Linear Algebra

Foundations of Machine Learning

Paul A. Jensen
University of Illinois at Urbana-Champaign

Contents

Part II

Nonlinear Systems

Part I of this book described methods for solving linear systems. These methods are definite — the solvability theorems tell us if a system is solvable and exactly how many solutions exist. The tools for linear systems are also constructive. If solutions exist, we have deterministic methods to find them.

In Part II we turn our attention to nonlinear systems. The nonlinearities remove the luxuries we encountered with linear systems. We will not know if a nonlinear system is solvable or how many solutions exist. Even when a solution exists, we will be forced to rely on iterative or stochastic methods to search for it.

There are nonlinear systems that are exempt from the above problems. The linear least-squares problems of Chapter 8 are nonlinear (quadratic), but we used the pseudoinverse to find a unique solution. In Chapter 11 we will see that the linear least-squares problem belongs to a special class of nonlinear problems because it is convex. Convex problems can be solved with relative ease, and learning to identify and exploit convexity is a powerful tool for nonlinear systems.

We will focus on two nonlinear problems. The first is the *root finding* problem: For a system of nonlinear equations $\mathbf{g}(\mathbf{x})$, what are the values of the vector $\mathbf{x}$ such that $\mathbf{g}(\mathbf{x}) = \mathbf{0}$? The second problem is the *optimization* problem: Find a vector $\mathbf{x}$ that minimizes the scalar function $f(\mathbf{x})$. These two problems are related, and algorithms that solve one problem can be used to solve the other. For example, consider a continuously differentiable function $f(\mathbf{x})$. If f has a minimum, it occurs when the gradient of f is equal to the zero vector. Minimizing the function f is equivalent to find a vector $\mathbf{x}$ such that $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ when $\mathbf{g}$ is the gradient of f. Similarly, imagine we want to find a zero of the nonlinear function $\mathbf{g}(\mathbf{x})$. If we define $f(\mathbf{x}) = \|\mathbf{g}(\mathbf{x})\|$, then the zero of the function $\mathbf{g}$ corresponds to the point at the minimum of the function f.

Solving nonlinear systems is more of an "art" than solving linear systems. We will learn several strategies that work well for some problems but not for others. There is no single best method for solving nonlinear problems, and we will focus on the strengths and weaknesses of each technique. In practice, you will learn to try methods that are inspired by the features of each problem.

Chapter 10

Root Finding

We've seen multiple methods for solving linear systems of equations. In this chapter we develop a method to solve nonlinear systems of equations using linear algebra. We begin with Newton's method for finding the roots of a single nonlinear equation. Then we generalize the method to systems of equations using a matrix formalism.

10.1 Nonlinear Functions

A nonlinear function is, simply put, a function that fails the tests for linearity. You might have been surprised that the affine function $g(x) = ax + b$ was nonlinear. The functions $g(x) = \cos x$, $g(x) = x^2$, and $g(x) = \log x$ are all nonlinear with respect to the independent variable x.

By convention we write nonlinear functions in the form

$$g(x) = 0$$

This convention is not a limitation, as any nonlinear function with a nonzero right hand side can be rewritten by moving the right hand terms to the left side. Writing nonlinear functions in this way lets us solve the function by identifying values where the function equals zero, i.e. by finding the *roots* of the function. For example, the equation

$$(x-1)^3 = 8$$

has a unique solution when $x = 3$. We can rewrite this equation as the function

$$g(x) = (x-1)^2 - 8 = 0$$

Notice that the function $g(x)$ has a root when $x = 3$, which is also the solution to the equation $(x-1)^3 = 8$.

Linear systems have exactly zero, one, or infinitely many solutions. By contrast, nonlinear systems can have any number of solutions. The function $g(x) = x^2 - 4$ has two roots: $x = 2$ and $x = -2$. Unlike linear systems, there is no grand solvability theorem for nonlinear systems. Except in special cases (for example, polynomials), we cannot tell *a priori* how many unique solutions exist for a nonlinear equation. Even when we know a solution exists, we do not have a general procedure like Gaussian elimination for finding solutions to nonlinear equations. Instead, we often rely on numerical techniques to find *some* of the roots of nonlinear functions.

10.2 Newton's Method

Given a function $g(x)$, how do we find its roots? One powerful method builds on an observation regarding the tangent lines of $g(x)$ near its roots. Imagine we are at a point x_0 that is near a root. The tangent line of $g(x)$ at the point x_0 will itself have a root that is closer to the root of $g(x)$. Let's call this new point x_1.

If we draw another tangent line for g at x_1, we see that the root of the tangent line is again closer to the root of g. We can repeat this procedure again and again, each time moving closer to the root of g. Rather than solve the nonlinear function g, we only need to solve a series of affine equations describing the tangent line at each iteration.

Let's formalize the above procedure. The starting point x_0, the values of g and its derivative g', and the root x_1 of the tangent line are related by

$$g'(x_0) = \frac{g(x_0)}{x_0 - x_1}$$

You can interpret this formula as "the slope of the tangent line at x_0 ($g'(x_0)$) is equal to the height of the function at x_0 ($g(x_0)$) divided by the distance between x_0 and x_1." Rearranging, we can find the root of the tangent line based on values at our current point.

$$x_1 = x_0 - \frac{g(x_0)}{g'(x_0)}$$

Now we know the location of x_1, a point closer to the root of the original function g. We can apply the same procedure starting at x_1 to find a closer point x_2, and so on.

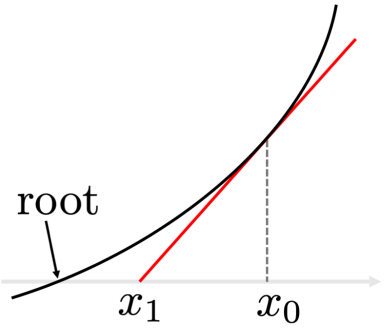


Figure 10.1: If a point x_0 is close to the root of a function (black), the tangent line (red) intersects the horizontal axis at a point x_1 that is closer to the root.

In other words, the slope of the tangent line $g'(x_0)$ is its rise $g(x_0)$ divided by its run $(x_0 - x_1)$.

Newton published a very limited version of the method that bears his name. British mathematician Thomas Simpson was the first to apply the technique to general systems of nonlinear equations. He also noted connections between nonlinear systems and optimization.

$$x_2 = x_1 - \frac{g(x_1)}{g'(x_1)}$$
$$x_3 = x_2 - \frac{g(x_2)}{g'(x_2)}$$
$$\vdots$$
$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)}$$

10.3 Convergence of Newton's Method

Let's find a root for the equation

$$g(x) = (x-4)^3 - 2x$$

By plotting the function, we see there is a root somewhere between $x = 6$ and $x = 6.5$. We can use Newton's Method to find a more precise estimate of the root. We first calculate the derivative

$$g'(x) = 3(x-4)^2 - 2$$

Let's choose our initial guess to be $x_0 = 6.0$. We're ready to calculate x_1.

$$\begin{aligned} x_1 &= x_0 - \frac{g(x_0)}{g'(x_0)} \\ &= x_0 - \frac{(x_0-4)^3 - 2x_0}{3(x_0-4)^2 - 2} \\ &= 6.0 - \frac{(6.0-4)^3 - 2(6.0)}{3(6.0-4)^2 - 2} \\ &= 6.4 \end{aligned}$$

We can check if we've found a root by evaluating $g(x_1)$. If x_1 is a root, $g(x_1)$ should equal zero.

$$g(x_1) = g(6.4) = 1.024 \neq 0$$

We haven't arrived at a root yet. Let's try another iteration of Newton's method to find a second guess (x_2) using x_1.

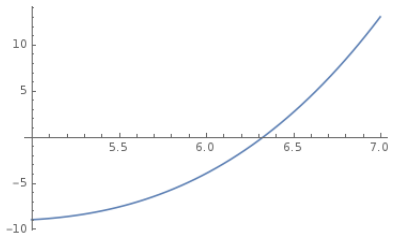


Figure 10.2: The function $g(x) = (x-4)^3 - 2x$ has a root between $x = 6$ and $x = 6.5$.

When studying numerical methods we will extend our answers far beyond the number of significant figures. As engineers we later trim or *truncate* our answers to an appropriate number of significant figures based on the uncertainty in the system.

$$\begin{aligned} x_2 &= x_1 - \frac{g(x_1)}{g'(x_1)} \\ &= x_1 - \frac{(x_1-4)^3 - 2x_1}{3(x_1-4)^2 - 2} \\ &= 6.4 - \frac{(6.4-4)^3 - 2(6.4)}{3(6.4-4)^2 - 2} \\ &= 6.332984293 \end{aligned}$$

The new value x_2 is closer to being a root: $g(6.332984293) = 0.03203498$. We can always move closer using more iterations as shown in the following table.

i	x_i	$g(x_i)$
0	6	-4
1	6.4	1.024
2	6.332984293	0.032034981
3	6.330748532	0.000034974
4	6.330746086	4.18421×10^{-11}

Newton's method converges quadratically once the x_i are close to the actual root. "Close" is not well defined and varies with each function. If an initial guess is far from the true root, Newton's method can either 1.) converge slowly until it becomes close enough for quadratic converge to kick in, or 2.) not converge at all. If Newton's method is converging slowly or diverges, you should try a different initial guess.

The quadratic convergence stems from our use of a linear approximation for the function, leaving a residual bounded by the quadratic terms.

10.4 Multivariable Functions

Newton's method works well for nonlinear functions of a single variable. We use a variant of Newton's method to solve multivariable functions. Multivariable functions accept a vector of inputs and produce a vector of outputs. We write the names of multivariable functions using bold, non-italicized font — $\mathbf{g}(\mathbf{x})$ — to remind us that a multivariable functions return a vector of outputs.

Multivariable functions are also called *multivariate* or *vector-valued* functions.

We're already familiar with linear multivariable functions like $\mathbf{g}(\mathbf{x}) = \mathbf{A}\mathbf{x}$. This function accepts a vector of inputs ($\mathbf{x}$) and returns another vector of outputs ($\mathbf{A}\mathbf{x}$). We can also define nonlinear multivariable functions. An example with three inputs and three outputs is

$$\mathbf{g}(\mathbf{x}) = \begin{pmatrix} x_1 - x_3 \\ x_3^2 + 2x_2 \\ \cos x_1 \end{pmatrix}$$

If $\mathbf{x} = \begin{pmatrix} 0 \\ -1 \\ 2 \end{pmatrix}$, then

$$\mathbf{g}(\mathbf{x}) = \begin{pmatrix} 0 - 2 \\ 2^2 + 2(-1) \\ \cos 0 \end{pmatrix} = \begin{pmatrix} -2 \\ 2 \\ 1 \end{pmatrix}$$

It's sometimes convenient to talk individually about the entries in the nonlinear function. We can write a multivariable function using the following notation

We use lowercase and italicized font (g_i) when referencing individual entries in a multivariable function since each entry produces only a single output.

$$\mathbf{g}(\mathbf{x}) = \begin{pmatrix} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ g_n(x_1, x_2, \ldots, x_n) \end{pmatrix}$$

For the example above, $g_1 = x_1 - x_3$; $g_2 = x_3^2 + 2x_2$; and $g_3 = \cos x_1$.

10.5 The Jacobian Matrix

For functions of a single variable, Netwon's method uses the derivative to construct a linear approximation. The multivariable analog of the derivative is matrix of partial derivatives called the *Jacobian*, which we write as $\mathbf{J}(\mathbf{x})$.

The Jacobian is named after German mathematician Carl Gustav Jacob Jacobi. I assume it is based on his last name, or possibly his second-to-last name.

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_n}{\partial x_1} & \frac{\partial g_n}{\partial x_2} & \cdots & \frac{\partial g_n}{\partial x_n} \end{pmatrix}$$

The (i,j)th entry in the Jacobian is the partial derivative of the ith function with respect to the jth variable. If a multivariable function has n inputs and n outputs, its Jacobian is a square $n \times n$ matrix.

Let's compute the Jacobian for the function $\mathbf{g}(\mathbf{x}) = \begin{pmatrix} x_1 - x_3 \\ x_3^2 + 2x_2 \\ \cos x_1 \end{pmatrix}$.

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{\partial}{\partial x_1}(x_1 - x_3) & \frac{\partial}{\partial x_2}(x_1 - x_3) & \frac{\partial}{\partial x_3}(x_1 - x_3) \\ \frac{\partial}{\partial x_1}\left(x_3^2 + 2x_2\right) & \frac{\partial}{\partial x_2}\left(x_3^2 + 2x_2\right) & \frac{\partial}{\partial x_3}\left(x_3^2 + 2x_2\right) \\ \frac{\partial}{\partial x_1}(\cos x_1) & \frac{\partial}{\partial x_2}(\cos x_1) & \frac{\partial}{\partial x_3}(\cos x_1) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & -1 \\ 0 & 2 & 2x_3 \\ -\sin x_1 & 0 & 0 \end{pmatrix}$$

10.6 Multivariable Newton's Method

For functions of a single variable, Newton's method iterates with the formula

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

Using a multivariable linear approximation, we can define the multivariable analogue of Newton's method.

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}^{-1}(\mathbf{x}_i)\mathbf{g}(\mathbf{x}_i)$$

As an example, let's find a root of the function

$$\mathbf{g} = \begin{pmatrix} x_1 x_2 - 2 \\ -x_1 + 3x_2 + 1 \end{pmatrix}$$

First we calculate the Jacobian matrix.

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} x_2 & x_1 \\ -1 & 3 \end{pmatrix}$$

Using an initial guess of $\mathbf{x}_0 = \begin{pmatrix} -1 \\ -1 \end{pmatrix}$ we begin iterating.

$$\begin{aligned}
\mathbf{x}_1 &= \mathbf{x}_0 - \mathbf{J}^{-1}(\mathbf{x}_0)\mathbf{g}(\mathbf{x}_0) \\
&= \begin{pmatrix} -1 \\ -1 \end{pmatrix} - \begin{pmatrix} -1 & -1 \\ -1 & 3 \end{pmatrix}^{-1} \begin{pmatrix} (-1)(-1) - 2 \\ -(-1) + 3(-1) + 1 \end{pmatrix} \\
&= \begin{pmatrix} -2 \\ -1 \end{pmatrix}
\end{aligned}$$

Now we use $\mathbf{x}_1$ to find the next guess $\mathbf{x}_2$.

$$\begin{aligned}
\mathbf{x}_2 &= \mathbf{x}_1 - \mathbf{J}^{-1}(\mathbf{x}_1)\mathbf{g}(\mathbf{x}_1) \\
&= \begin{pmatrix} -2 \\ -1 \end{pmatrix} - \begin{pmatrix} -1 & -2 \\ -1 & 3 \end{pmatrix}^{-1} \begin{pmatrix} (-2)(-1) - 2 \\ -(-2) + 3(-1) + 1 \end{pmatrix} \\
&= \begin{pmatrix} -2 \\ -1 \end{pmatrix}
\end{aligned}$$

Our guess $\mathbf{x}_2$ is exactly equal to the previous guess $\mathbf{x}_1$. Since our guess didn't change we are probably at a root. We can check by evaluating $\mathbf{g}(\mathbf{x}_2)$.

$$\mathbf{g}(\mathbf{x}_2) = \begin{pmatrix} (-2)(-1) - 2 \\ -(-2) + 3(-1) + 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Indeed, the vector $\begin{pmatrix} -2 \\ -1 \end{pmatrix}$ is a solution to our equation.

Nonlinear systems often have many solutions. Newton's method converges to the solution nearest the initial guess. If we chose the point $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ as our initial guess, Newton's method on the same function would converge to the root $\mathbf{x} = \begin{pmatrix} 3 \\ 2/3 \end{pmatrix}$ after four iterations.

"Nearest" in the topological sense, i.e. the solution that is down the gradient of the function at the initial guess.

10.7 * Gauss-Newton Method

The multivariate Newton's method assumes that the inputs and outputs of the function $\mathbf{g}$ have the same dimension. If the dimensions disagree, the Jacobian matrix will not be square and its inverse will not be defined. In some cases,

we can apply a related method — the Gauss-Newton Method — that uses the pseudoinverse of the nonsquare Jacobian.

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}^+(\mathbf{x}_i)\mathbf{g}(\mathbf{x}_i)$$

For convergence, we require that the function $\mathbf{g}$ accepts an n-dimensional input vector and outputs an m-dimensional vector, where $m > n$.

10.8 Root Finding with Finite Differences

In all of our examples we have been able to calculate the derivative of the function g (or the Jacobian of multivariate function $\mathbf{g}$) using calculus. This is not always possible. Sometimes the function g is unknown to us or is very complicated. Sometimes g is a simulation that includes random numbers, like a traffic simulator that models random arrivals and departures of cars. In this case we cannot calculate the derivative without knowing what random numbers will appear when the function is later evaluated.

An alternative is to use finite differences to approximate the derivative. Recall from Chapter 5 that the derivative $g'(x)$ can be approximated by

$$g'(x) \approx \frac{g(x + \Delta x) - g(x)}{\Delta x}$$

for some small value Δx. Let's return to an example from earlier in this chapter:

$$g(x) = (x - 4)^3 - 2x$$

We can approximate the derivative at $x = 1$ with $\Delta x = 0.1$.

$$\begin{aligned} g'(1) &\approx \frac{g(1.1) - g(1)}{0.05} \\ &= \frac{(1.1 - 4)^3 - 2(1.1) - (1 - 4)^3 + 2(1)}{0.1} \\ &= 24.0 \end{aligned}$$

The actual value of the derivative at $x = 1$ is

$$\begin{aligned} g'(1) &= 3(1 - 4)^2 - 2 \\ &= 25 \end{aligned}$$

The accuracy of our approximation depends on both the size of the perturbation Δx and on the nonlinearity of the function. Using $\Delta x = 0.01$ puts us closer to

the correct value of the derivative ($g'(x) \approx 25$), while a perturbation of $\Delta x = 0.2$ makes the approximation worse ($g'(x) \approx 23$). This is a big problem when the function g is *stochastic*, meaning it depends on random values. Stochastic functions are "noisy" and their outputs always include error. Making the perturbation smaller can amplify the effects of this error, but large perturbations will lead to a poor approximation of the derivative. One solution is to construct our approximation of the derivative using multiple finite difference measurements. Multiple measurements can average out the error, but they require more computation.

In addition to numerical issues, a finite difference approximation can be expensive to evaluate for multivariate functions. To approximate a partial derivative we perturb the vector $\mathbf{x}$ along a single dimension. We can write the perturbation using the Cartesian unit vectors

$$\frac{\partial g}{\partial \mathbf{x}_i} \approx \frac{g(\mathbf{x} + \Delta \hat{\mathbf{e}}_i) - g(\mathbf{x})}{\Delta}$$

where the scalar Δ is the perturbation size. Every entry in the Jacobian must be approximated using a function evaluation with a perturbed input. The Jacobian of an n-dimensional function has n^2 entries, so a finite difference approximation requires n^2 function evaluations.

A more recent solution is a technique called *automatic differentiation*, also known as "autodiff" or "autograd" (which is short for automatic gradient). Automatic differentiation uses specialized software to compute derivatives by tracking the mathematical operations in a function and applying the chain rule. Automatic differentiation is available in many state-of-the-art machine learning packages. It can calculate the true derivative of a function when applied correctly. Many implementations include an option to check the automatic differentiation results using finite differences.

10.9 Practical Considerations

Solving nonlinear equations is an art. Here are some tips.

- Nonlinear equations rarely have a single solution. Solvers try many (hundreds or thousands) of initial guesses to find several solutions. There is no general method for determining the total number of roots for a nonlinear system.
- Software packages like MATLAB's `fsolve` function can find roots with a variety of algorithms. Many techniques find points near roots and use Newton's method to finish the search.

- Software packages often allow users to provide both the function and the Jacobian. Knowing the Jacobian explicitly almost always improves speed and numerical stability. If the user doesn't provide a Jacobian, the software will estimate the Jacobian at every iteration using finite differences.

- Single variable Newton's method requires the function be continuously differentiable. Multivariable functions require the Jacobian be invertible. So-called "gradient free" algorithms are available for functions with poorly behaving, computationally expensive, or discontinuous derivatives.

- The multivariable Newton's method involves inverting the Jacobian, which is computationally expensive. Instead, numerical solvers rearrange the iteration equation:

$$\begin{aligned} \mathbf{x}_{i+1} &= \mathbf{x}_i - \mathbf{J}^{-1}(\mathbf{x}_i)\mathbf{g}(\mathbf{x}_i) \\ \mathbf{J}(\mathbf{x}_i)\mathbf{x}_{i+1} &= \mathbf{J}(\mathbf{x}_i)\mathbf{x}_i - \mathbf{J}(\mathbf{x}_i)\mathbf{J}^{-1}(\mathbf{x}_i)\mathbf{g}(\mathbf{x}_i) \\ \mathbf{J}(\mathbf{x}_i)\left(\mathbf{x}_{i+1} - \mathbf{x}_i\right) &= -\mathbf{g}(\mathbf{x}_i) \end{aligned}$$

  In this form, the solver can use Gaussian elimination on the augmented matrix $[\mathbf{J}(\mathbf{x}_i) \;\; -\mathbf{g}(\mathbf{x}_i)]$ to solve for $\mathbf{x}_{i+1} - \mathbf{x}_i$; adding $\mathbf{x}_i$ gives the new estimate for $\mathbf{x}_{i+1}$.

Chapter 11

Optimization and Convexity

We formulated the least squares method and linear regression as optimization problems. Our goal was to minimize the sum of the squared errors by choosing parameters for the linear model. Optimization problems have enormous utility in data science, and most model fitting techniques are cast as optimizations. In this chapter, we will develop a general framework for describing and solving several classes of optimization problems. We begin by reviewing the fundamentals of optimization. Next, we discuss convexity, a property that greatly simplifies the search for optimal solutions. Finally we derive vector expressions for common geometric constructs and show how linear systems give rise to convex problems.

11.1 Optimization

Optimization is the process of minimizing or maximizing a function by selecting values for a set of variables or parameters (called *decision variables*). If we are free to choose any values for the decision variables, the optimization problem is *unconstrained*. If our solutions must obey a set of constraints, the problem is a *constrained optimization*. In constrained optimization, any set of values for the decision variables that satisfies the constraints is called a *feasible solution*. The goal of constrained optimization is to select the "best" feasible solution.

Optimization problems are formulated as either minimizations or maximizations. We don't need to discuss minimization and maximization separately, since minimizing $f(x)$ is equivalent to maximizing $-f(x)$. Any algorithm for minimizing can be used for maximizing by multiplying the objective by -1, and vice versa. For the rest of this chapter, we'll talk about minimizing functions. Keep in mind

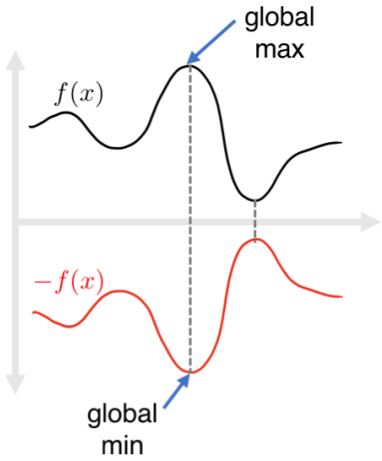


Figure 11.1: The maximum of a function $f(x)$ can be found by minimizing $-f(x)$.

that everything we discuss can be applied to maximization problems by switching the sign of the objective.

During optimization we search for minima. A minimum can either be *locally* or *globally* minimal. A global minimum is has the smallest objective value of any feasible solution. A local minimum has the smallest objective value for any of the feasible solutions in the surrounding area. The input to a function that yields the minimum is called the *argmin*, since it is the argument to the function that gives the minimum. Similarly, the *argmax* of a function is the input that gives the function's maximum. Consider the function $f(x) = 3 + (x-2)^2$. This function has a single minimum, f(2) = 3. The minimum is 3, while the argmin is $x = 2$, the value of the decision variable at which the minimum occurs. For optimization problems, the minimum (or maximum) is called the *optimal objective value*. The argmin (or argmax) is called the *optimal solution*.

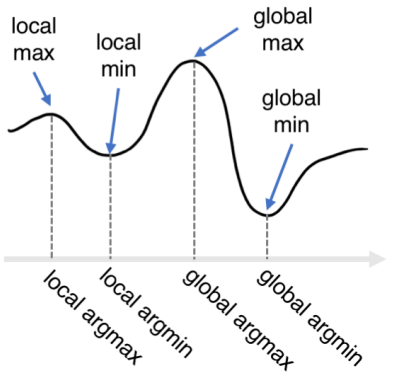


Figure 11.2: Minima and maxima of a function can be local or global.

11.1.1 Unconstrained Optimization

You already know how to solve unconstrained optimization problems in a single variable: set the derivative to the function equal to zero and solve. This method of solution relies on the observation that both maxima and minima occur when the slope of a function is zero. However, it is important to remember that not all roots of the derivative are maxima or minima. Inflection points (where the derivative changes sign) also have derivatives equal to zero. You must always remember to test the root of the derivative to see if you've found a minimum, maximum, or inflection point. The easiest test involves the sign of the second derivative. If the second derivative at the point is positive, you've found a minimum. If it's negative, you've found a maximum. If the second derivative is zero, you've found an inflection point.

A similar approach works for optimizing multivariate functions. In this case one solves for points where the gradient is equal to zero, checking that you've not found an inflection point (called "saddle points" in higher dimensions).

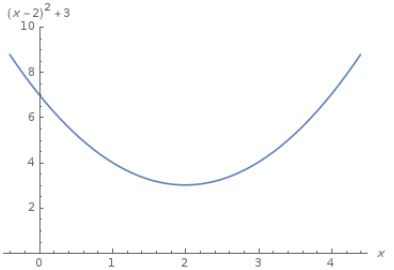


Figure 11.3: The function $f(x) = 3 + (x-2)^2$ has a minimum of $f = 3$ at argmin $x = 2$.

Any point where the derivative of a function equals zero is called an *extreme point* or *extremum*. Setting the derivative of a function equal to zero and solving for the extrema is called *extremizing* a function.

11.1.2 Constrained Optimization

Constrained optimization problems cannot be solved by finding roots of the derivatives of the objective. Why? It is possible that the minima or maxima of the unconstrained problem lie outside the feasible region of the constrained problem. Consider our previous example of $f(x) = 3 + (x-2)^2$, which we know has an argmin at $x = 2$. Say we want to solve the constrained problem

$$\min f(x) = 3 + (x-2)^2 \quad \text{s.t.} \quad x \leq 1$$

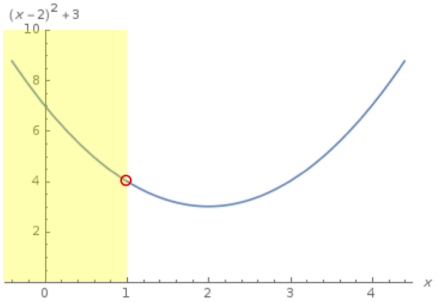


Figure 11.4: The yellow region is the feasible space ($x \leq 1$). The global argmin occurs at $x = 1$. The derivative of the function is not zero at this point.

The root of the derivative of f is still at $x = 2$, but values of x greater than one are not feasible. From the graph we can see that the minimum feasible value occurs at $x = 1$. The value of the derivative at $x = 1$ is -2, not zero.

In general, constrained optimization is a challenging field. Finding global optima for constrained problems is an unsolved area or research, one which is beyond the scope of this course. However, there are classes of problems that we can solve to optimality using the tools of linear algebra. These problems form the basis of many advanced techniques in data science.

11.2 Convexity

Many "solvable" optimization problems rely on a property called *convexity*. Both sets and functions can be convex.

11.2.1 Convex sets

A set of points is *convex* if given any two points in the set, the line segment connecting these points lies entirely in the set. You can move from any point in the set to any other point in the set without leaving the set. Circles, spheres, and regular polygons are examples of convex sets.

To formally define convexity, we construct the line segment between any two points in the set.

Definition. *A set S is convex if and only if given any $\mathbf{x} \in S$ and $\mathbf{y} \in S$ the points $\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}$ are also in S for all scalars $\lambda \in [0, 1]$.*

The expression $\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}$ is called a *convex combination* of $\mathbf{x}$ and $\mathbf{y}$. A convex combination of two points contains all points on the line segment between the two points. To see why, consider the 1-dimensional line segment between points 3 and 4.

$$\lambda(3) + (1 - \lambda)(4) = 4 - \lambda, \quad \lambda \in [0, 1]$$

When $\lambda = 0$, the value of the combination is 4. As λ moves from 0 to 1, the value of the combination moves from 4 to 3, covering all values in between.

Convex combinations work in higher dimensions as well. The convex combination of the vectors $\begin{pmatrix}1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\1\end{pmatrix}$ is

$$\lambda \begin{pmatrix}1\\0\end{pmatrix} + (1 - \lambda) \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\lambda\\1 - \lambda\end{pmatrix}$$

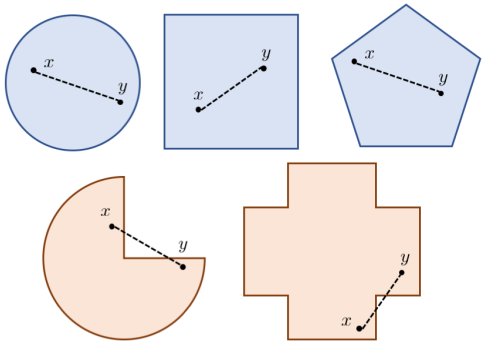


Figure 11.5: The blue shapes are convex. The red shapes are not convex.

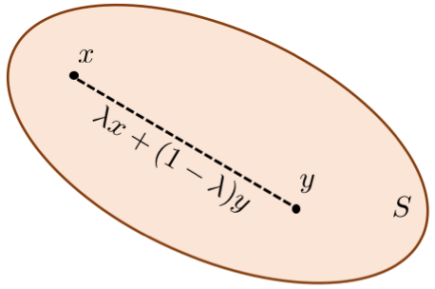


Figure 11.6: The segment connecting x and y can be defined as $\lambda x + (1 - \lambda)y$ for $\lambda \in [0, 1]$.

The combination goes from the first point $\begin{pmatrix}0\\1\end{pmatrix}$ when $\lambda = 0$ to the second point $\begin{pmatrix}1\\0\end{pmatrix}$ when $\lambda = 1$. Halfway in between, $\lambda = 1/2$ and the combination is $\begin{pmatrix}1/2\\1/2\end{pmatrix}$, which is midway along the line connecting $\begin{pmatrix}1\\0\end{pmatrix}$ and $\begin{pmatrix}0\\1\end{pmatrix}$. Sometimes it is helpful to think of a convex combination as a weighted sum of $\mathbf{x}$ and $\mathbf{y}$. The weighting (provided by λ) moves the combination linearly from $\mathbf{y}$ to $\mathbf{x}$ as λ goes from 0 to 1.

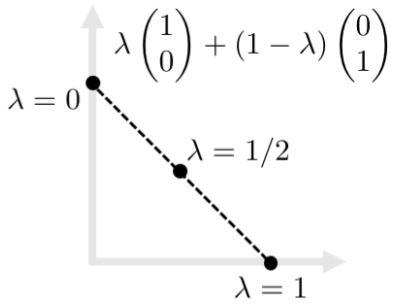


Figure 11.7: A convex combination in 2D.

11.2.2 Convex functions

There is a related definition for *convex functions*. This definition formalizes our visual idea of convexity (lines that curve upward) and concavity (lines that curve downward).

Definition 1. *A function f is convex if and only if*

$$f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y}), \quad \lambda \in [0,1]$$

This definition looks complicated, but the intuition is simple. If we plot a convex (upward curving) function, any chord – a segment drawn between two points on the line – should lie above the line. We can define the chord between any two points on the line, say $f(\mathbf{x})$ and $f(\mathbf{y})$ as a convex combination of these points, i.e. $\lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y})$. This is the right hand side of the above definition. For convex functions, we expect this cord to be greater than or equal to the function itself over the same interval. The interval is the segment from $\mathbf{x}$ to $\mathbf{y}$, or the convex combination $\lambda\mathbf{x}+(1-\lambda)\mathbf{y}$. The values of the function over this interval are therefore $f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y})$, which is the left hand side of the definition.

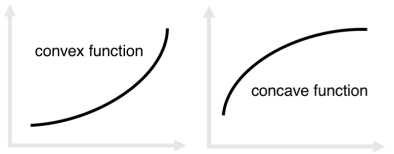


Figure 11.8: Convex functions curve upward. Concave functions curve downward.

11.2.3 Convexity in Optimization

Why do we care about convexity? In general, finding local optima during optimization is easy; just pick a feasible point and move downward (during minimization) until you arrive at a local minimum. The truly hard part of optimization is finding global optima. How can you be assured that your local optimum is a global optimum unless you try out all points in the feasible space?

Fortunately, convexity solves the local vs. global challenge for many important problems, as we see with the following theorem.

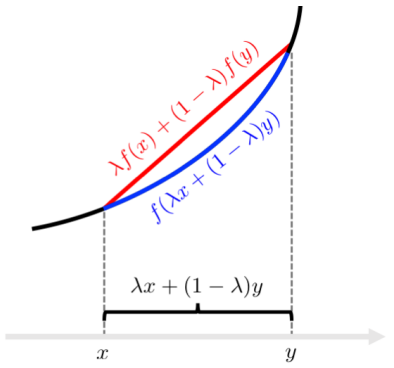


Figure 11.9: The chord connecting any two points of a convex function (red) lies above the function (blue).

Theorem. *When minimizing a convex function over a convex set, all local minima are global minima.*

Convex functions defined over convex sets must have a special shape where no *strictly* local minima exist. There can be multiple local minima, but all of these local minima must have the same value (which is the global minimum).

All local minima are *less than or equal to* the global minimum. Strictly local minima must be *less than* the global minimum.

Let's prove that all local minima are global minima when minimizing a convex function over a convex set.

Proof. Suppose the convex function f has a local minimum at $\mathbf{x}'$ that is not the global minimum (which is at $\mathbf{x}^*$). By the convexity of f,

$$f(\lambda\mathbf{x}' + (1-\lambda)\mathbf{x}^*) \leq \lambda f(\mathbf{x}') + (1-\lambda)f(\mathbf{x}^*)$$

Since $\mathbf{x}'$ is at a local, but not global, minimum, we know that $f(\mathbf{x}') > f(\mathbf{x}^*)$. If we replace $f(\mathbf{x}^*)$ on the right hand side by the larger quantity $f(\mathbf{x}')$, the inequality ($\leq$) becomes a strict inequality ($<$). (Even if both sides were equal, adding a small amount to the right hand side would still make it larger.) We now have

$$f(\lambda\mathbf{x}' + (1-\lambda)\mathbf{x}^*) < \lambda f(\mathbf{x}') + (1-\lambda)f(\mathbf{x}')$$

which, by simplifying the right hand side, becomes

$$f(\lambda\mathbf{x}' + (1-\lambda)\mathbf{x}^*) < f(\mathbf{x}')$$

This statement says that the value of the function f on any point on the line segment from $\mathbf{x}'$ to $\mathbf{x}^*$ is less than the value of the function at $\mathbf{x}'$. If this is true, we can find a point arbitrarily close to $\mathbf{x}'$ that is below our supposed local minimum $f(\mathbf{x}')$. Clearly, $f(\mathbf{x}')$ cannot be a local minimum if we can find a lower point arbitrarily closer to it. Our conclusion contradicts our original supposition. No local minimum can exist that are not equal to the global minimum. □

For a simpler, yet less intuitive argument, let $\lambda = 1$. Then the inequality becomes $f(\mathbf{x}') < f(\mathbf{x}')$, which is nonsense.

The previous proof seemed to rely only on the convexity of the objective function, not on the convexity of the solution set. The role of convexity of the set is hidden. When we make an argument about a line drawn from the local to the global minimum, we assume that all the points on the line are feasible. Otherwise, it does not matter if they have a lower objective than the local minimum, since they would not be allowed. By assuming the solution set is convex, we are assured that any point on this line is also feasible.

11.2.4 Convexity of Linear Systems

This course focuses on linear functions and systems of linear equations. It would be enormously helpful if linear functions and the solution set of linear systems

were convex. Then we can look for local optima during optimization and know that we've found global optima.

Let's first prove the convexity of linear functions. For a function to be convex, we require that a line segment connecting any two points in the line lie above or one the line. For linear functions, this is intuitively true. The line segment connecting any two points is the line itself, so it always lies on the line. As a more formal argument, we describe a linear function as the product between a vector of coefficients $\mathbf{c}$ and $\mathbf{x}$, i.e. $f(\mathbf{x}) = \mathbf{c}^\mathsf{T}\mathbf{x}$. Let's start with the values of the function over the range spanned by arbitrary points $\mathbf{x}$ and $\mathbf{y}$. The segment of the domain corresponds to the convex combination $\lambda\mathbf{x} + (1-\lambda)\mathbf{y}$. The values of the function over this interval are

By convention, all vectors are column vectors, including $\mathbf{c}$; this requires a transposition to be conformable for multiplication by $\mathbf{x}$.

$$\begin{aligned} f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) &= \mathbf{c}^\mathsf{T}(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \\ &= \mathbf{c}^\mathsf{T}\lambda\mathbf{x} + \mathbf{c}^\mathsf{T}(1-\lambda)\mathbf{y} \\ &= \lambda\mathbf{c}^\mathsf{T}\mathbf{x} + (1-\lambda)\mathbf{c}^\mathsf{T}\mathbf{y} \\ &= \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y}) \end{aligned}$$

which satisfies the definition of convexity: $f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y})$.

Now let's turn to a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$. We want to show that the set of all solutions for this system (the *solution space*) is convex. Let's assume we have two points in the solution space, $\mathbf{x}$ and $\mathbf{y}$. Since $\mathbf{x}$ and $\mathbf{y}$ are solutions, we know that $\mathbf{A}\mathbf{x} = \mathbf{b}$ and $\mathbf{A}\mathbf{y} = \mathbf{b}$. If the solution set is convex, any point in the convex combination of $\mathbf{x}$ and $\mathbf{y}$ is also a solution.

Following the conventions of the optimization field, we call the right hand side of linear systems the column vector $\mathbf{b}$, not $\mathbf{y}$ as we have said previously.

$$\begin{aligned} \mathbf{A}(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) &= \mathbf{A}\lambda\mathbf{x} = \mathbf{A}(1-\lambda)\mathbf{y} \\ &= \lambda\mathbf{A}\mathbf{x} + (1-\lambda)\mathbf{A}\mathbf{y} \\ &= \lambda\mathbf{b} + (1-\lambda)\mathbf{b} \\ &= \mathbf{b} \end{aligned}$$

Since $\mathbf{A}(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) = \mathbf{b}$, we know that all points on the line between $\mathbf{x}$ and $\mathbf{y}$ are solutions, so the solution set is convex.

Chapter 12

Gradient Descent

It's time to formalize the walking downhill method of optimization. This section introduces the *gradient descent* method, an iterative technique that takes steps downhill until a local minimum is found. As we will see in the coming chapters, gradient descent is not a single algorithm but instead a family of related algorithms. The defining feature of gradient descent is the use of local curvature of the objective function — the gradient — to identify the downhill direction.

12.1 Optimization by Gradient Descent

Let's begin with some notation. Our goal is to solve the problem

$$\min_{\mathbf{x}} f(\mathbf{x}),$$

which is read "minimize, by choice of $\mathbf{x}$, the function $f(\mathbf{x})$". We are searching for an input vector $\mathbf{x}$ that minimizes the scalar-valued function f. Optimization problem require that the objective function f be scalar-valued.

If the objective is a multivariate function $\mathbf{f}$, we can minimize $\|\mathbf{f}\|$ instead.

Gradient descent is an iterative technique. We begin with an initial guess $\mathbf{x}^{(0)}$, we find a sequence of better guesses

$$\mathbf{x}^{(0)} \to \mathbf{x}^{(1)} \to \mathbf{x}^{(2)} \to \cdots \to \mathbf{x}^{(k-1)} \to \mathbf{x}^{(k)}$$

so that each guess decreases the objective function:

$$f(\mathbf{x}^{(0)}) > f(\mathbf{x}^{(1)}) > f(\mathbf{x}^{(2)}) > \cdots > f(\mathbf{x}^{(k-1)}) > f(\mathbf{x}^{(k)}).$$

We keep iterating with gradient descent until there is no downhill direction, at which point we are by definition at a local minimum. The final guess $\mathbf{x}^{(k)}$ will be the local argmin.

The key to gradient descent is the *update rule*, a formula that tells us how to pick a next guess from the current guess. Imagine we are in the middle of gradient descent at guess $\mathbf{x}^{(k)}$. The update rules says that the next guess $\mathbf{x}^{(k+1)}$ will be our current guess plus a step in the downhill direction, or

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \big(\text{downhill step}\big)\,.$$

Whatever this "downhill step" is, we can already see that it must be a vector. The current guess $\mathbf{x}^{(k)}$ is a vector, and addition is only defined if the downhill step is a vector of the same size. Each entry in the downhill step vector is a downhill step for the corresponding entry in our guess $\mathbf{x}^{(k)}$.

It helps to break the downhill step vector into two parts: a vector that points in the downhill direction, and a scalar that represents the size of the step we'll take. We can rewrite the update rule as a product of these two parts:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \big(\text{step size}\big)\,\big(\text{downhill direction}\big)\,.$$

The step size is simply a scalar, so let's call it α and forget about it for a while. The step size is a *hyperparameter* of gradient descent. A hyperparameter is a variable in a training algorithm that is not a parameter of the model. Hyperparameters affect how a model is trained but are not used to make predictions once the training is complete. We'll have much more to say about the step size hyperparameter later in the chapter. For now, our update rule is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\,\big(\text{downhill direction}\big)\,.$$

What direction is downhill? We can find a direction that decreases the objective function f using its gradient. Let's define the function $\mathbf{g}(\mathbf{x})$ to be the gradient of the function f. Notice how the function $\mathbf{g}$ is vector-valued (and therefore written in bold font). The gradient is a vector of partial derivatives, one for every input:

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

Importantly, **the gradient points uphill, not downhill**, as shown in Figure 12.1. The downhill direction is the negative of the gradient: $-\mathbf{g}(\mathbf{x})$.

Putting everything together, our final update rule is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha\,\mathbf{g}(\mathbf{x}^{(k)}) \tag{12.1}$$

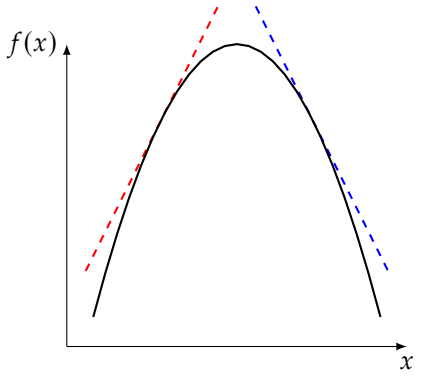


Figure 12.1: The gradient points in the uphill direction. The red tangent line has a positive gradient (slope), but the downhill direction is in the negative direction. The blue tangent line has a negative gradient but the downhill direction points toward $+x$.

where $\mathbf{g}(\mathbf{x}^{(k)})$ is the gradient of the objective function evaluated at the current iterate $\mathbf{x}^{(k)}$. We'll see this update rule repeatedly, and we must remember the origin of the minus sign. The new iterate $\mathbf{x}^{(k+1)}$ is the previous iterate $\mathbf{x}^{(k)}$ plus a step in the downhill direction; however, the downhill direction is $-\mathbf{g}(\mathbf{x}^{(k)})$, and the negative sign on the gradient can mislead us into thinking that we're subtracting, rather than adding, a step in the downhill direction. If you find yourself forgetting why there's a minus sign, just remember that the update rule can also be written $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\,(-\mathbf{g}(\mathbf{x}^{(k)}))$.

In multivariable calculus you may have written the gradient of a function $f(\mathbf{x})$ as $\nabla f(\mathbf{x})$. We avoid this notation in favor of $\mathbf{g}(\mathbf{x})$ for three reasons.

1. The symbol $\mathbf{g}(\mathbf{x})$ is simpler and can be bolded to remind us that the gradient is a vector.

2. It emphasizes the connection between optimization and root finding using the gradient ($\mathbf{g}(\mathbf{x}) = \mathbf{0}$).

3. Some gradient descent algorithms do not use the true gradient of the objective function, instead relying on an approximation or estimate of the gradient. We can think of the function $\mathbf{g}(\mathbf{x})$ as any "gradient-like" thing and still use the update rule in equation (12.1).

Let's stop for some examples. The first example is a the one-dimensional polynomial

$$f(x) = x^4 - 2x^3 - 23x^2 + 24x + 147.$$

We can use gradient descent to find a local minimum beginning with the guess $x^{(0)} = 2$. In one dimension, the gradient of $f(x)$ is the ordinary derivative

$$g(x) = \frac{df}{dx} = 4x^3 - 6x^2 - 46x + 24.$$

Let's assume we're given a step size $\alpha = 0.01$; we'll play around with the step size later. We begin iterating with our update rule.

$$\begin{aligned} x^{(1)} &= x^{(0)} - \alpha\, g(x^{(0)}) \\ &= 2 - 0.1\, g(2) \\ &= 2.6 \end{aligned}$$

The initial iterate $x^{(0)}$ had an objective value of $f(x^{(0)}) = 103$. After one round of gradient descent, the next iterate ($x^{(1)} = 2.6$) decreased the objective function to $f(x^{(1)}) = 64.4656$. The following tables shows the results of the first eight iterations.

Iteration	x	$f(x)$
0	2	103
1	2.6	64.46560
2	3.25856	24.53280
3	3.77059	5.41262
4	3.97379	3.03341
5	3.99919	3.00003
6	3.99998	3.00000
7	4.00000	3.00000
8	4.00000	3.00000

12.2 Linear Least-squares with Gradient Descent

As a second example, let's fit a linear model using gradient descent. As we learned in Chapter 8, the linear regression problem $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$ can be solved by pseudoinverting the design matrix $\mathbf{X}$ to find the parameter estimates $\boldsymbol{\beta} = \mathbf{X}^{+}\mathbf{y}$. Pseudoinversion gives the least-squares estimates for the parameters $\boldsymbol{\beta}$, and the same solution can be obtained by minimizing the loss function

$$L(\boldsymbol{\beta}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i^{\text{pred}} - y_i^{\text{true}}\right)^2$$

Let's solve the linear regression problem using gradient descent on the loss function. In the univariate case with an intercept, our linear model takes the form $y^{\text{pred}} = \beta_0 + \beta_1 x$. Using the five data points from the table on page 58, our loss function is

$$L(\boldsymbol{\beta}) = \frac{1}{2}\sum_{i=1}^{5}\left(\beta_0 + \beta_1 x_i - y_i^{\text{true}}\right)^2$$

Be careful with the notation here. The function we are minimizing is the loss L (not f as before), and we are searching for a parameter vector $\boldsymbol{\beta}$ to minimize the loss. As for all linear regression problems, the pairs of data (x_i, y_i) are known.

Our first step is to calculate the gradient of the loss function

$$\mathbf{g}(\boldsymbol{\beta}) = \begin{pmatrix} \frac{\partial L}{\partial \beta_0} \\ \frac{\partial L}{\partial \beta_1} \end{pmatrix}.$$

Following the procedure in Section 8.3 for taking derivatives of sums, the entries in the gradient function are

$$\frac{\partial L}{\partial \beta_0} = \sum_{i=1}^{5} \left(\beta_0 + \beta_1 x_i - y_i^{\text{true}}\right)$$

and

$$\frac{\partial L}{\partial \beta_1} = \sum_{i=1}^{5} \left(\beta_0 + \beta_1 x_i - y_i^{\text{true}}\right) x_i.$$

The update rule for gradient descent is

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \alpha\, \mathbf{g}(\boldsymbol{\beta}^{(k)}),$$

remembering again that we are iterating over the parameters $\boldsymbol{\beta}$, not $\mathbf{x}$. We need an initial guess for the parameters, and lacking any insight from the problem we will choose the zero vector: $\boldsymbol{\beta}^{(0)} = \mathbf{0}$. Using a step size $\alpha = 0.1$, we can begin iterating.

Iteration	Loss	β_0	β_1
0	1.1375	0.0	0.0
50	0.0599557	0.0979449	1.04396
100	0.0544082	0.0332741	1.17936
150	0.0542539	0.0224895	1.20194
200	0.0542496	0.0206910	1.20571
250	0.0542495	0.0203911	1.20634
300	0.0542495	0.0203411	1.20644
350	0.0542495	0.0203327	1.20646
400	0.0542495	0.0203313	1.20646
450	0.0542495	0.0203311	1.20646
500	0.0542495	0.0203311	1.20646

Gradient descent found the same parameters as pseudo-inversion for our linear regression example. This is expected since the least-squares problem is convex and has a unique solution. Gradient descent terminates at a local minimum, and all local minima are global minima for convex problems. If gradient descent works so well, why don't we use it on linear least-squares problems in practice? There are two reasons:

1. Statistics of the parameters (p-values, confidence intervals, etc.) are computed from matrices that are also used to find the pseudoinverse. If we used

gradient descent on a linear regression problem, we would need to perform most of the pseudoinverse calculation anyway.

2. Gradient descent requires an initial parameter guess and a value for the step size hyperparameter. Both of these can be avoided by pseudoinversion.

Still, gradient descent plays an important role in regression. In the coming chapters we will introduce two powerful extensions to linear models — logistic regression and regularized regression — that cannot be fit using pseudoinversion. We will use gradient descent to parameterize these models.

12.3 Termination Conditions

Using the update rule (equation (12.1)) we can always find a next estimate of the input that is closer to a local argmin. In both of the previous examples, however, the iterates became so close to the local argmin that iterates stopped moving. Gradient descent conveniently self-terminates once we find a local minimum. To see why, consider the update rule $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha\, \mathbf{g}(\mathbf{x}^{(k)})$. Imagine if $\mathbf{x}^{(k)}$ is exactly a local argmin. The gradient is flat in all direction at a local minimum, so $\mathbf{g}(\mathbf{x}^{(k)}) = \mathbf{0}$. Therefore, the update rule says that

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \alpha\, \mathbf{g}(\mathbf{x}^{(k)}) \\ &= \mathbf{x}^{(k)} - 0 \\ &= \mathbf{x}^{(k)}\end{aligned}$$

and the next iterate remains at the local argmin.

Self-termination is convenient, but it is rarely practical. Gradient descent terminates only when the gradient is exactly zero, so termination requires we land exactly on the a local argmin (or at least land within the precision of the computer). Usually we're not so lucky. Also, self-termination requires the gradient to shrink nicely to zero in a small region around the local argmin. As we will see in Chapter 14, there are many important optimization problems where such continuity is not guaranteed.

Alternatively, we can terminate gradient descent when we are satisfactorily near a local minimum. Two criteria are commonly used to halt gradient descent:

1. **Iterate convergence**. As we approach a local minimum and the gradient shrinks, the steps between iterates should also decrease. One strategy is to terminate gradient descent if the iterate $\mathbf{x}^{(k+1)}$ is very close to $\mathbf{x}^{(k)}$. Formally, we define some small value ϵ and stop iterating if $\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right\| < \epsilon$. Any

norm (1-norm or 2-norm) will work, but a common choice is the max-norm (∞-norm) that measures the maximum distance between any corresponding elements in $\mathbf{x}^{(k+1)}$ and $\mathbf{x}^{(k)}$. The max-norm will keep gradient descent running if at least one dimension is still moving.

2. **Objective convergence**. The value of the objective function should also stop changing as gradient descent approaches a local minimum. A second termination strategy is to stop iterating when

$$\left|f(\mathbf{x}^{(k+1)}) - f(\mathbf{x}^{(k)})\right| < \epsilon$$

for some small number ϵ. Terminating based on the objective value avoids the need to choose a norm since the objective function is always scalar-valued. However, a large change in the input to a function may lead to a small change in the objective value, so the objective convergence method may cause gradient descent to terminate while the iterates are still changing.

Both iterate and objective convergence have weaknesses. For example, objective convergence should not be used if the objective function is very flat, and iterate convergence can fail to terminate if the gradient is discontinuous near the local minimum. Some software packages apply both tests and terminate if either the iterates or objective values converge.

If the gradient is zero at a local minimum, why can't we use the magnitude of the gradient as a termination test? There are at least two reasons to avoid testing the gradient. First, it is common to use gradient descent with only an estimate of the gradient, and this estimate may not vanish completely at the local minimum. Second, terminating based on the gradient assumes the gradient is continuous and defined near the local argmin. As we'll see later, we can still solve optimization problems with gradient descent even if none of these conditions hold.

12.4 * Step Sizes

Walking downhill via gradient descent will bring us closer to a local minimum, but we also need to stop walking once we reach the bottom. The process of stopping at a local minimum is called *convergence*. We need some assurance that gradient descent will converge. We want our steps to become smaller as we approach the local minimum and disappear completely if we happen to arrive exactly at the local minimum. Conversely, if we are far away from the local minimum, we want to take large steps so we reach the local minimum in a reasonable amount of time. Think about finding a parking space for your car. You drive relatively quickly up

and down the lanes of the parking lot until you close in on an open spot; then you slow down considerably to avoid hitting adjacent cars when entering the spot.

The step taken during each iteration of gradient descent is the product of two parts: a step size α and the direction $\mathbf{g}$. We have two methods of altering the length of the step at each iteration: 1.) decrease the step size α as the iterations increase, or 2.) decrease the magnitude of the gradient. Let's consider each method, starting with the step size.

12.4.1 * Step Size Scheduling

Gradient descent moves us closer to a local minimum at each iteration, so decreasing the step size α at each iteration will force us to take smaller steps as we get closer to the local minimum. The problem is timing these two events. We cannot say in advance how many iterations we need to get close to the local minimum. Decreasing the step size early on will slow convergence, but decreasing it too late can make us overshoot or zigzag around the local minimum.

Changing the step size requires the creation of a *step size schedule*. The schedule is simply a method that tells the gradient descent algorithm what step size to use at each iteration. A step size that follows a schedule requires a slight change in notation for the update rule at each iteration:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(x_k).$$

Rather than have a single, constant value α for all iterations, the step size at iteration k is α_k. The value of α_k is determined by the step size schedule.

There are many methods for constructing step size schedules, and the optimal schedule depends heavily on the function f to be minimized. While there is no universally best schedule, there are some properties of schedules that guarantee convergence. Remember that gradient descent goes on forever, moving us ever closer to a local minimum but never exactly there. We usually terminate gradient descent after a finite number of iterations, but we could let it run forever using infinitely many step sizes from the schedule. One method for ensuring convergence constrains the sums of all the step sizes in the schedule. The two constraints are

$$\sum_{k=0}^{\infty} \alpha_k = \infty \tag{12.2}$$

and

$$\sum_{k=0}^{\infty} \alpha_k^2 < \infty. \tag{12.3}$$

Equation (12.2) forces our step sizes to be large enough so that gradient descent does not stop prematurely before we are near the local minimum. The infinite sum of all the step sizes in the schedule must diverge to infinity.

It's easy to find a sequence of step sizes that sum to infinity — a constant step size would satisfy equation (12.2). We also need the step sizes to decrease as each iteration moves us closer to the local minimum. One method is to have the step size approach zero, and equation (12.3) ensures the decrease is rapid enough for convergence. Taken together, equations (12.2) and (12.3) define a "sweet spot" for step sizes schedules. The step sizes must be large enough so their sum diverges, but small enough so their squared sum converges. One step size schedule that satisfies the convergence criteria is $\alpha_k = 1/k$ (with $\alpha_0 \equiv 1$ to avoid dividing by zero). The first seven step sizes from this schedule are shown in the table below.

iteration	0	1	2	3	4	5	6
step size (α_k)	1.000	1.000	0.500	0.333	0.250	0.200	0.167

The series $\alpha_k = 1/k$ is the harmonic series, and the sum of this series diverges. The sum of the squares of the harmonic series converges, although the exact value it converges to is not important.

You might have noticed that if the sum of the step sizes is infinite but the sum of the squared step sizes is finite, then $\alpha_k^2 < \alpha_k$, at least when k is large. This implies that eventually $\alpha_k < 1$, and in practice it is rare to start with step sizes larger than one.

Chapter 13

Logistic Regression

Let's return to the problem of binary classification where a feature vector $\mathbf{x}$ is used to predict the class y of a sample. We've already used the Support Vector Machine to solve the binary classification problem. Recall that the SVM uses optimization to find a hyperplane $\mathbf{a} \cdot \mathbf{x} = b$ that separates the two classes. Although the SVM works well, it is difficult to understand how the algorithm predicts the class of new data. We could try to examine the support vectors that lie nearest the separating hyperplane, but in general we cannot directly interpret SVM models.

By contrast, we've seen how straightforward it is to interpret linear statistical models. We are able to assign meaningful interpretations to the fitted coefficients, and the relative importance of the predictor variables is quantified by the statistical outputs of the `fitlm` function. Ideally we would use linear models for the binary classification problem. However, there are two problems:

- The predictions of a classification algorithm are binary, while linear models make continuous predictions.
- Even if we force a linear model to make discrete predictions, we must also force the outputs of a linear model to stay within the set of classes.

In this chapter we develop a variant of linear regression called *logistic regression*. Logistic regression uses a linear model to predict binary outcomes. Rather than predict the class of a sample directly, a logistic regression model predicts the probability that the sample is in each class. We will show how a *link function* can be used to map the output of a linear model into a bounded range, like the interval $[0, 1]$ for probabilities.

13.1 Predicting Odds

You're probably familiar with probabilities as the long-run expectation of an uncertain process. Logistic regression uses a related concept called the *odds*. You may have used the term "odds" interchangeably with "probability," but they are not the same. Let's assume that a random variable y has two possible outcomes, 0 and 1. The odds of y is the ratio of the probability that y equals 1 to the probability that y equals 0, or

Pun intended.

$$\text{odds}(y) = \frac{P(y=1)}{P(y=0)}.$$

For example, if $\text{odds}(y) = 2$ then the probability that $y = 1$ is twice as large as the probability that $y = 0$. We can convert between probabilities and odds by remembering that probabilities sum to one, or P(y=0) + P(y=1) = 1. Then

Odds are usually expressed as a proportion, so an odds of 2 is written as 2:1, or "two to one".

$$\text{odds}(y) = \frac{P(y=1)}{P(y=0)} = \frac{P(y=1)}{1-P(y=1)} \Rightarrow P(y=1) = \frac{\text{odds}(y)}{1+\text{odds}(y)}.$$

The odds function lives interval $[0, \infty)$. The odds of y become infinite as the probability that $y = 1$ increases. The odds of y go to zero as the probability that $y = 0$ increases. This means that the logarithm of the odds, or the "log odds" is a continuous value in the interval $(-\infty, \infty)$, which is the same range as the predictions of a linear model. We can build a binary classifier by using a linear model to predict the log odds of the response variable y, i.e.

Some people go further and refer to the log odds as the "lods".

$$\log(\text{odds}(y)) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

The function $\log(\text{odds}(y))$ is called the *logit* function. Because it links the response variable to the linear models, we refer to the logistic (and other similar functions) as *link functions*.

To summarize:

$$\begin{aligned} y &\in 0 \text{ or } 1 \\ P(y=1) &\in [0,1] \\ \text{odds}(y) &\in [0,\infty) \\ \log(\text{odds}(y)) &\in (-\infty,\infty) \end{aligned}$$

13.2 From Odds to Probabilities

Log odds can be predicted using linear models, but it is difficult for most people to interpret the odds, much less their logarithm. Ideally we would have our logistic regression model predict probabilities. The logistic regression model from above was

$$\log(\text{odds}(y)) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

Exponentiating both sides to gives

$$\text{odds}(y) = e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p} \equiv e^t$$

where the placeholder t equals the output of the linear model. We can solve for the probability that y equals 1 using the relationship between probabilities and odds.

$$\begin{aligned} P(y = 1) &= \frac{\text{odds}(y)}{1 + \text{odds}(y)} \\ &= \frac{e^t}{1 + e^t} \\ &= \frac{1}{1 + e^{-t}} \end{aligned}$$

For the last step we divided both the numerator and denominator by e^t.

This function is called the *logistic* or *sigmoid* function, and its shape is shown in Figure 13.1. The output of the function is restricted to the interval $[0, 1]$ even though the inputs are unbounded. The bounded output makes it possible to interpret the outputs of the logistic function as probabilities.

Making predictions with logistic models is a two-step process. First, we use a linear model to predict the placeholder value t. The value of t is used to calculate the probability that the response is equal to one. If we were interested in classifying the response, we would say that $y = 1$ if $P(y = 1) > 0.5$ and choose $y = 0$ otherwise. Note that the point $P(y = 1) = 0.5$ occurs when $t = 0$. When classifying with a logistic regression model, our response prediction switches from class 0 to class 1 when the output of the linear model $t = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$ switches from negative to positive.

Logistic regression is used for binary classification, so the model should alternate between predicting class 0 and class 1. The sigmoid shape is a compromise; it is smooth and continuous but still transitions rapidly from 0 to 1. The smoothness of the logistic function (and its convenient derivative) makes it easier to fit logistic regression models by gradient descent.

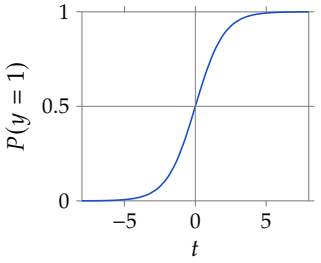


Figure 13.1: The logistic function.

13.3 Example: Predicting the risk of Huntington's Disease

Huntington's Disease is an inherited genetic condition caused by repeated CAG sequences in the Huntingtin (*HTT*) gene. Too many CAG repeats create a "glutamine knot" in the protein, causing toxic protein aggregates in neurons. Symptoms of Huntington's appear later life, and an individual's risk for developing the disease correlates with the number of CAG repeats.

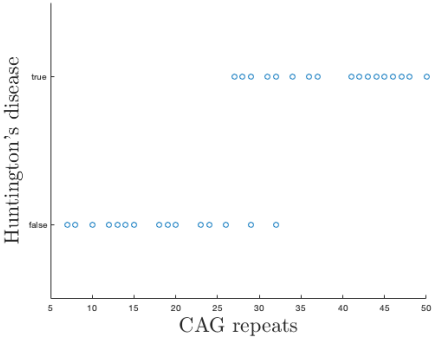
Source: Walker FO. Huntington's disease. *The Lancet*. 2007: **369**, (9557), 218–228.

# of CAG Repeats	Disease Outcome
< 28	Not affected.
28–35	Increases risk.
36–40	Affected; some offspring affected.
> 40	Affected; all offspring affected.

Let's build a model to predict the probability of developing Huntington's based on the number of CAG repeats. The response variable is binary (Huntington's disease or not) and the predictor variable is continuous (the number of CAG repeats in the *HTT* gene). To train the model we counted the number of CAG repeats in 50 individuals with and without the disease.

MATLAB code

```
load huntington.mat
scatter(hunt.CAGs,categorical(hunt.disease))
xlabel('CAG repeats',axargs{:})
ylabel("Huntington's disease",axargs{:})
```



We see from these data that predicting disease status with low (<25) or high (>35) CAG repeats is straightforward. However, there is a region between 25 and 35 CAG repeats where disease status is ambiguous. Let's build a logistic regression model to predict Huntington's status. We use the MATLAB function `fitglm`, for "fit generalized linear model". The `fitglm` function is similar to `fitglm`; the first argument is a table of data, and the second argument is a formula describing the model. However, `fitglm` can use a wide range of link functions and datatypes when fitting linear models. For logistic regression using binary responses we need to specify the logit link function and a binomial distribution.

MATLAB code

```
model = fitglm(hunt,'disease ~ CAGs','link','logit', ...
         'Distribution','binomial')
```

MATLAB output

```
model =
Generalized linear regression model:
  logit(disease) ~ 1 + CAGs
  Distribution = Binomial

Estimated Coefficients:
              Estimate       SE       tStat     pValue
              ________    _______    _______    ________
(Intercept)   -14.032      5.7832    -2.4263    0.015252
CAGs.          0.50558    0.20395    2.4789    0.013179

50 observations, 48 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 55, p-value = 1.18e-13
```

Remember that the model we're fitting is

$$\log(\text{odds}(\text{disease})) = \beta_0 + \beta_1[\text{CAGs}].$$

We know the best fit values of β_0 and β_1 from the output of the `fitglm` model:

$$\log(\text{odds}(\text{disease})) = -14.032 + 0.50558[\text{CAGs}].$$

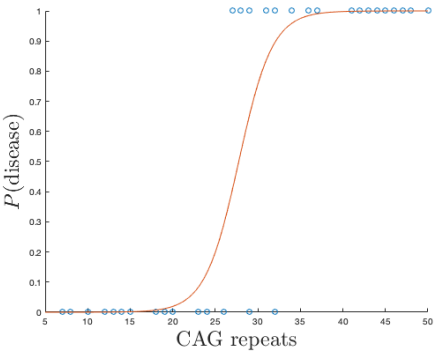
We can also rewrite this model to predict the probability of having Huntington's disease

$$P(\text{disease}) = \frac{1}{1 + e^{-14.032+0.50558[\text{CAGs}]}},$$

which we plot below along with the training data.

MATLAB code

```
scatter(hunt.CAGs,hunt.disease)
hold on
cag_range = linspace(5,50,100);
beta = model.Coefficients.Estimate;
plot(cag_range, 1./(1+exp(-(beta(1)+beta(2)*cag_range))))
hold off
xlabel('CAG repeats',axargs{:});
ylabel('$$P(\mathrm{disease})$$',axargs{:});
```



When the output of the linear model is zero,

$$P(y = 1) = \frac{1}{1 + e^0} = \frac{1}{2}$$

.

We are often interested in the point where $P(\text{disease}) = 0.5$, as this is the threshold number of CAG repeats where a person is equally likely to have or not have Huntington's. The logistic function reaches its midpoint when the linear model moves from negative to positive. Thus we can simply solve for when $\beta_0 + \beta_1[\text{CAGs}] = 0$.

$$-14.03 + 0.51[\text{CAGs}] = 0 \Rightarrow [\text{CAGs}] = 14.03/0.51$$
$$\approx 28 \text{ CAG repeats}$$

13.4 Interpreting coefficients as odds ratios

The coefficients of the linear part of a logistic regression equation are not directly interpretable. The coefficients describe how the linear model changes given a unit change in the input variables, but the outputs of the linear model undergo a nonlinear transformation before becoming a probability. Instead, we interpret logistic regression models by calculating the change in odds that accompany a unit change in an input variable. This change is odds is called the *odds ratio*. For example, we can define the odds ratio that corresponds to increasing variable x_i by 1 as

$$\text{odds ratio}(x_i) = \frac{\text{odds}(x_i + 1)}{\text{odds}(x_i)}.$$

Let's calculate the odds ratio for Huntington's disease that accompanies an increase of one CAG repeat.

$$\begin{aligned}\text{odds ratio}([\text{CAGs}]) &= \frac{\text{odds}([\text{CAGs}]+1)}{\text{odds}([\text{CAGs}])} \\ &= \frac{e^{\beta_0+\beta_1([\text{CAGs}]+1)}}{e^{\beta_0+\beta_1[\text{CAGs}]}} \\ &= \frac{e^{\beta_0}e^{\beta_1[\text{CAGs}]}e^{\beta_1}}{e^{\beta_0}e^{\beta_1[\text{CAGs}]}} \\ &= e^{\beta_1}\end{aligned}$$

Since $\beta_1 = 0.51$ in out model, having one more CAG repeat increases the odds of developing Huntington's disease by $e^{0.51} = 1.67$-fold. For any logistic regression model, the odds ratio for variable x_i is the exponential of the corresponding coefficient β_i.

$$\text{odds ratio}(x_i) = \frac{\text{odds}(x_i+1)}{\text{odds}(x_i)} = e^{\beta_i}$$

If β_i is negative the odds ratio e^{β_i} will be less than one and the odds will decrease.

You may have heard news reports that "doing X increases your risk of Y". Researchers performing this type of study often use logistic regression models to predict the odds of developing condition Y based on input variable X. The reported increase in risk is simply the odds ratio associated with the coefficient of X.

13.5 Fitting Logistic Regression Models

We fit a logistic regression model using a set of n training point $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ is a vector of input features and y_i is a binary output variable (either 0 or 1). The output of the logistic regression model is $P(y_i)$, the probability that $y_i = 1$ given an input $\mathbf{x}_i$. A perfect model would predict that

$$\begin{aligned}P(y_i) &= 1 \quad \text{when } y_i \text{ is } 1 \\ P(y_i) &= 0 \quad \text{when } y_i \text{ is } 0\end{aligned}$$

A common loss function for logistic regression is

$$L(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=1}^{n}\left[-y_i \log P(y_i) - (1-y_i)\log(1-P(y_i))\right] \tag{13.1}$$

where $P(y_i)$ is the output of the logistic function

$$P(y_i) = \frac{1}{1 + e^{-t}}, \quad t = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

Let's examine the summand of the loss function

$$-y_i \log P(y_i) - (1 - y_i) \log(1 - P(y_i)).$$

When $y_i = 0$, this expression reduces to $-\log(1 - P(y_i))$, which reaches a minimum of zero only when $P(y_i) = 0$. (When $P(y_i) \neq 0$, $1 - P(y_i)$ is less than one, so $-\log(1 - P(y_i))$ is a positive, nonzero value). The other option is that $y_i = 1$, in which case the loss summand becomes $-\log P(y_i)$. This expression is minimized when $P(y_i) = 1$. Both cases are what we want in a loss function — to minimize the loss we set $P(y_i) = 0$ when $y_i = 0$ and $P(y_i) = 1$ when $y_i = 1$.

Remember that $P(y_i)$ is a probability, so it must lie in the range $[0, 1]$. Also, the logarithm of 1 is 0, the logarithm of a number smaller than 1 is negative; and the logarithm of 0 approaches negative infinity.

The loss function in equation (13.1) isn't the only loss function that would work for logistic regression. The function $P(y_i)^{1-y_i}(1 - P(y_i))^{y_i}$ is also minimized when the probability $P(y_i)$ matches the value of y_i. However, this loss function also has a maximum value of one for each training point. We prefer that our loss functions be unbounded above so that no matter how terrible our model is, making it worse will always increase the loss. Said another way, we always want to distinguish between "bad" solutions and "very bad" solutions; otherwise, if we started training at a very bad solution, there would be little change in the loss by improving to a merely bad solution. Since the derivative of the loss function drives our training updates, any plateau in our loss function will decrease the training rate.

The loss function in equation (13.1) did not appear out of thin air. Minimizing (13.1) is equivalent to maximizing the likelihood of the model predicting the training values. Actually, it maximizes the log-likelihood since taking the logarithm of the likelihood doesn't change the argmax but makes the function easier to compute.

13.5.1 Gradient Descent

Our loss function is nonlinear and must be minimized using gradient descent or another iterative approach. We first need to compute the gradient of the loss with respect to each parameter in $\boldsymbol{\beta}$.

$$\mathbf{g}(\boldsymbol{\beta}) = \begin{pmatrix} \frac{\partial L}{\partial \beta_0} \\ \vdots \\ \frac{\partial L}{\partial \beta_p} \end{pmatrix}$$

The loss function L depends on the probabilities P, which depend on the output t of the linear model, which depends on the parameters $\boldsymbol{\beta}$. This nested structure is

a perfect opportunity to use the chain rule to compute the entries of the gradient. For a single parameter β_j

$$\frac{\partial L}{\partial \beta_j} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L_i}{\partial P_i} \frac{\partial P_i}{\partial t_i} \frac{\partial t_i}{\partial \beta_j}.$$

Let's compute each of the righthand side derivative in turn. For the loss function, it is simpler to compute the derivative separately when $y_i = 0$ and $y_i = 1$.

$$\frac{\partial L_i}{\partial P_i} = \begin{cases} \frac{1}{1-P(y_i)}, & \text{when } y_i = 0 \\ -\frac{1}{P(y_i)}, & \text{when } y_i = 1 \end{cases}$$

The probability P is the output of the logistic function, which has a convenient derivative.

$$\begin{aligned} \frac{\partial P_i}{\partial t_i} &= P(t_i)(1 - P(t_i)) \\ &= \frac{1}{1 + e^{-t_i}} \frac{1}{1 + e^{t_i}} \end{aligned}$$

Finally, we compute the derivative of the linear model $t = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$, being careful to handle the special case of the intercept ($i = 0$).

$$\frac{\partial t_i}{\partial \beta_j} = \begin{cases} 1, & \text{when } j = 0 \\ x_{ij}, & \text{when } j > 0 \end{cases}$$

The notation x_{ij} requires an explanation. Remember that we have n pairs of training data $(\mathbf{x}_i, y_i)$. The value x_{ij} is the jth entry in the ith feature vector of the training set.

With the derivative in hand, all we need to begin gradient descent is an initial guess for the parameter vector $\boldsymbol{\beta}$. A convenient guess is $\boldsymbol{\beta} = \mathbf{0}$. Setting all parameters equal to zero makes the output of the linear model $t = 0$ for every training point. The logistic function takes the value 0.5 when $t = 0$, so a zero initial guess begins right in the middle with a prediction that y_i is equally likely to be 0 or 1 for every training point. Unless we have some prior information that says otherwise, guessing 0 or 1 with equal probability is a fine place to start.

An accompanying MATLAB workbook implements gradient descent to fit the Huntington's model from earlier in this chapter.

Chapter 14

Bias, Variance, and Regularization

14.1 Learning vs. Memorizing

Just because we trained a model does not mean it learned anything useful from the data. We need to *test* the model to assess its accuracy. We test a model using data that were not used for training so we can distinguish between *learning* and *memorizing*. Memorizing occurs when a model simply remembers the correct outputs for each of the training inputs. When shown a training input again, the model can produce the correct result. However, if the model is given an input that was not included in the training set (i.e. an input that it has not memorized), the model cannot predict the correct value. By contrast, learning occurs when a model can predict correctly without memorizing. Models learn by finding relationships in the input features that hold information about the correct output. A model that learns well can usually make good predictions on new data since the feature relationships are still valid. Models that predict accurately on data that were not part of the training data are said to *generalize*.

Models that cannot generalize well are not useful. Such models have only memorize the correct answers for the training data, but we already know the answers to the training data! Testing our models with our training data cannot distinguish between models that memorize and models that learn. We need separate data that have not been shown to the model. Models that memorize will perform poorly on these data, but models that learned will do better.

All models more accurately predict the results of their training data compared to the testing data, so do not be alarmed if the testing accuracy is lower than the training accuracy. The reduced performance on testing data is called the *generalization gap*, and it affects all models even if they are not memorizing. In this

chapter we will discuss why the generalization gap occurs and how to reduce it. We will also discuss methods to assess both the training and testing accuracy of our models.

14.2 Holdout

The simplest approach for validating a model is called *holdout*. Holdout removes a minority of the training data and sets it aside for testing. The holdout set can be used for validation since it was not used during training.

There are no rules for how much of the training data should be removed for holdout. The number of holdout observations, not the fraction of the entire dataset, is most important. For example, imagine if we only included two points in our holdout set when training a binary classifier (like logistic regression). There are only three possible outcomes when testing our model: 0/2, 1/2, or 2/2, making our accuracy 0, 0.5, or 1.0. This is clearly a crude assessment of our model's accuracy. If the holdout dataset includes n observations, the resolution of our accuracy estimate is $1/n$. A holdout set with 10 observations can only estimate the accuracy to within 0.1, and this is an upper bound. Small holdout sets are hampered by stochasticity. Testing points that happen to be similar to a training data are easier for a model to predict correctly. A few "good" or "bad" points can have a big effect if the holdout set is small.

It is common in machine learning to refer to accuracy as a fractional value, not as a percentage.

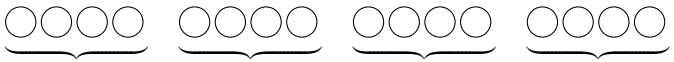
Perhaps counterintuitively, larger training sets require a smaller *fraction* of data be reserved for holdout. A training set with 20 observations could require 50% or more the data be set aside for holdout, and even then the validation accuracy would have a precision of no less than 0.1. By contrast, the Netflix Prize contest, a community-based competition to predict personalized ratings for movies, used a training dataset with over 100 million observations. The final validation set included only 1.36% of these points to award a $1,000,000 prize to the winning team.

Although the Netflix Prize holdout set contained over one million observations, the top two teams tied for accuracy. The tiebreaker went to the team that submitted 20 minutes before the other.

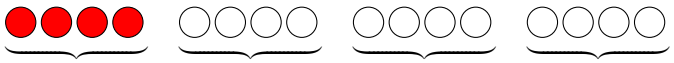
Holdout works well for large datasets where only a small fraction of the data need to be excluded from training. In small datasets a large fraction of data need to be removed for validation. **After validation, the holdout data can be added back to the training set before training a final model.** Thus the holdout data are not "lost", but for small datasets with large holdout the validation accuracy will be a poor estimate of the accuracy of the final model trained with the entire dataset. For large datasets, the expense of retraining the model often outweighs the gain in accuracy from including the small fraction of holdout points.

14.3 Cross Validation

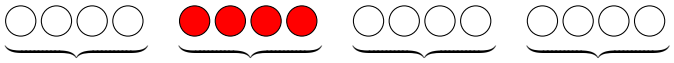
Cross validation is an alternative to holdout. In cross validation, all points in the dataset are used for training and testing, but never at the same time. Cross validation begins by splitting the dataset into a set of k groups of roughly equal size. Each group of data is called a *fold*, and points are randomly assigned to the folds. For example, at dataset with 16 points could be divided into $k = 4$ folds.



To begin cross validation, one of the folds is set aside for validation, similar to holdout. A model is trained using the remaining $k - 1$ folds and tested against the holdout fold.



Next we put fold 1 back into the training set and set aside fold 2 for testing. Then we re-train our model using folds 1, 3, and 4 and validate with fold 2.



This process continues $k = 4$ times, with the final model trained on folds 1–3 and validated with the data in fold 4. The final step is to average the accuracies across all k folds. This average is reported as the final accuracy, and a full model can be trained using all of the data.

The advantage of k-fold cross validation is that every point in the dataset is used for testing, so the method is not sensitive to which data are selected for holdout. However, the method is still stochastic as the accuracy of each model depends on how the data are randomly assigned to the folds. A k-fold cross validation requires training k separate models in addition to the final model with all of the data. This might be costly for very large datasets, so cross validation is more common in small- to medium-sized problems.

14.3.1 Leave-one-out Cross Validation

There is no rule for determining the number of folds (k) for a cross validation. Smaller datasets benefit from higher values of k since fewer points are held out

at a time and training multiple models is not computationally prohibitive. One extreme of k-fold cross validation occurs when k equals the number of points in the dataset. In this case each fold contains only a single point, hence the name *leave-one-out* cross validation. Leave-one-out is the most computationally demanding strategy for cross validation, as a new model must be trained for each point in the dataset. However, leave-one-out provides the best estimate of the accuracy of the final model trained with all the data. Each of the validation sub-models is trained with all but one point, so these models closely resemble the performance of a model trained with all of the data. Since each fold contains one point, there is no randomness to the validation procedure if the training algorithm is deterministic.

Leave-one-out is abbreviated "L1O". Assigning two points per fold is called leave-two-out (L2O) cross validation.

14.4 Bias vs. Variance

Cross validation measures the accuracy of our models. We need to understand why our models are inaccurate before we can develop strategies to improve them. As a reminder, we care most about generalization accuracy — the ability to predict results that were not included in the training set. The fundamental source of all model inaccuracies is limited data. Take, for example, the data in Figure 14.1. The center panel shows a continuous function that we are trying to learn using six data points. The other eight panels show six randomly sampled points from the center function. It would be difficult to estimate the original function using any of these subsets alone. Any model fit to a subset of data would not generalize well to parts of the function that were not we represented in the training data.

Most datasets contain more than six training points, but remember that the function in Figure 14.1 is one-dimensional. Many of our machine learning methods are applied to high-dimensional data, so the *density* of training data may be lower than the sampling shown in the Figure 14.1. Data acquisition is enormously expensive, so we are often left with far less training data than we would like.

A model's error can be divided into two sources. The first source is *bias*. Bias appears when the model underfits the data because the model lacks the flexibility to match the underlying system. Imagine fitting a model that predicts how many text messages a person sends per day based on their age. Even if we had millions of training data, we could not possibly predict everyone's texting patterns using such a simple model. A teenager with a phone might text a lot, but not all teenagers have phones. Our model's predictions will have high bias since adding more data or switching to a new sample will not improve the predictions. Bias is robust to subsampling, meaning we will fit similar models to different datasets.

You can see high-bias models in the first column of Figure 14.2. Each model is a two-parameter linear model ($y = \beta_0 + \beta_1 x$) fit to six points sampled from the

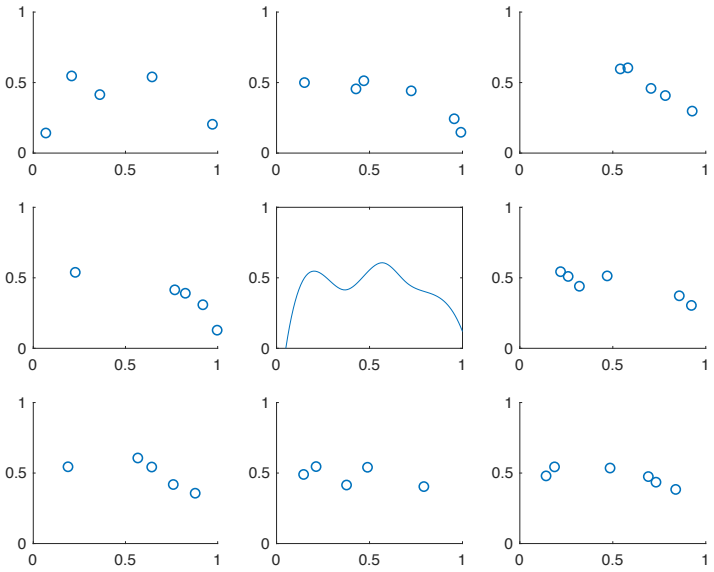


Figure 14.1: Samples of six points vary in their approximations of the function in the center panel.

true function. Although the six-point samples vary widely, the fit models and their generalization error are similar. Although the models are not sensitive to changes in the dataset, they do not approximate the function well. After all, we cannot expect a purely linear model to reproduce the nonlinear function shown at the top of the figure.

The other source of generalization error is model *variance*. Models with high variance are overfit to the data. High variance can been seen in right column of Figure 14.2. These are six-parameter curvilinear models ($y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_5 x^5$). Notice how well these models predict the training data (blue circles). In fact, since there are six parameters and six data points, these models predict the training data exactly! However, you should also notice how poorly some of these models would generalize. The top model predicts a huge spike between 0.5 and 1, and the middle model predicts an increase, not decrease, near 1.0. Neither of these model match the true function at the top of the figure. Models with high variance have two characteristics: 1.) they are good at memorizing training data, and 2.) they are highly sensitive to the training data. The two-parameter models on the left are similar for all three datasets, but the six-parameter models have very

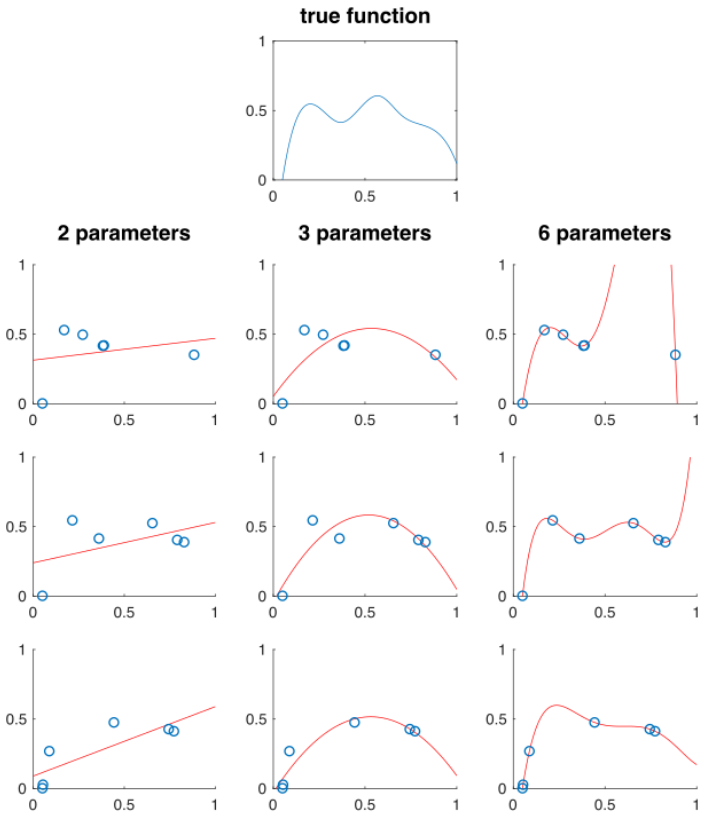


Figure 14.2: A model's bias and variance depend on the number of parameters.

different shapes.

The key to fitting models is to balance bias and variance. We want models that are flexible enough to match the true system (low bias) but not overly sensitive to the specifics of our training data (low variance). A three-parameter quadratic model ($y = \beta_0 + \beta_1 x + \beta_2 x^2$) achieves this balance for the function in Figure 14.2 (center column). The models are not exact, but each model resembles the true function regardless of which points are included in the training set. This example shows overly simple and overly complex models can both lead to poor generalization.

14.5 Regularization

Figure 14.2 is a toy example. We knew the true function and were able to collect multiple subsamples to test our model's accuracy. This allowed us to adjust the number of parameters until we found a model that balanced bias and variance. In reality, we won't be able to tune and retrain our model, partly because we don't know the true function we are trying to replicate.

A more general strategy is to start with a model that has more parameters than necessary and try to minimize overfitting. This approach is called *regularization*, and it relies on an observation that overfit models tend to have many parameters with large magnitudes. Keeping parameters small during training tends to produce models that generalize better. We regularize an algorithm by penalizing it whenever the model's parameters get too big. Formally, this is accomplished by adding a regularization term to the objective function.

14.5.1 The LASSO

Let's use regularization to prevent overfitting of a linear model. Linear models are trained using a quadratic loss function

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \left(y_i^{\text{pred}} - y_i^{\text{true}} \right)^2.$$

Here $\boldsymbol{\beta}$ is a vector of parameters. If the linear model has input features $\mathbf{x}$, the output $y^{\text{pred}} = \mathbf{x} \cdot \boldsymbol{\beta}$. We can substitute this model into our loss function to make it clear that we are minimizing the loss by selecting a set of parameters $\boldsymbol{\beta}$.

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \left(\mathbf{x} \cdot \boldsymbol{\beta} - y_i^{\text{true}} \right)^2$$

Now we can add regularization. Remember that the goal of regularization is to keep the parameters in $\boldsymbol{\beta}$ from becoming too large and overfitting the model. We can add a term to our objective function so our algorithms tries to minimize both the total loss and the magnitudes of the parameters.

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \left(\mathbf{x} \cdot \boldsymbol{\beta} - y_i^{\text{true}}\right)^2 + \lambda \sum_{i=1}^{p} |\beta_i| \tag{14.1}$$

The regularization term adds up the magnitudes of the parameters. This sum is weighted by a hyperparameter λ that balances the two objectives: minimize loss or minimize the parameter magnitudes. The hyperparamter λ can be any non-negative value. Setting $\lambda = 0$ eliminates all regularization, making Equation (14.1) equivalent to normal least-squares regression. Setting λ to a large value will focus most of the algorithm's attention on keeping the parameters small. If λ is large enough, the algorithm will ignore the loss entirely and set all of the parameters to zero. There is no definite rule for selecting a value for λ. Like all hyperparameters, it must be tuned for each problem to maximize performance. Cross validation can be used to ensure the value of λ promotes generalization of the trained model.

The regularized form of linear regression in Equation (14.1) is called the "least absolute shrinkage and selection operator", or LASSO. The effects of regularization can be seen in Figure 14.3. When $\lambda = 0$, the LASSO is equivalent to standard linear regression, so a six-parameter model overfits random samples of six data points. Adding regularization ($\lambda = 0.001$) decreases the variance of the models, as seen by the similar shapes of the models in the center column. Increasing the regularization ($\lambda = 0.01$) further reduces the variance, and all the models in the right column have the same overall shape. However, we are starting to see the effects of over-regularizing the model. Regularization penalizes large parameters, so the models are beginning to underpredict the data. Said another way, high regularization reduces the variance so much that we begin to see increased model bias.

The goal of regularization is to improve generalization, but this goal causes decreased accuracy on the training data. The first column of Figure 14.3 fits the training data exactly, but the curves are clearly overfit and do not resemble the true function. The regularized models in columns two and three do not predict the training exactly, although they will generalize better since they more reliably predict data outside the training set. Do not be alarmed if adding regularization decreases the training accuracy of your model. You are reducing your model's ability to memorize in hopes that it will generalize better.

The example in Figure 14.3 uses regularization to keep a polynomial model from overfitting data. This is only one application of the LASSO. As discussed in

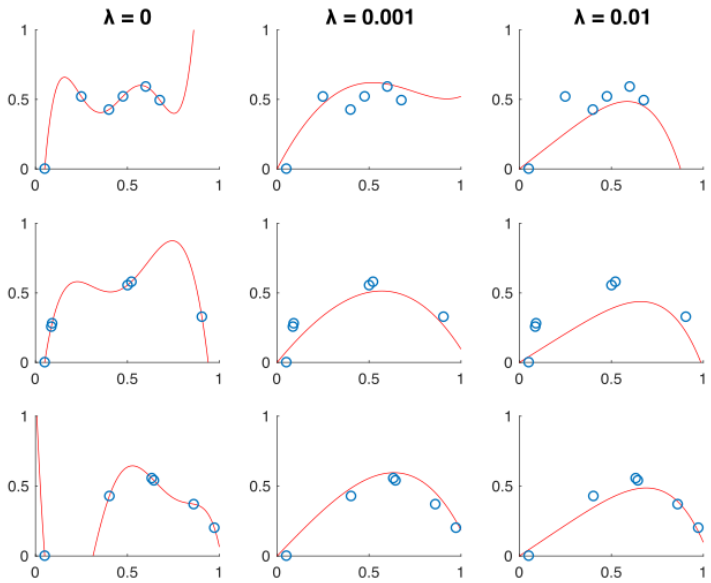


Figure 14.3: Increasing values of λ decrease the variance of linear models trained by the LASSO.

section 14.5.2, the absolute value in the LASSO leads to solutions where many of the parameters are set to zero. This *selection* property is useful when we have far too many input features for our model. For example, many genome-wide association studies use logistic regression to compute the risk of a disease given mutations (SNPs) in a genome. Every humans has thousands of SNPs, so any regression model trained with all SNPs would be vastly overfit. A properly regularized logistic regression model will only assign nonzero effect sizes to a small number of informative SNPs, essentially selecting the best SNPs for predicting the risk of the disease. Regularization not only improves generalization; it also helps refine datasets by selecting interesting features.

14.5.2 Generalized Regularization

The LASSO combines regularization and linear regression, but regularization can be applied to any machine learning technique. In general, machine learning algorithms find parameter values that minimize a loss function, so they can be cast as

optimization problems of the form

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} L(\boldsymbol{\beta})$$

where $\boldsymbol{\beta}$ is a vector of parameters and L is a loss function that depends on the parameters. Just as we did with the LASSO, we can add a regularization term to the objective function to give the regularized problem

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} L(\boldsymbol{\beta}) + \lambda \left\| \boldsymbol{\beta} \right\|_k .$$

The hyperparameter λ determines how badly we penalize the parameters based on the sum of their k-norms. The LASSO used the 1-norm (the absolute value) of each parameter, but we can use any norm to measure the size of each parameter. Each norm has a different effect on the regularization, as we explain in the following sections.

0-norm Regularization

The 0-norm measures the number of nonzero parameters. The 0-norm of any nonzero value is equal to 1, or

$$\left\| \beta \right\|_0 = \begin{cases} 0, & \beta = 0 \\ 1, & \beta \neq 0 \end{cases} .$$

The 0-norm is not a true norm since it violates some of the defining properties of norms. However, it is useful for "counting" nonzero values. Regularizing with the 0-norm penalizes the number of nonzero parameters, not their magnitudes. A 0-norm regularized model will minimize the loss function using the smallest number of parameters. This regularization strategy mimics what we accomplished in Figure 14.2 by changing the number of parameters in our curvilinear models. Changing the model from six to two parameters simplified the model but did not place any constraints on the values of the parameters.

The 0-norm would be excellent for regularization except for one problem — it is computationally intractable. Problems that include a 0-norm are essentially discrete problems where parameters are either "on" or "off". Such problems are combinatorially complex and their difficulty increases exponentially with the number of parameters. Large-scale machine learning problems would be impossible to solve if they included 0-norm regularization. Fortunately, the 1-norm approximates many of the features of the 0-norm in a computationally efficient way.

1-norm Regularization

As its name implies, the LASSO performs "shrinkage and selection" on the parameters of a linear model. Shrinkage penalizes parameters based on their magnitude, while selection encourages the model to have nonzero values for only a subset of the parameters. Selection is akin to 0-norm regularization, but the LASSO used only the 1-norm, or absolute value, when penalizing the parameters. It turns out that 1-norm regularization performs both operations. Minimizing the absolute value of parameters forces many of the parameters to zero. While there is no guarantee that minimizing the 1-norm will achieve the least number of nonzero parameters (as the 0-norm would), the 1-norm provides a decent approximation of the 0-norm. At the same time, the 1-norm also penalizes parameters based on their magnitudes, all while remaining computationally tractable!

The combination of shrinkage, selection, and efficiency make the 1-norm a popular choice for regularization. It is ideal for generating *sparse* solution, i.e. models with relatively few nonzero parameters. Despite its strengths, the 1-norm has two disadvantages. First, the derivative of the absolute value is discontinuous at zero, so 1-norm regularized problems can be difficult to solve by gradient descent. Alternative methods like coordinate descent or stochastic gradient descent (Chapter **??**) can be used instead. Second, the solutions to 1-norm regularized problems are not unique. Many different parameter sets can have the same 1-norm penalty.

2-norm Regularization

The 2-norm, like the 1-norm, penalizes based on the magnitude of the model's parameters; however, this is where the similarities end. The 2-norm penalty increases quadratically with the size of a parameter, so 2-norm regularization tries very hard to avoid any large parameters. Instead, the 2-norm encourages solutions with many nonzero parameters with small magnitudes. This is the opposite of the sparse solutions produced by 1-norm regularization. Problems with the 2-norm have unique solutions and continuous derivatives (provided the loss function is continuously differentiable). Both of these features are attractive for large-scale optimization.

Regularization with the 2-norm is also called least-squares or Tikhonov regularization.

The Elastic Net: 1-norm + 2-norm Regularization

Some algorithms try to combine the desirable features of both 1-norm and 2-norm regularization. Both regularization terms are added to the objective function.

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} L(\boldsymbol{\beta}) + \lambda_1 \left\|\boldsymbol{\beta}\right\|_1 + \lambda_2 \left\|\boldsymbol{\beta}\right\|_2$$

Each type of regularization is weighted by separate hyperparameter: λ_1 for the 1-norm and λ_2 for the 2-norm. The relative size of these hyperparameters determines the importance of each type of regularization. Setting $\lambda_1 = 0$ is equivalent to 2-norm regularization, and setting $\lambda_2 = 0$ performs 1-norm regularization. Combining both 1-norm and 2-norm regularization is sometimes called *Elastic Net* regularization.

Chapter 15

Geometry

15.1 Geometry of Linear Equations

Why do linear systems have convex solution spaces? Before answering, we should understand the shape of individual equations (rows) in the systems $\mathbf{Ax} = \mathbf{b}$. The equation corresponding to the i^{th} row is

$$\mathbf{A}(i,:) \cdot \mathbf{x} = b_i$$

which we will simplify by using the notation

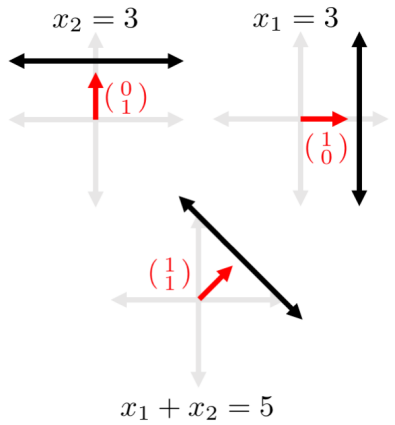
$$\mathbf{a} \cdot \mathbf{x} = b$$

where $\mathbf{a}$ and $\mathbf{x}$ are vectors and the value b is a scalar. In two dimensions, this expression defines a line

$$a_1 x_1 + a_2 x_2 = b$$

The above representation of a line is the *standard form*, which differs from the *slope-intercept* form you remember from algebra ($y = mx + b$). It seems intuitive why the slope-intercept form is a line; a change in x produces a corresponding change $m\Delta x$ in y, with an intercept b when $x = 0$. What is the analogous reasoning for why $\mathbf{a} \cdot \mathbf{x} = b$ is a line?

First, we note that the vector $\mathbf{a}$ always point perpendicular, or *normal* to the line. For the horizontal line $y = 3$, the vector $\mathbf{a} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ points vertically. For the vertical line $x = 3$, the vector $\mathbf{a} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ point horizontal. For the line $x_1 + x_2 = 1$, $\mathbf{a} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, which is still perpendicular to the original line.



To help visualize the equation $\mathbf{a} \cdot \mathbf{x} = b$, we need to know the length of $\mathbf{a}$. The easiest solution is to normalize $\mathbf{a}$ by dividing both sizes of the equation by the norm of $\mathbf{a}$.

$$\frac{1}{\|\mathbf{a}\|}\mathbf{a} \cdot \mathbf{x} = b/\|\mathbf{a}\|$$

If we use our previous notation of $\hat{\mathbf{a}}$ for the normalized form of $\mathbf{a}$ and define scalar $d = b/\|\mathbf{a}\|$, we have

$$\hat{\mathbf{a}} \cdot \mathbf{x} = d$$

We know that $\hat{\mathbf{a}}$ is a unit vector normal to the line. What is the meaning of d? Let's compute the dot product $\hat{\mathbf{a}} \cdot \mathbf{x}$ using an arbitrary point $\mathbf{x}$ on the line.

$$d = \hat{\mathbf{a}} \cdot \mathbf{x} = \|\hat{\mathbf{a}}\| \, \|\mathbf{x}\| \cos\theta = \|\mathbf{x}\| \cos\theta$$

Thus, d is the projection of the magnitude of $\mathbf{x}$ onto the normal vector. For any point on the line, this projection is always the same length – the distance between the origin and the nearest point on the line. Conversely, a line is the set of all vectors whose projection against a vector $\hat{\mathbf{a}}$ is a constant distance (d) from the origin.

The same interpretation follows in higher dimensions. In 3D, the expression $\hat{\mathbf{a}} \cdot \mathbf{x} = d$ defines a plane with normal vector $\hat{\mathbf{a}}$ at a distance d from the origin. This definition fits with the algebraic definition of a plane that you may have seen previously: $a_1x_1 + a_2x_2 + a_3x_3 = d$. In higher dimensions (four or more), this construct is called a *hyperplane*.

Remember that when analyzing an expression of the form $\hat{\mathbf{a}} \cdot \mathbf{x} = d$, the constant on the right hand side (d) is only equal to the distance between the line and the origin if the vector $\hat{\mathbf{a}}$ is normalized. For example, the line

$$3x_1 + 4x_2 = 7$$

has coefficient vector $\mathbf{a} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$, which is not normalized. To normalize $\mathbf{a}$, we divide both sides by $\|\mathbf{a}\| = \sqrt{3^2 + 4^2} = 5$, yielding

$$\frac{3}{5}x_1 + \frac{4}{5}x_2 = \frac{7}{5}$$

Now we can say that the distance between this line and the origin is 7/5.

The equation $\hat{\mathbf{a}} \cdot \mathbf{x} = d$ is called the Hesse normal form of a line, plane, or hyperplane.

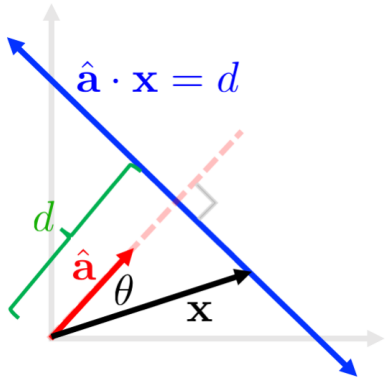


Figure 15.2: A line is the set of all points $\mathbf{x}$ whose projection onto $\hat{\mathbf{a}}$ is the distance d.

15.2 Geometry of Linear Systems

The equation $\hat{\mathbf{a}} \cdot \mathbf{x} = d$ defines a hyperplane. It is also a single row in the linear system $\mathbf{Ax} = \mathbf{b}$. What does the entire system of equations look like? First, let's

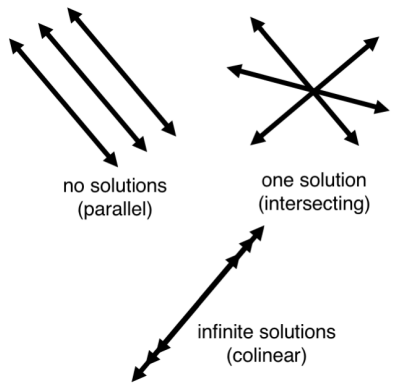


Figure 15.3: A system of linear equations can have zero, one, or infinitely many points of intersection.

consider a set of three equations in two dimensions (so we can visualize them as lines). Solutions to $\mathbf{Ax} = \mathbf{b}$ are points of intersection of all three equations. If the lines are parallel, no solutions exist. If the lines all intersect at one point, there is a unique solution. If the lines are *colinear* (all fall upon the same line), infinitely many solutions exist. Note that these are the only options – zero, one, or infinitely many solutions, just as predicted by the grand solvability theorem. It is impossible to draw three straight lines that intersect in only two places.

If linear systems $\mathbf{Ax} = \mathbf{b}$ are a set of intersecting lines in 2D, what is do the inequalities $\mathbf{Ax} \leq \mathbf{b}$ represent? Each inequality states that the projection of $\mathbf{x}$ onto the normal vector $\mathbf{a}$ must be less than d. These points form a *half-plane* – all the points on one side of a hyperplane. The system $\mathbf{Ax} \leq \mathbf{b}$ has a solution space that is the overlap of multiple half-planes (one for each row in $\mathbf{A}$). As we proved earlier, this solution set is a convex set.

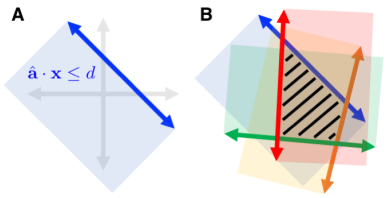


Figure 15.4: **A**. A single inequality defines a half-plane. **B**. Multiple half-planes intersect to form a convex solution set for the system $\mathbf{Ax} \leq \mathbf{b}$.

Chapter 16

Support Vector Machines

In this chapter we focus on *classification*, the problem of using *features* to predict which *class* an observation belongs to. We are particularly interested in distinguishing among two classes, a problem known as binary classification. We will introduce the Support Vector Machine, or SVM, a framework for solving classification problems using optimization and linear algebra.

"Machine" refers to an algorithm. We'll explain what a support vector is later in the chapter.

As an example, consider the following dataset that reports the blood pressure and cholesterol levels of 20 patients. Twelve of the patients have not experienced a heart attack, but the remaining eight have. Let's load and plot the data.

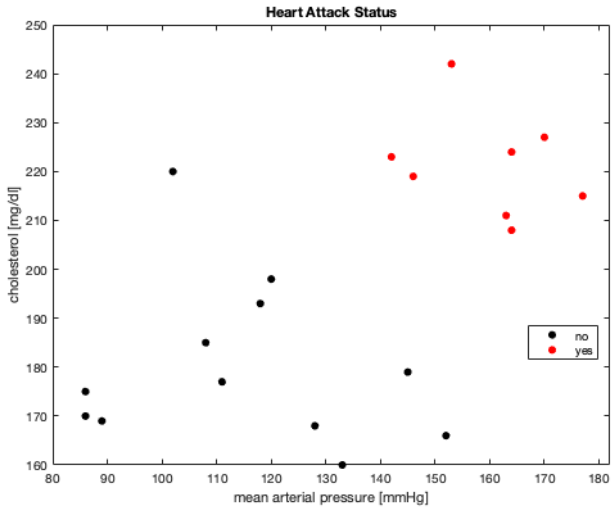
MATLAB code

```
load HeartAttack.mat
hatk
```

	BloodPressure	Cholesterol	HeartAttack
1	133	160	'no'
2	152	166	'no'
3	128	168	'no'
4	89	169	'no'
5	86	170	'no'
6	86	175	'no'
7	111	177	'no'
8	145	179	'no'
9	108	185	'no'
10	118	193	'no'

MATLAB code

```
gscatter(hatk.BloodPressure,hatk.Cholesterol, ...
         hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
title('Heart Attack Status')
hold off
```



It's clear that we can separate the patients who experienced a heart attack from the ones who did not. However, the separation requires knowledge of both blood pressure and cholesterol levels. There is no cholesterol level alone that separates the two classes, and the same is true for blood pressure. We want to identify a hyperplane that separates the classes so we can predict the heart attack risk for other patients.

For small datasets like this, it is possible to simply draw a line that separates the classes. For problems with only two features, classification is often trivial. However, classifying with thousands of features cannot be done *ad hoc*. Fortunately, everything we will learn in lower dimensions generalizes easily to higher dimensions.

16.1 Separating Hyperplanes

First we *code* the points by setting one group equal to +1 and the other group to −1. For our heart attack data, patients who experienced a heart attack are +1 and the rest are −1.

The term *code* in this context is unrelated to computer programming.

MATLAB code

```
hatk.HeartAttack = [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
                     1  1  1  1  1  1  1  1]'
```

	BloodPressure	Cholesterol	HeartAttack
1	133	160	-1
2	152	166	-1
3	128	168	-1
4	89	169	-1
5	86	170	-1
6	86	175	-1
7	111	177	-1
8	145	179	-1
9	108	185	-1
10	118	193	-1

The +1 and −1 designations are arbitrary — it doesn't matter which group is which. Switching the +1 and −1 codings will give the same classifer. The resulting hyperplane will have the normal vector pointing the opposite way, but this does not affect performance of the classifier.

Our goal is to find a hyperplane that separates the +1 and −1 points. Recall that any hyperplane can be represented in the Hesse form as

This is the Hesse form, not the Hesse normal form since **a** has not been normalized.

$$\mathbf{a} \cdot \mathbf{x} = b$$

where **a** is a vector of coefficients and b is a scalar. Normally our goal is to find values for the vector **x**. For classification problems we know that values of **x** (the features) for each point. Our goal is to find the coefficients **a** and the scalar b that define the separating hyperplane.

We want to choose **a** and b such that all of the +1 points are on one side of the hyperplane and all of the −1 points lie on the other size. By convention, we will put the +1 points above the plane, which we enforce with the constraint

$$\mathbf{a} \cdot \mathbf{x} \geq b$$

for any values $\mathbf{x}$ that are coded $+1$. Similarly, we require the -1 points be below the hyperplane with the constraint

$$\mathbf{a} \cdot \mathbf{x} \leq b$$

for any values $\mathbf{x}$ that are coded -1. Note that there are usually infinitely many hyperplanes that can separate the $+1$ and -1 points. We want to find the hyperplane that maximizes the gap, or *margin*, between the $+1$ and -1 points. The hyperplane that maximizes this gap is called the *maximal margin hyperplane*.

To find the maximal margin hyperplane, we start with two parallel hyperplanes. We require all the $+1$ points be above the top plane and all -1 points be below the bottom plane. We push the two planes apart until the top plane hits the nearest $+1$ point and the bottom plane hits the nearest -1 point. When the gap between the two planes is maximized, we know that the maximal margin hyperplane will sit halfway in between the two planes.

Let's formalize this procedure. We define the top plane to be $\mathbf{a} \cdot \mathbf{x} = b + 1$ and the bottom plane to be $\mathbf{a} \cdot \mathbf{x} = b - 1$. Since both planes have the same normal vector $\mathbf{a}$ we know they are parallel. The ± 1 terms are arbitrary since we aren't requiring the vector $\mathbf{a}$ to be a unit vector. How far apart are these planes? Let's convert the planes to the Hess normal form. Then the top plane is at a distance of $(b + 1)/\|\mathbf{a}\|$ from the origin and the bottom plane is at distance $(b - 1)/\|\mathbf{a}\|$. The distance between the planes is therefore

$$\frac{b+1}{\|\mathbf{a}\|} - \frac{b-1}{\|\mathbf{a}\|} = \frac{2}{\|\mathbf{a}\|}$$

The gap between the planes is inversely proportional to the magnitude of $\mathbf{a}$. Maximizing the separation between the planes is equivalent to minimizing the magnitude of $\mathbf{a}$. All together, the maximal margin hyperplane is the solution to the following constrained optimization problem:

$$\begin{aligned} \underset{\mathbf{a},b}{\text{minimize}} \quad & \|\mathbf{a}\| \\ \text{subject to} \quad & \mathbf{a} \cdot \mathbf{x} \geq b + 1 \quad \text{for the } +1 \text{ points} \\ & \mathbf{a} \cdot \mathbf{x} \leq b - 1 \quad \text{for the } -1 \text{ points} \end{aligned}$$

This might seem like a difficult optimization, but there is an important simplification. Remember the definition of the magnitude

$$\|\mathbf{a}\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$$

The square root function is *monotonically increasing*, meaning it always increases as its argument increases. Because of monotonicity, minimizing the square root of an

Functions like $\cos(x)$ are not monotonic as they both increase and decrease as x increases.

input is equivalent to minimizing the input itself. Rather than minimize $\|\mathbf{a}\|$ we can simply minimize the expression $a_1^2 + a_2^2 + \cdots + a_n^2$. The classification problem becomes

$$\begin{aligned} \underset{\mathbf{a},b}{\text{minimize}} \quad & a_1^2 + a_2^2 + \cdots + a_n^2 \\ \text{subject to} \quad & \mathbf{a} \cdot \mathbf{x} \geq b + 1 \qquad \text{for the } +1 \text{ points} \\ & \mathbf{a} \cdot \mathbf{x} \leq b - 1 \qquad \text{for the } -1 \text{ points} \end{aligned}$$

Since the objective is purely quadratic with positive coefficients, we know it is convex. We also know that the constraints are linear and therefore also convex. We are minimizing a convex function over a convex set, which is easily solved.

16.2 Setting up the SVM Quadratic Program

The SVM problem outlined above is a *quadratic program (QP)*, a term in optimization that means a problem with a quadratic objective function and a set of linear constraints. Let's set up a QP for four observations from our heart attack data:

Blood Pressure	Cholesterol	HeartAttack
133	160	−1
89	169	−1
164	224	+1
153	242	+1

1. Define the dimensions of the problem. We have two predictor variables: blood pressure and cholesterol level. Let's set x_1 = blood pressure and x_2 = cholesterol. We then know that $\mathbf{a}$ has two dimensions (a_1 and a_2).

2. Write out the objective function. The objective is to minimize the magnitude of $\mathbf{a}$, or
$$\underset{a_1,a_2,b}{\text{minimize}} \; a_1^2 + a_2^2$$

3. Write out constraints for each point by substituting values for $\mathbf{x}$. We have four points so we will have four constraints. The constraints for the −1 points are
$$\begin{aligned} 133a_1 + 160a_2 &\leq b - 1 \\ 89a_1 + 169a_2 &\leq b - 1 \end{aligned}$$

For the +1 points we have

$$\begin{aligned} 164a_1 + 224a_2 &\geq b + 1 \\ 153a_1 + 242a_2 &\geq b + 1 \end{aligned}$$

All together, the quadratic program for finding the SVM classifier for these data is

$$\begin{aligned} \underset{a_1,a_2,b}{\text{minimize}} \quad & a_1^2 + a_2^2 \\ \text{subject to} \quad & 133a_1 + 160a_2 \leq b - 1 \\ & 89a_1 + 169a_2 \leq b - 1 \\ & 164a_1 + 224a_2 \geq b + 1 \\ & 153a_1 + 242a_2 \geq b + 1 \end{aligned}$$

16.3 SVM in MATLAB

Setting up an SVM problem by hand is informative but unwieldy for large datasets. There are several software libraries for efficiently formulating and solving SVM problems. We will use the `fitcsvm` function to fit an SVM classifier. The function takes two arguments: a matrix of features and a vector with class codings. Let's begin by putting our two features into a matrix.

The name `fitcsvm` stands for "fit classifier SVM".

MATLAB code

```
features = [hatk.BloodPressure hatk.Cholesterol]
```

MATLAB output

```
features = 20x2
   133   160
   152   166
   128   168
    89   169
    86   170
    86   175
   111   177
   145   179
   108   185
   118   193
```

Now we call `fitcsvm` and store the output in a variable that we'll call `model`.

MATLAB code

```
model = fitcsvm(features, hatk.HeartAttack)
```

MATLAB output

```
model =
  ClassificationSVM
              ResponseName: 'Y'
    CategoricalPredictors: []
               ClassNames: [-1 1]
           ScoreTransform: 'none'
          NumObservations: 20
                    Alpha: [3x1 double]
                     Bias: -16.4864
         KernelParameters: [1x1 struct]
           BoxConstraints: [20x1 double]
          ConvergenceInfo: [1x1 struct]
          IsSupportVector: [20x1 logical]
                   Solver: 'SMO'
```

The model object contains lots of information. Some important pieces are the values for $\mathbf{a}$ (`model.Beta`) and value of the scalar b (`model.Bias`)

MATLAB code

```
model.Beta
```

MATLAB output

```
ans = 2x1
    0.0465
    0.0488
```

MATLAB code

```
model.Bias
```

MATLAB output

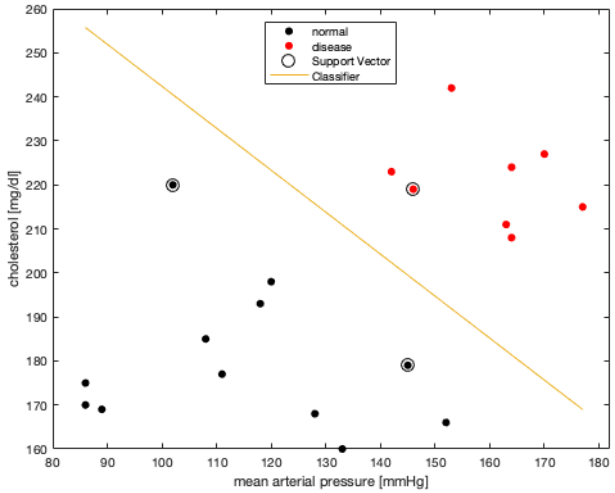
```
ans = -16.4864
```

We can use these values to plot the maximal margin hyperplane.

MATLAB code

```
bp = hatk.BloodPressure; ch = hatk.Cholesterol;
gscatter(bp,ch,hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
plot(bp(model.IsSupportVector), ...
     ch(model.IsSupportVector), ...
     'ko', 'MarkerSize',10);
```

```
plot(bp, ...
     -model.Beta(1)/model.Beta(2)*bp ...
         - (model.Bias)/model.Beta(2))
legend('normal','disease','Support Vector','Classifier');
hold off
```



We've also identified the *support vectors* on the above plot. Remember that to find the maximal margin hyperplane we push two parallel planes outward until they hit the nearest +1 and −1 points. These nearest points are called the support vectors since they "support" the planes. Support vectors are what determine the location of the maximal margin hyperplane; their importance gives rise to the name Support Vector Machine.

So far we've trained an SVM model. We can also make predictions about new patients using the model object and the Matlab function `predict`. The `predict` function accepts a model object and a matrix of features for the unknown observations. It returns predictions (+1 or −1) for each observation. Let's make predictions for two new patients with the following data:

Blood Pressure	Cholesterol
153	230
99	132

MATLAB code

```
predict(model, [153 230; 99 132])
```

MATLAB output

```
ans = 2x1
     1
    -1
```

Our model predicts the first patient would have a history of heart attack while the second patient would not.

16.4 *k*-fold Cross Validation in Matlab

Performing a *k*-fold cross validation requires 1.) randomizing the folds, 2.) retraining the model, and 3.) classifying each fold. Fortunately, there is a Matlab function to perform *k*-fold cross validation. We can perform a 5-fold cross validation on our heart attack SVM model as follows

MATLAB code

```
xval = crossval(model,'Kfold',5)
```

MATLAB output

```
xval =
  classreg.learning.partition.ClassificationPartitionedModel
    CrossValidatedModel: 'SVM'
         PredictorNames: {'x1'  'x2'}
           ResponseName: 'Y'
        NumObservations: 20
                  KFold: 5
              Partition: [1x1 cvpartition]
             ClassNames: [-1 1]
         ScoreTransform: 'none'
```

The object returned by `crossval` contains information about how the folds were created and tested. The accuracy of the classifier is measured by the *loss*, with lower loss meaning a better model. We can find the loss by calling the `kfoldLoss` function on our `crossval` return object.

Note the capital "L" in `kfoldLoss`.

MATLAB code

```
kfoldLoss(xval)
```

MATLAB output

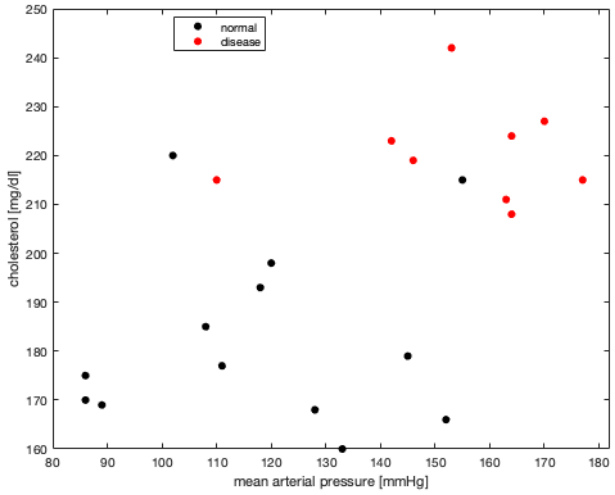
```
ans = 0
```

16.5 Soft Classifiers

Our heart attack data was perfectly classifiable since we could cleanly separate the $+1$ and -1 classes. This is not always the case, especially for biological datasets. Let's add two points to our dataset and replot the data.

MATLAB code

```
hatk(end+1,:) = {155,215,-1};
hatk(end+1,:) = {110,215, 1};
gscatter(hatk.BloodPressure,hatk.Cholesterol, ...
         hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
legend('normal','disease')
hold off
```

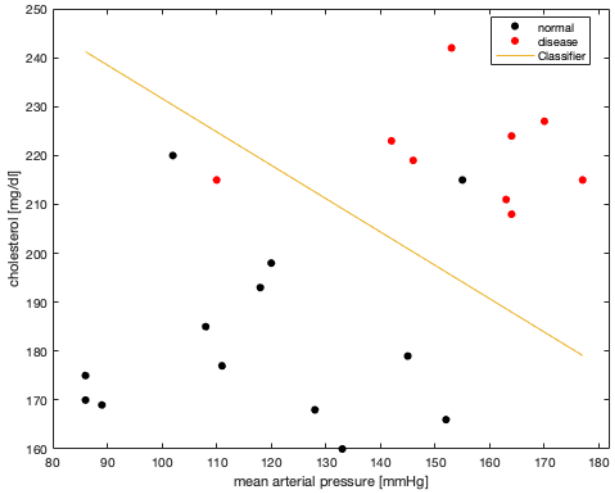


With the new data, it doesn't appear that we can perfectly separate the heart attacks from the rest. Let's try to refit our model.

MATLAB code

```
mdl = fitcsvm([hatk.BloodPressure hatk.Cholesterol], ...
              hatk.HeartAttack);
gscatter(hatk.BloodPressure,hatk.Cholesterol, ...
```

```
        hatk.HeartAttack,'kr');
hold on
xlabel('mean arterial pressure [mmHg]');
ylabel('cholesterol [mg/dl]');
plot(hatk.BloodPressure, ...
     -mdl.Beta(1)/mdl.Beta(2)*hatk.BloodPressure ...
         - (mdl.Bias)/mdl.Beta(2))
legend('normal','disease','Classifier');
hold off
```



Now we have some points that sit on the wrong side of the classifier. Our accuracy should have decreased (i.e. out loss during cross validation should increase).

MATLAB code

```
xval = crossval(mdl,'Kfold',5);
kfoldLoss(xval)
```

MATLAB output

```
ans = 0.0909
```

The SVM formulation we described above is called a *hard classifier* since it requires that all points be on the correct side of the classifier. In practice, SVM software packages use a *soft classifier* where points can appear on the wrong side.

When solving SVM problems with soft classifiers, the goal is to both maximize the separation and minimize the number of incorrectly classified points. We will not discuss the mathematical formulation of soft classifiers in this class.