `contour()` (like plotting level curves). Here, l is the petal length; w is the petal width.



- The dashed line is the model's decision boundary; it represents the points where the model estimates a 50% probability.

  The decision boundary is linear.

  The squares and triangles are real data points.

- Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right).

- Just like the other linear models, logistic regression models can be

regularized using l_1 or l_2 penalties. Scikit-Learn actually adds an l_2 penalty by default.

The hyperparameter controlling the regularization strength of a Scikit-Learn `LogisticRegression` model is not alpha (as in other linear models), but its inverse: `C`. The higher the value of `C`, the less the model is regularized.

Multiclass Logistic Regression

- One can train a multiclass classifier using the full iris dataset with 4 input features and 3 classes.

```
X4 = iris["data"]
y4 = iris["target"]
```

```
log_reg.fit(X4, y4)
```

```
▸ LogisticRegression
```

```
log_reg.predict(X4[:, :])
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

- There is a hyperparemter `multi_class`, which can be set to

{‘auto’, ‘ovr’, ‘multinomial’}. The default is ‘auto’.

We have learned how to build a multiclass classifier from a binary classifier using ‘ovr’.

Multinomial Logistic Regression

- Also known as **Softmax Regression**.
- The goal is to classify feature vectors into $K \geq 2$ classes.
- We switch notation for parameters from θ to the matrix $W = (w_1, \ldots, w_K)$, where each w_k is a $(n+1)$-dimensional column vector, denoted by $w_k = (w_{k0}, w_{k1}, \ldots, w_{kn})^T$.
- We set w_K to be the zero vector $(0, 0, \ldots, 0)^T$.

  Since we will be computing the probabilities of K classes, which adds to 1, we only need the probabilities for classes $k = 1, \ldots, K-1$.
- Given an n-dimensional feature vector $x = (x_1, \ldots, x_n)$, let $\tilde{x} = (1, x_1, \ldots, x_n)^T$.

For the feature vector x, there is a score for each class k:

$$s_k(x) = w_k^T \tilde{x}.$$

($\tilde{x}^T w_k$ is also fine, depending on which one is viewed as variables.)

- The conditional probability is assumed to be

$$p_k \triangleq P(Y = k|X = x; W) = \frac{\exp(w_k^T \tilde{x})}{\sum_{j=1}^{K} \exp(w_j^T \tilde{x})} = \frac{\exp(w_k^T \tilde{x})}{1 + \sum_{j=1}^{K-1} \exp(w_j^T \tilde{x})},$$

for $k = 1, \ldots, K$.

In other words, $p_k \propto \exp(w_k^T \tilde{x})$. The denominator above is just for normalization.

- The model has the property that, for any two classes k and j,

$$\log \frac{P(Y = k|X = x; W)}{P(Y = j|X = x; W)} = w_k^T \tilde{x} - w_j^T \tilde{x} = (w_k^T - w_j^T)\tilde{x}.$$

- Once the model is trained, the predicted class is usually the mode of

$P(Y = k|X = x; \hat{W})$, i.e.,

$$\hat{y} = \arg\max_k P(Y = k|X = x; \hat{W}) = \arg\max_k \ \hat{w}_k^T \tilde{x}.$$

- Then, the decision boundaries are hyperplanes.
- The region of x to be classified into class k is $\{x : P(Y = k|X = x; \hat{W}) \geq P(Y = j|X = x; \hat{W}), \forall j\}$, which is the same as

  $$\{x : (\hat{w}_k^T - \hat{w}_j^T)\tilde{x} \geq 0, \forall j\}.$$

  The boundaries of the region lie on the hyperplanes $\{x : \hat{w}_k^T \tilde{x} = \hat{w}_j^T \tilde{x}\}$, one for each $j \neq k$.
- Example below: $n = 2$; the boundary hyperplanes are lines.

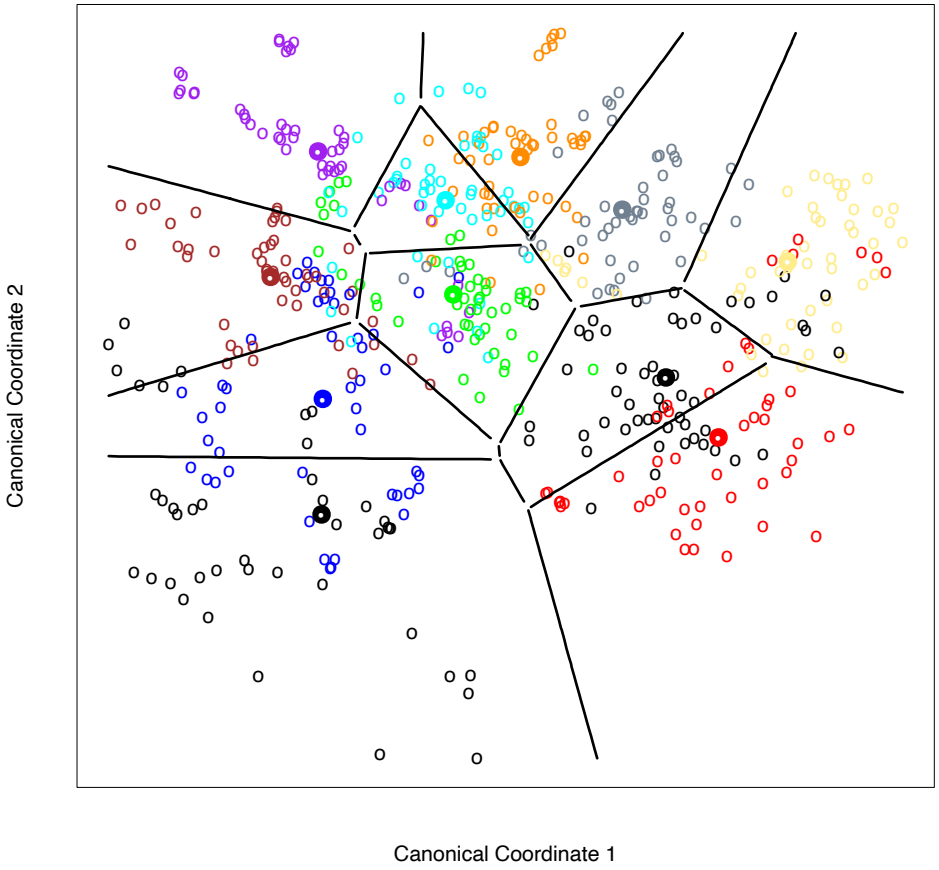


Figure 18: Decision regions - Figure 4.11 in Hastie et al.

- Terminology: Given K numbers $z_1, \ldots, z_K$, the mapping

$$\sigma = (\sigma_1, \ldots, \sigma_K) : (z_1, \ldots, z_K) \mapsto \left(\frac{e^{z_1}}{\sum_{j=1}^K e^{z_j}}, \ldots, \frac{e^{z_K}}{\sum_{j=1}^K e^{z_j}} \right)$$

  is called the standard **softmax function**, where each σ_k is the mapping:

$$\sigma_k : (z_1, \ldots, z_K) \mapsto \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}.$$

  - The softmax function is useful to pick out the maximum element in $(z_1, \ldots, z_K)$ in a smooth fashion.
  - There is often another scalar parameter $\beta > 0$, and the mapping becomes:

$$\sigma_\beta : (z_1, \ldots, z_K) \mapsto \left(\frac{e^{\beta z_1}}{\sum_{j=1}^K e^{\beta z_j}}, \ldots, \frac{e^{\beta z_K}}{\sum_{j=1}^K e^{\beta z_j}} \right).$$

  - When β becomes very large, the largest z_k (if unique) will dominate in the value $\frac{e^{\beta z_k}}{\sum_{j=1}^K e^{\beta z_j}}$, which is nearly 1; every other

term becomes nearly 0.

- When the softmax function values are interpreted as probabilities, we see that the probability mass is concentrated on the largest z_k.
- For combinatorial optimization such as finding a max-weight independent set on a graph, randomized algorithms are often devised with the help of the softmax function.

Softmax Regression: Objective Function

- In the training phase, softmax regression minimizes the **cross-entropy** cost function:

$$J(W) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{ik} \log p_{ik},$$

  where

  - $y_{ik} = 1$ if instance i has label k; 0 otherwise. Here, y_{ik} is understood as the actual probability that instance i is in class k.
  - each p_{ik} is the conditional probability that the label is class k given the input features are the vector x_i, i.e.,

$$p_{ik} = P(Y = k|X = x_i; W) = \sigma_k(w_k^T \tilde{x}_i) = \frac{\exp(w_k^T \tilde{x}_i)}{\sum_{j=1}^{K} \exp(w_j^T \tilde{x}_i)}.$$

- Note that for $K = 2$ (2 classes), the objective function is the same as

that of (16) for the 2-class logistic regression.

- In general, the cross entropy between two discrete probability distributions p and q is defined as $H(p, q) = -\sum_i p_i \log q_i$, where $p = (p_i)$ and $q = (q_i)$ are the probability mass functions.
- The cross entropy gradient vector for class k is:

$$\nabla_{w_k} J(W) = \frac{1}{m} \sum_{i=1}^{m} (p_{ik} - y_{ik}) x_i.$$

  One can compute the gradient for every class. This gives the gradient vector w.r.t. to all the variables in W.

- Then, one can use the gradient descent algorithm or its variants to minimize the cost function.
- Here is the code to use softmax regression for all the three classes but with only two input features (because we want to see the decision boundaries).

```
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10, random_state=42)
softmax_reg.fit(X, y)
```

  - Here, we specify the hyperparameter `multi_class` to be "multinomial", which means softmax regression.
  - We also specify the solver "lbfgs".
  - It also applies l_2 regularization by default, which you can control using the hyperparameter `C`.

- Some results:

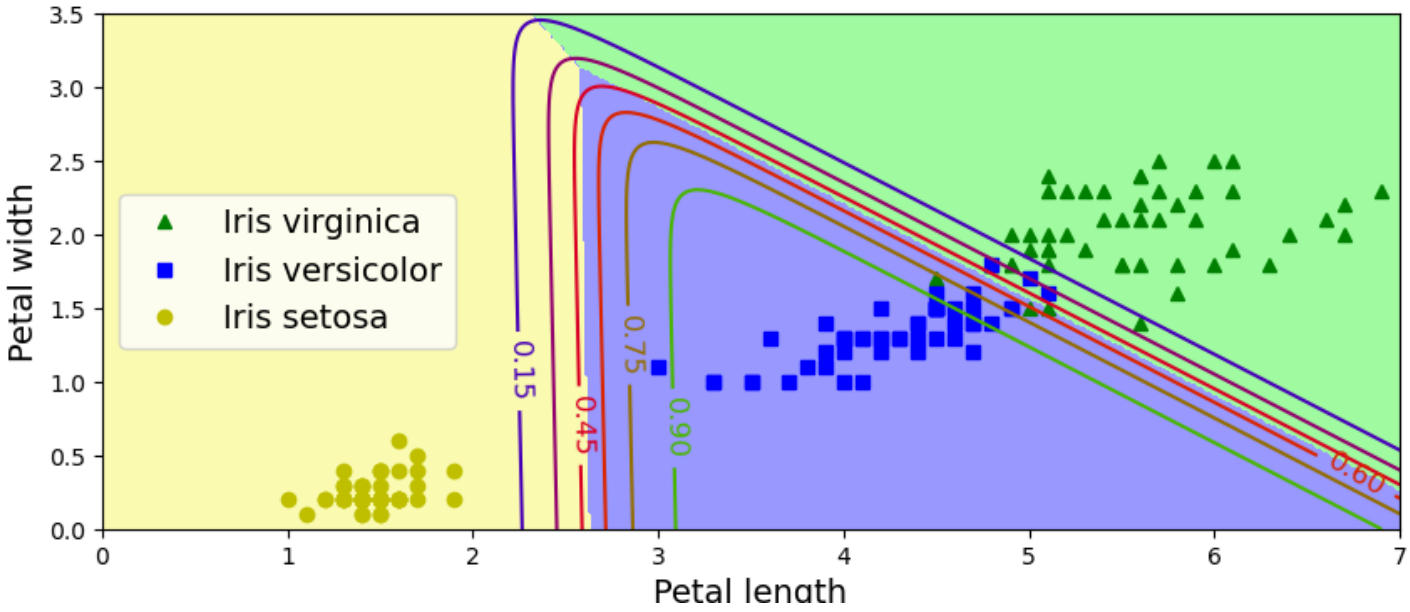
```
softmax_reg.predict([[5, 2]])
```

```
array([2])
```

```
softmax_reg.predict_proba([[5, 2]])
```

```
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

- Here are the decision regions and boundaries:



  - The decision regions are represented by three background colors. The decision boundaries between any two classes are linear: there are three such boundaries, each of which is a straight line.
  - The decision regions are carved by the three lines.
  - The level curves (curved lines) are for the probabilities for the Iris

versicolor class.

  - Since there are three classes, it is possible that the winning class has a probability only slightly greater than 33%, for instance, at the region for Iris Versicolor but close to where the three lines meet.

- Finally, let us try softmax regression on all 4 input features.

```
softmax_reg_4 = LogisticRegression(multi_class="multinomial",
            solver="lbfgs", C=10, max_iter=1000, random_state=42)
softmax_reg_4.fit(X4, y4)
```

▸ LogisticRegression

```
y4_predict=softmax_reg_4.predict(X4)
```

```
(y4==y4_predict).astype(int)
```

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

- We needed more iterations for the training algorithm to converge.
- The performance on the training data is good. There are 3 classification errors out of 150 instances.