

Chapter 4

Regression and Model Selection

All of machine learning revolves around optimization. This includes regression and model selection frameworks which aim to provide parsimonious and interpretable models for data[244]. Curve fitting is the most basic of regression techniques, with polynomial and exponential fitting resulting in solutions that come from solving the linear system

$$\mathbf{Ax} = \mathbf{b} . \quad (4.1)$$

When the model is not prescribed, then optimization methods are used to select the best model. This then changes the underlying mathematics for function fitting to either an overdetermined or underdetermined optimization problem for linear systems given by:

$$\underset{\mathbf{x}}{\operatorname{argmin}} (\|\mathbf{Ax} - \mathbf{b}\|_2 + \lambda g(\mathbf{x})) \quad (4.2a)$$

$$\underset{\mathbf{x}}{\operatorname{argmin}} g(\mathbf{x}) \text{ subject to } \|\mathbf{Ax} - \mathbf{b}\|_2 , \quad (4.2b)$$

where $g(\mathbf{x})$ is a given penalization (with penalty parameter λ for overdetermined systems). For over and underdetermined linear systems of equations, which result in either no solutions or an infinite number of solutions of (4.1), a choice of constraint or penalty must be made in order to produce a solution. For instance, one can enforce a solution minimizing the smallest ℓ_2 norm in an underdetermined system so that $\min g(\mathbf{x}) = \min \|\mathbf{x}\|_2$. More generally, when considering regression to nonlinear models, then the overall mathematical framework takes the general form

$$\underset{\mathbf{x}}{\operatorname{argmin}} (f(\mathbf{A}, \mathbf{x}, \mathbf{b}) + \lambda g(\mathbf{x})) \quad (4.3a)$$

$$\underset{\mathbf{x}}{\operatorname{argmin}} g(\mathbf{x}) \text{ subject to } f(\mathbf{A}, \mathbf{x}, \mathbf{b}) \quad (4.3b)$$

which is often solved using gradient descent algorithms. Indeed, this general framework is also at the center of deep learning algorithms. The next few chapters will outline how each of these methods arise in practice, and will highlight

the choices that need to be made in applying meaningful constraints $g(\mathbf{x})$ so as to achieve interpretable solutions.

4.1 Classic curve fitting

Curve fitting is one of the most basic and foundational tools in data science. From our earliest educational experiences in the engineering and physical sciences, least-square polynomial fitting was advocated for understanding the dominant trends in real data. Andrien-Marie Legendre used least-squares as early as 1805 to fit astronomical data [300], with Gauss more fully developing the theory of least squares as an optimization problem in a seminal contribution of 1821 [182]. Curve fitting in such astronomical applications was highly effective given the simple elliptical orbits (quadratic polynomial functions) manifest by planets and comets. Thus one can argue that data science has long been a cornerstone of our scientific efforts. Indeed, it was through Kepler's access to Tycho Brahe's state-of-the art astronomical data that he was able, after eleven years of research, to produce the foundations for the laws of planetary motion, positing the elliptical nature of planetary orbits which were clearly best-fit solutions to the available data [262].

A broader mathematical viewpoint of curve fitting, which we will advocate throughout this text, is *regression*. Like curve fitting, regression attempts to estimate the relationship among variables using a variety of statistical tools. Specifically, one can consider the general relationship between independent variables \mathbf{X} , the dependent variables \mathbf{Y} and some unknown parameters β :

$$\mathbf{Y} = f(\mathbf{X}, \beta) \quad (4.4)$$

where the regression function $f(\cdot)$ is typically prescribed and the parameters β are found by optimizing the *goodness-of-fit* of this function. In what follows, we will consider curve fitting as a special case of regression. Importantly, regression and curve fitting discover relationships among variables by optimization. Broadly speaking, machine learning is framed around regression techniques, which are themselves framed around optimization. Thus at its absolute mathematical core, machine learning and data science revolve around positing an optimization problem. Of course, the success of optimization itself depends critically on defining an *objective function* to be optimized.

Least-Square Fitting Methods

To illustrate the concepts of regression, we will consider classic least-square polynomial fitting for characterizing trends in data. The concept is straightforward and simple: use a simple function to describe a trend by minimizing

the least-square error between the selected function $f(\cdot)$ and its fit to the data. As we show here, classical curve fitting is formulated as a simple solution of $\mathbf{Ax} = \mathbf{b}$.

Consider a set of n data points

$$(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_n, y_n). \quad (4.5)$$

Further, assume that we would like to fit a best fit line through these points. We can approximate the line by the function

$$f(x) = Ax + B \quad (4.6)$$

where the constants A and B , which are the parameters of the vector β of (4.4), are chosen to minimize some error associated with the fit. The line fit gives the *linear regression* model $\mathbf{Y} = f(\mathbf{A}, \beta) = \mathbf{AX} + B$. Thus the function gives a linear model which approximates the data, with the approximation error at each point being given by

$$f(x_k) = y_k + E_k \quad (4.7)$$

where y_k is the true value of the data and E_k is the error from this value.

Various error metrics can be minimized when approximating with a given function $f(x)$. The choice of error metric, or norm, used to compute a goodness-of-fit will be critical in this chapter. Three standard possibilities are often considered which are associated with the ℓ_2 (least-squares), ℓ_1 and ℓ_∞ norms. These are defined as follows

$$E_\infty(f) = \max_{1 \leq k \leq n} |f(x_k) - y_k| \quad \text{Maximum Error } (\ell_\infty) \quad (4.8a)$$

$$E_1(f) = \frac{1}{n} \sum_{k=1}^n |f(x_k) - y_k| \quad \text{Average Error } (\ell_1) \quad (4.8b)$$

$$E_2(f) = \left(\frac{1}{n} \sum_{k=1}^n |f(x_k) - y_k|^2 \right)^{1/2} \quad \text{Least-squares Error } (\ell_2). \quad (4.8c)$$

Such regression error metrics have been previously considered in Chapter 1, but they will be considered once again here in the framework of model selection. And in addition to the above norms, one can more broadly consider the ℓ_p -norm error metric

$$E_p(f) = \left(\frac{1}{n} \sum_{k=1}^n |f(x_k) - y_k|^p \right)^{1/p}. \quad (4.9)$$

For different p values, the best fit line will be different. In most cases, the differences are small. However, when there are outliers in the data set, the choice of norm can have a significant impact.

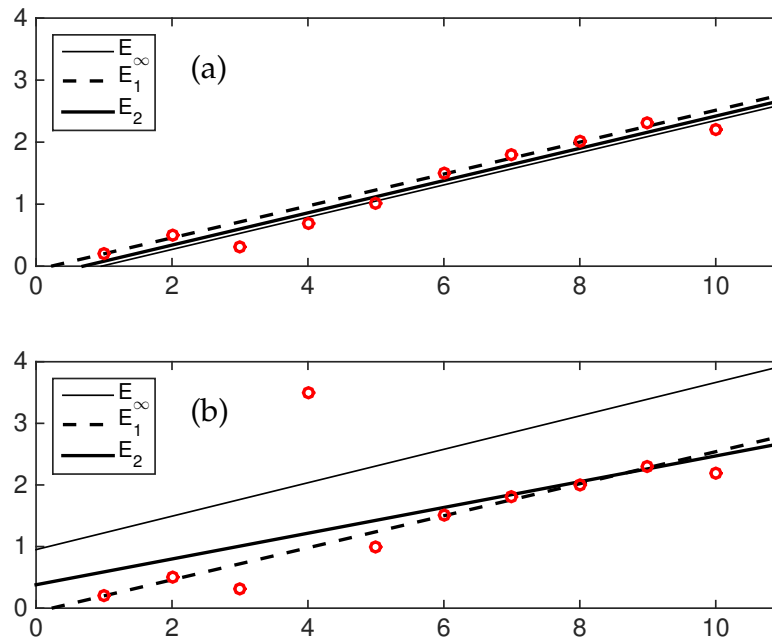


Figure 4.1: Line fits for the three different error metrics E_∞ , E_1 and E_2 . In (a), the data has not outliers and the three linear models, although different, produce approximately the same model. With outliers, (b) shows that the predictions are significantly different.

When fitting a curve to a set of data, the root-mean square error (4.8c) is often chosen to be minimized. This is called a *least-squares fit*. Figure 4.1 depicts three line fits for the errors E_∞ , E_1 and E_2 listed above. The E_∞ error line fit is strongly influenced by the one data point which does not fit the trend. The E_1 and E_2 line fit nicely through the bulk of the data, although their slopes are quite different in comparison to when the data has no outliers. The linear models for these three error metrics are constructed using MATLAB's **fminsearch** command. The code for all three is given as follows:

Code 4.1: Regression for linear fit.

```
% The data
x=[1 2 3 4 5 6 7 8 9 10]
y=[0.2 0.5 0.3 3.5 1.0 1.5 1.8 2.0 2.3 2.2]

p1=fminsearch('fit1',[1 1],[],x,y);
p2=fminsearch('fit2',[1 1],[],x,y);
p3=fminsearch('fit3',[1 1],[],x,y);

xf=0:0.1:11
yl=polyval(p1,xf);
```

```

y2=polyval(p2,xf);
y3=polyval(p3,xf);

subplot(2,1,2)
plot(xf,y1,'k'), hold on
plot(xf,y2,'k--','Linewidth',[2])
plot(xf,y3,'k','Linewidth',[2])
plot(x,y,'ro','Linewidth',[2]), hold on

```

For each error metric, the computation of the error metrics (4.8) must be computed. The **fminsearch** command requires that the objective function for minimization be given. For the three error metrics considered, this results in the following set of functions for **fminsearch**:

Code 4.2: Maximum error ℓ_∞ .

```

function E=fit1(x0,x,y)
E=max(abs(x0(1)*x+x0(2)-y));

```

Code 4.3: Sum of absolute error ℓ_1 .

```

function E=fit2(x0,x,y)
E=sum(abs(x0(1)*x+x0(2)-y));

```

Code 4.4: Least-square error ℓ_2 .

```

function E=fit3(x0,x,y)
E=sum(abs(x0(1)*x+x0(2)-y).^2);

```

Finally, for the outlier data, an additional point is added to the data set in order to help illustrate the influence of the error metrics (norm) on producing a linear regression model.

Code 4.5: Data which includes an outlier.

```

y=[0.2 0.5 0.3 0.7 1.0 1.5 1.8 2.0 2.3 2.2]

```

Least-Squares Line

Least-square fitting to linear models has critical advantages over other norms and metrics. Specifically, a potentially expensive optimization is not required since the error can be computed analytically. To show this explicitly, consider applying the least-square fit criteria to the data points $\{x_k, y_k\}$, where $k = 1, 2, 3, \dots, n$. To fit the curve

$$f(x) = Ax + B \quad (4.10)$$

to this data, the E_2 is found by minimizing the sum

$$E_2(f) = \sum_{k=1}^n |f(x_k) - y_k|^2 = \sum_{k=1}^n (Ax_k + B - y_k)^2. \quad (4.11)$$

Minimizing this sum requires differentiation. Specifically, the constants A and B are chosen so that a minimum occurs. thus we require: $\partial E_2/\partial A = 0$ and $\partial E_2/\partial B = 0$. Note that although a zero derivative can indicate either a minimum or maximum, we know this must be a minimum to the error since there is no maximum error, i.e. we can always choose a line that has a larger error. The minimization condition gives:

$$\frac{\partial E_2}{\partial A} = 0 : \sum_{k=1}^n 2(Ax_k + B - y_k)x_k = 0 \quad (4.12a)$$

$$\frac{\partial E_2}{\partial B} = 0 : \sum_{k=1}^n 2(Ax_k + B - y_k) = 0. \quad (4.12b)$$

Upon rearranging, the 2×2 system of linear equations is found for A and B

$$\begin{pmatrix} \sum_{k=1}^n x_k^2 & \sum_{k=1}^n x_k \\ \sum_{k=1}^n x_k & n \end{pmatrix} \begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^n x_k y_k \\ \sum_{k=1}^n y_k \end{pmatrix} \longrightarrow \mathbf{Ax} = \mathbf{b}. \quad (4.13)$$

This linear system of equations can be easily solved using the backslash command in MATLAB. Thus an optimization procedure is unnecessary since the solution is computed exactly from a 2×2 matrix.

This method can be easily generalized to higher polynomial fits. In particular, a parabolic fit to a set of data requires the fitting function

$$f(x) = Ax^2 + Bx + C \quad (4.14)$$

where now the three constants A , B , and C must be found. These can be found from the 3×3 system which results from minimizing the error $E_2(A, B, C)$ by taking

$$\frac{\partial E_2}{\partial A} = 0 \quad (4.15a)$$

$$\frac{\partial E_2}{\partial B} = 0 \quad (4.15b)$$

$$\frac{\partial E_2}{\partial C} = 0. \quad (4.15c)$$

In fact, any polynomial fit of degree k will yield a $(k+1) \times (k+1)$ linear system of equations $\mathbf{Ax} = \mathbf{b}$ whose solution can be easily found.

Data Linearization

Although a powerful method, the minimization procedure for general fitting of arbitrary functions results in equations which are nontrivial to solve. Specifically, consider fitting data to the exponential function

$$f(x) = C \exp(Ax). \quad (4.16)$$

The error to be minimized is

$$E_2(A, C) = \sum_{k=1}^n (C \exp(Ax_k) - y_k)^2. \quad (4.17)$$

Applying the minimizing conditions leads to

$$\frac{\partial E_2}{\partial A} = 0 : \sum_{k=1}^n 2(C \exp(Ax_k) - y_k) C x_k \exp(Ax_k) = 0 \quad (4.18a)$$

$$\frac{\partial E_2}{\partial C} = 0 : \sum_{k=1}^n 2(C \exp(Ax_k) - y_k) \exp(Ax_k) = 0. \quad (4.18b)$$

This in turn leads to the 2×2 system

$$C \sum_{k=1}^n x_k \exp(2Ax_k) - \sum_{k=1}^n x_k y_k \exp(Ax_k) = 0 \quad (4.19a)$$

$$C \sum_{k=1}^n \exp(Ax_k) - \sum_{k=1}^n y_k \exp(Ax_k) = 0. \quad (4.19b)$$

This system of equations is nonlinear and cannot be solved in a straightforward fashion. Indeed, a solution may not even exist. Or many solution may exist. Section 4.2 describes a possible iterative procedure, called gradient descent, for solving this nonlinear system of equations.

To avoid the difficulty of solving this nonlinear system, the exponential fit can be *linearized* by the transformation

$$Y = \ln(y) \quad (4.20a)$$

$$X = x \quad (4.20b)$$

$$B = \ln C. \quad (4.20c)$$

Then the fit function

$$f(x) = y = C \exp(Ax) \quad (4.21)$$

can be linearized by taking the natural log of both sides so that

$$\ln y = \ln(C \exp(Ax)) = \ln C + \ln(\exp(Ax)) = B + Ax \rightarrow Y = AX + B. \quad (4.22)$$

So by fitting to the natural log of the y -data

$$(x_i, y_i) \rightarrow (x_i, \ln y_i) = (X_i, Y_i) \quad (4.23)$$

the curve fit for the exponential function becomes a linear fitting problem which is easily handled. Thus if a transform exists which linearizes the data, then standard polynomial fitting methods, which are trivially applied, can be used to solve the resulting $\mathbf{Ax} = \mathbf{b}$ problem.

4.2 Nonlinear regression and gradient descent

Polynomial and exponential curve fitting allows for an analytically tractable, least-square fit solutions. However, such curve fits are highly specialized and a more general mathematical framework is necessary for solving a broader set of problems. For instance, one may wish to fit the nonlinear function of the form $f(x) = A \cos(Bx + C) + D$ to a set of data. Instead of solving a linear system of equations, general nonlinear curve fitting leads to a system of nonlinear systems of equations. The general theory of nonlinear regression assumes that the fitting take the general form

$$f(x) = f(x, C_1, C_2, C_3, \dots, C_M) \quad (4.24)$$

where the C_i are fitting coefficients used to minimize the error and $M < n$. The root-mean square error is then defined as

$$E_2(C_1, C_2, C_3, \dots, C_m) = \sum_{k=1}^n (f(x_k, C_1, C_2, C_3, \dots, C_M) - y_k)^2 \quad (4.25)$$

which can be minimized by considering the $M \times M$ system generated from minimizing with respect to each parameters C_j

$$\frac{\partial E_2}{\partial C_j} = 0 \quad j = 1, 2, \dots, M. \quad (4.26)$$

In general, this gives the *nonlinear* set of equations

$$\sum_{k=1}^n (f(x_k, C_1, C_2, C_3, \dots, C_M) - y_k) \frac{\partial f}{\partial C_j} = 0 \quad j = 1, 2, 3, \dots, M. \quad (4.27)$$

There are no general methods available for solving such nonlinear systems. Indeed, nonlinear systems can have no solutions, several solutions, or even an infinite number of solutions. Most attempts at solving nonlinear systems are based upon iterative schemes which require good initial guesses to converge to the global minimum. Regardless, the general fitting procedure is straightforward and allows for the construction of a best fit curve to match the data. In such a solution procedure, it is imperative that a reasonable initial guess be provided for by the user. Otherwise, the desired rapid convergence or convergence to the desired root may not be achieved.

Figure 4.2 shows two example functions to be minimized. The first is a convex function (Fig. 4.2(a)). Convex functions are ideal in that guarantees of convergence may be constructed for many algorithms, and gradient descent can be tuned to perform exceptionally well for such functions. The second illustrates a

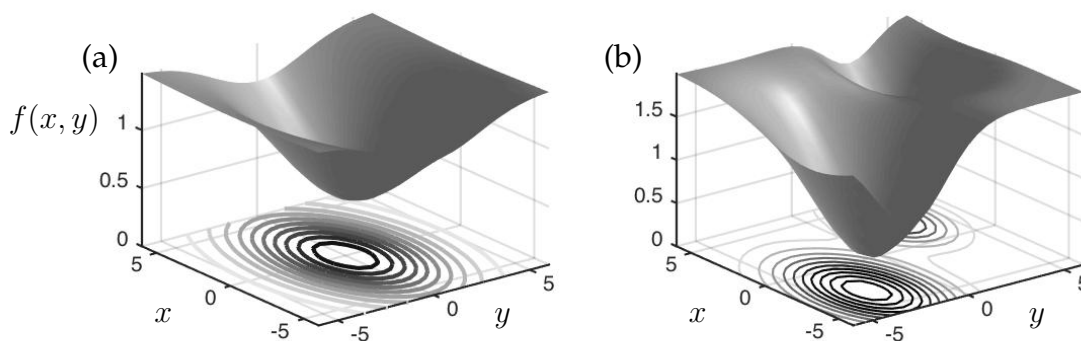


Figure 4.2: Two objective function landscapes representing (a) a convex function and (b) a non-convex function. Convex functions have many guarantees of convergence, while non-convex functions have a variety of pitfalls that can limit the success of gradient descent. For non-convex functions, local minima and an inability to compute gradient directions (derivatives that are near zero) make it challenging for optimization.

non-convex function and shows many of the typical problems associated with gradient descent, including the fact that the function has local minima as well as flat regions where gradients are difficult to actually compute, i.e. the gradient is near zero. Optimizing this non-convex function requires a good guess for the initial conditions of the gradient descent algorithm, although there many advancements around gradient descent for restarting and ensuring that one is not stuck in a local minima. Recent training algorithms for deep neural networks has greatly advanced gradient descent innovations. This will be further considered in our neural network Chapter 6.

Gradient descent

For high-dimensional systems, we generalize the concept of a minimum or maximum, i.e. an extremum for a multi-dimensional function $f(\mathbf{x})$. At an extremum, the gradient must be zero so that

$$\nabla f(\mathbf{x}) = 0. \quad (4.28)$$

Unlike the one-dimensional case, there is no simple second derivative test to apply to determine if the extremum point is a minimum or maximum, i.e. saddles in higher-dimensional spaces exist. The idea behind gradient descent, or steepest descent, is to use the derivative information as the basis of an iterative algorithm that progressively moves closer and closer to the minimum point $f(\mathbf{x}) = 0$.

To illustrate how to proceed in practice, consider the simple example two-dimensional surface

$$f(x, y) = x^2 + 3y^2 \quad (4.29)$$

which has the minimum located at the origin $(x, y) = 0$. The gradient for this function can be easily computed

$$\nabla f(\mathbf{x}) = \frac{\partial f}{\partial x} \hat{\mathbf{x}} + \frac{\partial f}{\partial y} \hat{\mathbf{y}} = 2x\hat{\mathbf{x}} + 6y\hat{\mathbf{y}} \quad (4.30)$$

where $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ are unit vectors in the x and y directions respectively.

Figure 4.3 illustrates the steepest descent (gradient descent) algorithm. At the initial guess point, the gradient $\nabla f(\mathbf{x})$ can be computed. This gives the steepest descent towards the minimum point of $f(\mathbf{x})$, i.e. the minimum is located in the direction given by $-\nabla f(\mathbf{x})$. Note that the gradient does not point at the minimum, but rather gives the steepest path for minimizing $f(\mathbf{x})$. The geometry of the steepest descent suggests the construction of an algorithm whereby the next point of iteration is picked by following the steepest descent so that

$$\xi(\tau) = \mathbf{x} - \tau \nabla f(\mathbf{x}) \quad (4.31)$$

where the parameter τ dictates how far to move along the gradient descent curve. This formula represents a generalization of a Newton method where the derivative is used to compute an update in the iteration scheme. In gradient descent, it is crucial to determine how much to step forward according to the computed gradient so that the algorithm is always *going downhill* in an optimal way. This requires the determination of the correct value of τ in the algorithm.

To compute the value of τ , consider the construction of a new function

$$F(\tau) = f(\xi(\tau)) \quad (4.32)$$

which must be minimized now as a function of τ . This is accomplished by computing $dF/d\tau = 0$. Thus one finds

$$\frac{\partial F}{\partial \tau} = -\nabla f(\xi) \nabla f(\mathbf{x}) = 0. \quad (4.33)$$

The geometrical interpretation of this result is the following: $\nabla f(\mathbf{x})$ is the gradient direction of the current iteration point and $\nabla f(\xi)$ is the gradient direction of the future point, thus τ is chosen so that the two gradient directions are orthogonal.

For the example given above with $f(x, y) = x^2 + 3y^2$, we can easily compute this conditions as follows:

$$\xi = \mathbf{x} - \tau \nabla f(\mathbf{x}) = (1 - 2\tau)x \hat{\mathbf{x}} + (1 - 6\tau)y \hat{\mathbf{y}}. \quad (4.34)$$

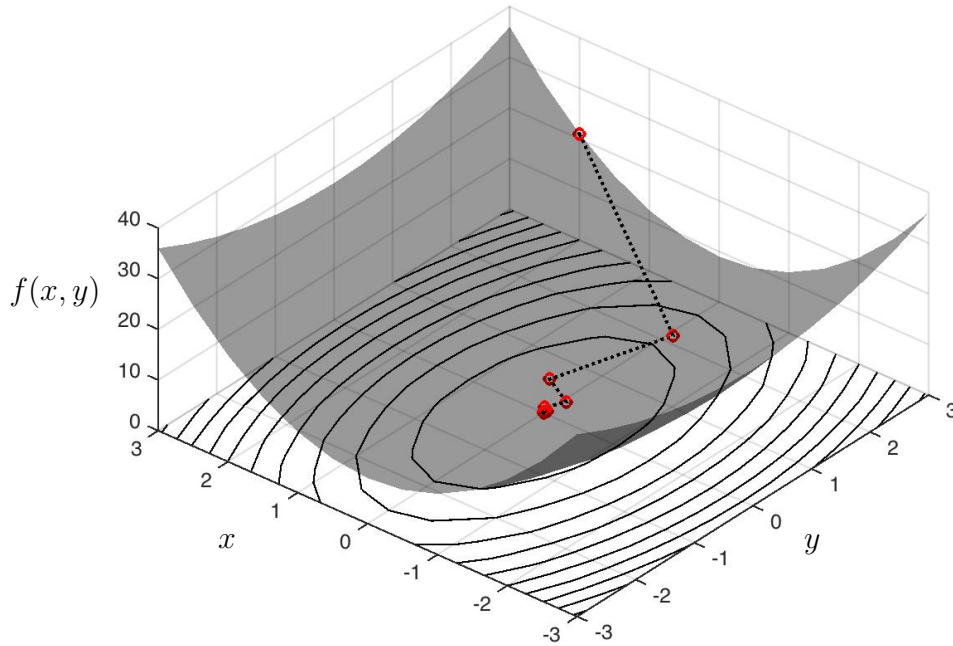


Figure 4.3: Gradient descent algorithm applied to the function $f(x, y) = x^2 + 3y^2$. In the top panel, the contours are plotted for each successive value (x, y) in the iteration algorithm given the initial guess $(x, y) = (3, 2)$. Note the orthogonality of each successive gradient in the steepest descent algorithm. The bottom panel demonstrates the rapid convergence and error (E) to the minimum (optimal) solution.

This is then used to compute

$$F(\tau) = f(\xi(\tau)) = (1 - 2\tau)^2 x^2 + 3(1 - 6\tau)^2 y^2 \quad (4.35)$$

whereby its derivative with respect to τ gives

$$F'(\tau) = -4(1 - 2\tau)x^2 - 36(1 - 6\tau)y^2. \quad (4.36)$$

Setting $F'(\tau) = 0$ then gives

$$\tau = \frac{x^2 + 9y^2}{2x^2 + 54y^2} \quad (4.37)$$

as the optimal descent step length. Note that the length of τ is updated as the algorithm progresses. This gives us all the information necessary to perform the steepest descent search for the minimum of the given function.

Code 4.6: Gradient descent example.

```

x(1)=3; y(1)=2; % initial guess
f(1)=x(1)^2+3*y(1)^2; % initial function value
for j=1:10
    tau=(x(j)^2 +9*y(j)^2)/(2*x(j)^2 + 54*y(j)^2);
    x(j+1)=(1-2*tau)*x(j); % update values
    y(j+1)=(1-6*tau)*y(j);
    f(j+1)=x(j+1)^2+3*y(j+1)^2;

    if abs(f(j+1)-f(j))<10^(-6) % check convergence
        break
    end
end
end

```

As is clearly evident, this descent search algorithm based upon derivative information is very much like Newton's method for root finding both in one-dimension as well as higher-dimensions. Figure 4.3 shows the rapid convergence for this convex function to the minimum. Moreover, the gradient descent algorithm is the core algorithm of advanced iterative solvers such as the bi-conjugate gradient descent method (**bicgstab** and generalized method of residuals (**gmres**) [203].

In the example illustrated, the gradient could be computed analytically. More generally, given just data itself, the gradient can be computed with numerical algorithms. The **gradient** command can be used to compute local or global gradients. Figure 4.4 shows the gradient terms $\partial f/\partial x$ and $\partial f/\partial y$ for the two functions shown in Fig. 4.2. The code used to produce these critical terms for the gradient descent algorithm is given by

Code 4.7: Computation of gradients.

```

[dfx,dfy]=gradient(f,dx,dy);

```

where the function $f(x,y)$ is a two-dimensional function computed from a known function or from data directly. The output are matrices containing the values of $\partial f/\partial x$ and $\partial f/\partial y$ over the discretized domain. The gradient can then be used to approximate either local or global gradients to execute the gradient descent. The following code, whose results are shown in Fig. 4.5, uses the **interp2** function to extract the values of the function and gradient of the function in Fig. 4.2(b).

Code 4.8: Gradient descent example using interpolation.

```

x(1)=x0(jj); y(1)=y0(jj);
f(1)=interp2(X,Y,F,x(1),y(1));
dfx=interp2(X,Y,dFx,x(1),y(1));
dfy=interp2(X,Y,dFy,x(1),y(1));

```

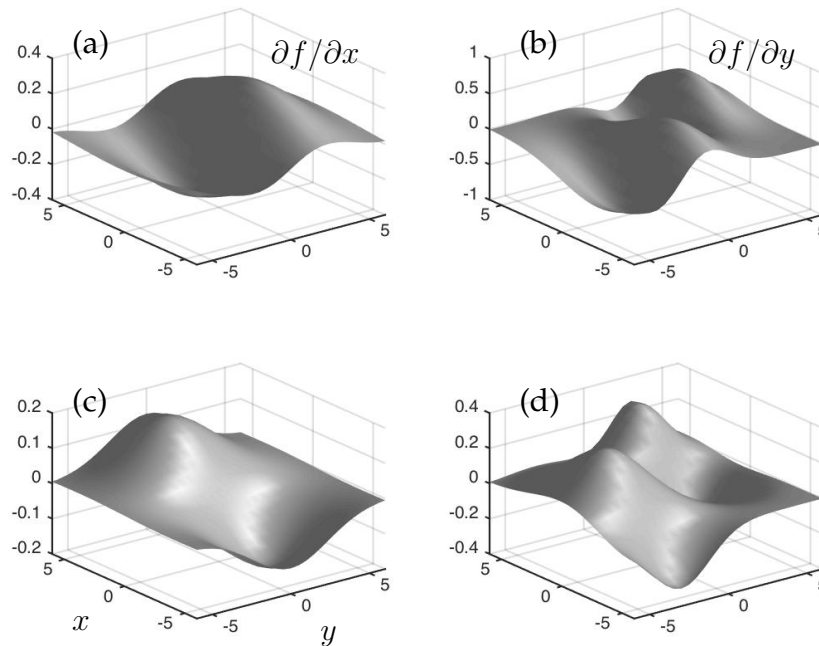


Figure 4.4: Computation of the gradient for the two functions illustrated in Fig. 4.2. In the left panels, the gradient terms (a) $\partial f/\partial x$ and (c) $\partial f/\partial y$ are computed for Fig. 4.2(a), while the right panels compute these same terms for Fig. 4.2(b) in (b) and (d) respectively. The **gradient** command numerically generates the gradient.

```

for j=1:10
    tau=fminsearch('tausearch',0.2,[],x(end),y(end),dfx,dfy,
        X,Y,F); % optimal tau
    x(j+1)=x(j)-tau*dfx; % update x, y, and f
    y(j+1)=y(j)-tau*dfy;
    f(j+1)=interp2(X,Y,F,x(j+1),y(j+1));
    dfx=interp2(X,Y,dFx,x(j+1),y(j+1));
    dfy=interp2(X,Y,dFy,x(j+1),y(j+1));

    if abs(f(j+1)-f(j))<10^(-6) % check convergence
        break
    end
end
end

```

In this code, the **fminsearch** command is used to find the correct value of τ . The function which attempts to optimize the size of the iterative step is given by

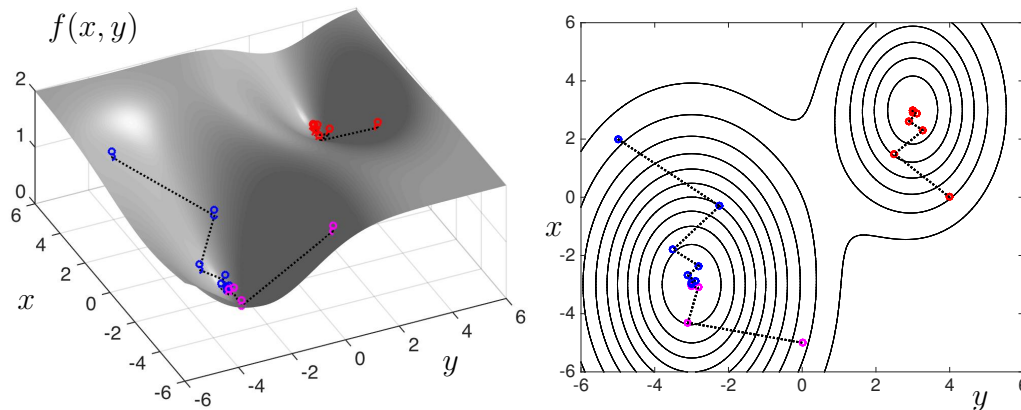


Figure 4.5: Gradient descent applied to the function featured in Fig. 4.2(b). Three initial conditions are shown: $(x_0, y_0) = \{(4, 0), (0, -5), (-5, 2)\}$. The first of these (red circles) gets stuck in a local minima while the other two initial conditions (blue and magenta) find the global minima. Interpolation of the gradient functions of Fig. 4.4 are used to update the solutions.

Code 4.9: Optimal step size τ .

```
function mintau=tausearch(tau,x,y,dfx,dfy,X,Y,F)
x0=x-tau*dfx;
y0=y-tau*dfy;
mintau=interp2(X,Y,F,x0,y0);
```

The above discussion provides a rudimentary discussion of gradient descent. A wide range of innovations have attempted to speed up this dominant nonlinear optimization procedure. This will be discussed further in the neural network chapter where gradient descent plays a critical role in training a network. For now, one can see that there are a number of issues for this nonlinear optimization procedure including determining the initial guess, step size τ and computing the gradient efficiently.

4.3 Regression and $Ax = b$: Over- and under-determined systems

Curve fitting, as shown in the previous two sections, results in an optimization problem. In many cases, the optimization can be mathematically framed as solving $Ax = b$. Before proceeding to discuss model selection and the various optimization methods available for this problem, it is instructive to consider that in many circumstances in modern data science, the linear system $Ax = b$ is typically massively over- or under-determined. In the former case, there are

generally no solutions satisfying the linear system, and instead, minimizing the error is advocated. In the latter case, there are an infinite number of solutions, and some choice of constraint must be made in order to select an appropriate and unique solution. The goal of this section is to highlight two different norms (ℓ_2 and ℓ_1) used for optimization that are relevant for $\mathbf{Ax} = \mathbf{b}$ for over and under-determined systems, and to demonstrate that the choice of norms has a profound impact on the optimal solutions achieved.

Before proceeding further, it should be noted that the $\mathbf{Ax} = \mathbf{b}$ considered here is a restricted instance of $\mathbf{Y} = f(\mathbf{X}, \beta)$ in (4.4). Thus the solution \mathbf{x} contains the *loadings* or *leverage scores* relating the relationship between the input data \mathbf{A} and outcome data \mathbf{b} . A simple solution for this linear problem capitalizes on the commonly used Moore-Penrose pseudo inverse operator \mathbf{A}^\dagger :

$$\mathbf{x} = \mathbf{A}^\dagger \mathbf{b}. \quad (4.38)$$

This operator is easily computed with the `pinv(A)` command in MATLAB. However, such a solution is restrictive, and a greater degree of flexibility is sought for model selection. Our particular aim in this section is to demonstrate the interplay in solving over- and under-determined systems using the ℓ_1 and ℓ_2 norms.

Over-determined Systems

Figure 4.6 shows the general structure of over-determined systems. As already stated, there are generally no solutions that satisfy $\mathbf{Ax} = \mathbf{b}$. Thus the optimization problem to be solved involves minimizing the error, E , for fitting the desired linear system:

$$\min_{\mathbf{x}} E = \argmin_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2. \quad (4.39)$$

Note that we have kept the least-square ℓ_2 norm for measuring the error.

This basic architecture does not explicitly enforce any constraints on the loadings \mathbf{x} . In order to both minimize the error and enforce a constraint on the solution, the basic optimization architecture can be modified to the following

$$\min_{\mathbf{x}} E = \argmin_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2 + \lambda_1 \|\mathbf{x}\|_1 + \lambda_2 \|\mathbf{x}\|_2 \quad (4.40)$$

where the parameters λ_1 and λ_2 control the penalization of the ℓ_1 and ℓ_2 norms respectively. This now explicitly enforces a constraint on the solution vector itself, not just the error. The ability to design the penalty by adding constraints is critical for model selection.

In the examples that follows, a particular focus will be given to the role of the ℓ_1 norm. The ℓ_1 norm, as already shown in Chapter 3, promotes sparsity so

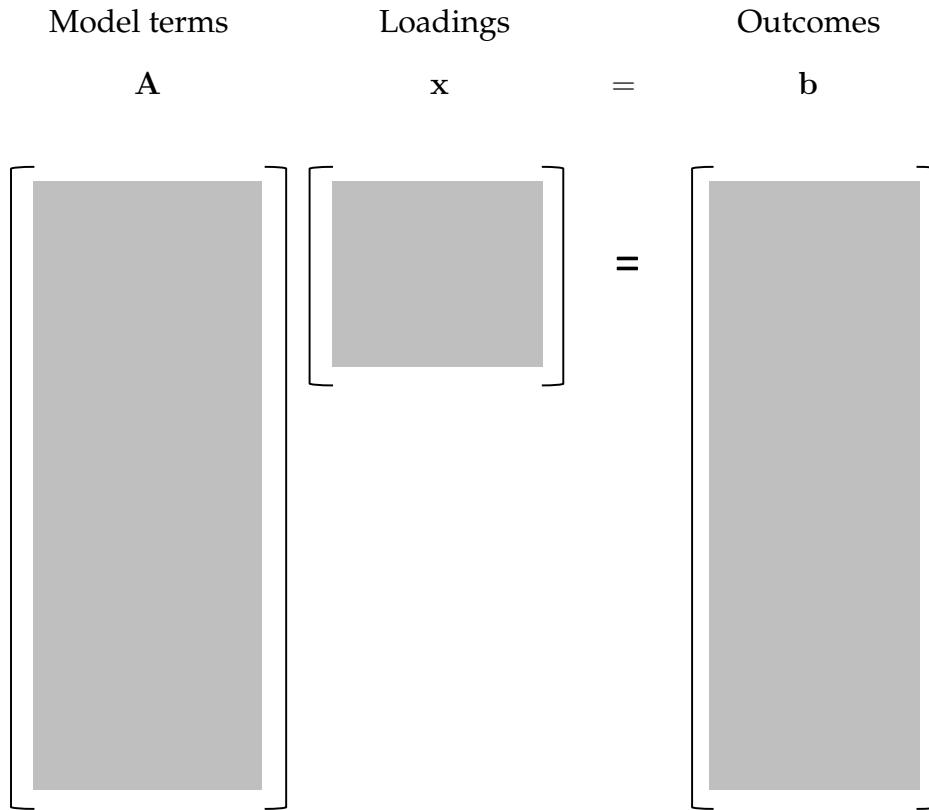


Figure 4.6: Regression framework for overdetermined systems. In this case, $\mathbf{Ax} = \mathbf{b}$ cannot be satisfied in general. Thus finding solutions for this system involves minimizing, for instance, the least-square error $\min E = \operatorname{argmin}_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2$ subject to some constraint on the solution \mathbf{x} such as minimizing the ℓ_2 norm $\min \|\mathbf{x}\|_2$.

that many of the loadings of the solution \mathbf{x} are zero. This will play an important role in variable and model selection in the next section. For now, consider solving the optimization problem (4.40) with $\lambda_2 = 0$. We use the convex optimization package `cvx`, which is an open source MATLAB package for convex optimization [202], for computing our solution to (4.40). The following code considers various values of the ℓ_1 penalization in producing solutions to an over-determined systems with 500 constraints and 100 unknowns.

Code 4.10: Solutions for over-determined system

```
n=500; m=100;
```

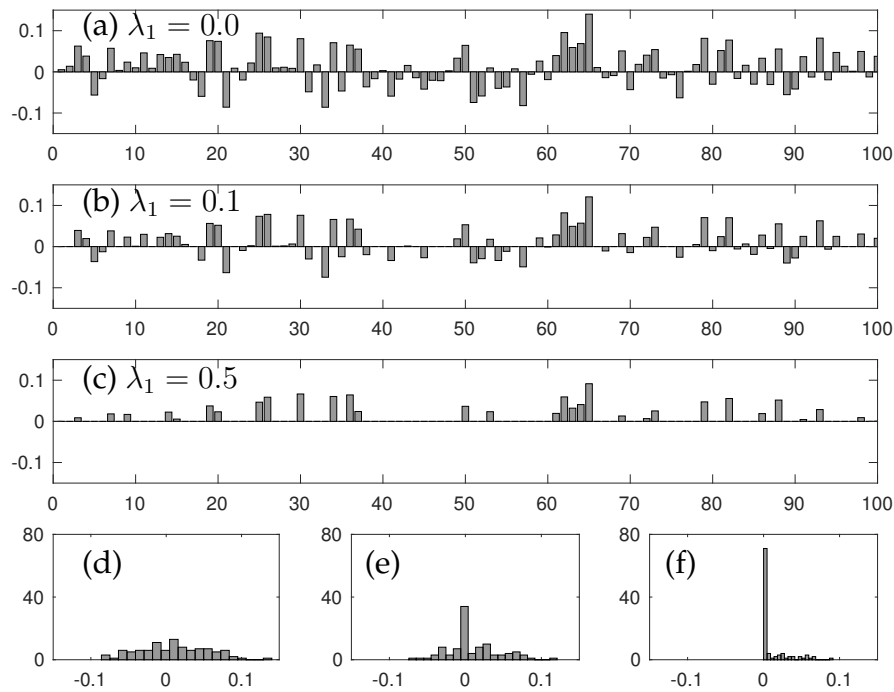



Figure 4.7: Solutions to an overdetermined system with 500 constraints and 100 unknowns. Panels (a)-(c) show a bar plot of the values of the loadings of the vectors x . Note that as the ℓ_1 penalty is increased from (a) $\lambda_1 = 0$ to (b) $\lambda_1 = 0.1$ to (c) $\lambda_1 = 0.5$, the number of zero elements of the vector increases, i.e. it becomes more sparse. A histogram of the loading values for (a)-(c) is shown in the panels (d)-(f) respectively. This highlights the role that the ℓ_1 norm plays in promoting sparsity in the solution.

```

A=rand(n,m);
b=rand(n,1);
xdag=pinv(A)*b;

lam=[0 0.1 0.5];
for j=1:3

    cvx_begin;
        variable x(m)
        minimize( norm(A*x-b,2) + lam(j)*norm(x,1) );
    cvx_end;

    subplot(4,1,j),bar(x,'FaceColor',[.6 .6 .6],'EdgeColor','k')
    subplot(4,3,9+j), hist(x,20)
end

```

Figure 4.7 highlights the results of the optimization process as a function of

the parameter λ_1 . It should be noted that the solution with $\lambda_1 = 0$ is equivalent to the solution `xdag` produced from computing the pseudo-inverse of the matrix A . Note that the ℓ_1 norm promotes a sparse solution where many of the components of the solution vector x are zero. The histograms of the solutions values of x in Fig. 4.7(d)-(f) are particularly revealing as they show the sparsification process for increasing λ_1 .

The regression for over-determined systems can easily be generalized to matrix systems as shown in Fig. 4.6. In this case, the `cvx` command structure simply modifies the size of the matrix b and solution matrix x . Consider the two solutions of an over-determined matrix system generated from the following code.

Code 4.11: Solutions for over-determined matrix system

```
n=300; m=60; p=20;
A=rand(n,m); b=rand(n,p);

lam=[0 0.1];
for j=1:2
    cvx_begin;
        variable x(m,p)
        minimize (norm(A*x-b,2) + lam(j)*norm(x,1));
    cvx_end;
    subplot(2,1,j), pcolor(x.'), colormap(hot), colorbar
end
```

Figure 4.8 shows the results of this matrix over-determined systems for two different values of the added ℓ_1 penalty. Note that the addition of the ℓ_1 norm sparsifies the solution and produces a matrix which is dominated by zero entries. The two examples in Figs. 4.7 and 4.8 show the important role that the ℓ_2 and ℓ_1 norms have in generating different types of solutions. In the following sections of this book, these norms will be exploited to produce parsimonious models of data-driven problems.

Under-determined Systems

For undetermined systems, there are an infinite number of possible solutions satisfying $Ax = b$. The goal in this case is to impose an additional constraint, or set of constraints, whereby a unique solution is generated from the infinite possibilities. The basic mathematical structure is shown in Fig. 4.9. As an optimization, the solution to the under-determined system can be stated as

$$\min \|x\|_p \text{ subject to } Ax = b \quad (4.41)$$

where the p denotes the p -norm of the vector x . For simplicity, we consider here the ℓ_2 and ℓ_1 norms only. As has already been shown for over-determined

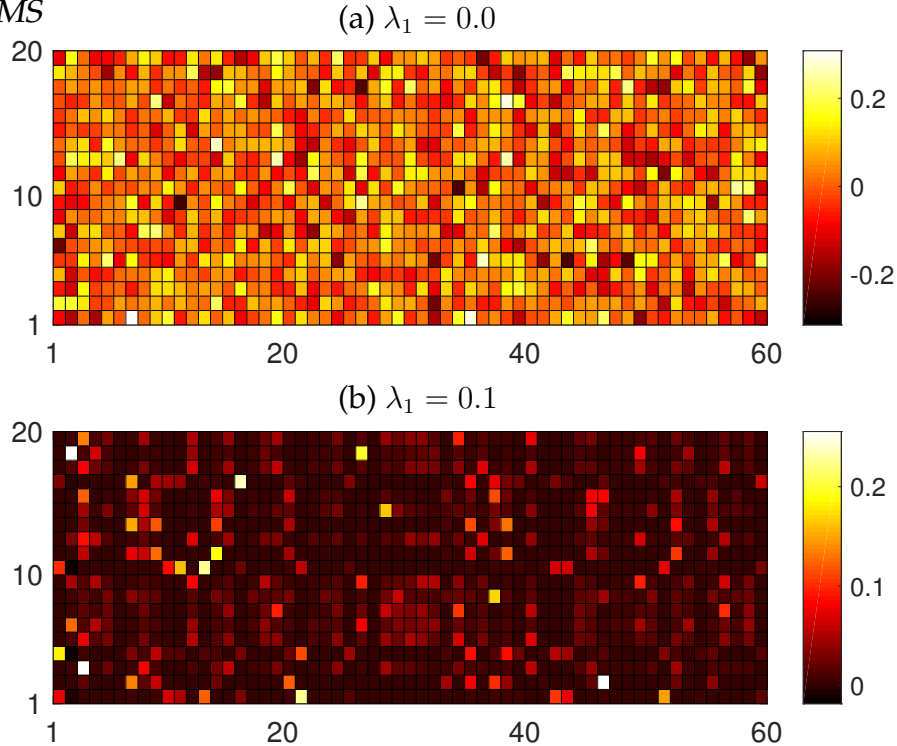


Figure 4.8: Solutions to an overdetermined system $Ax = b$ with 300 constraints and 60×20 unknowns. Panels (a) and (b) show a plot of the values of the loadings of the matrix x with ℓ_1 penalty (a) $\lambda_1 = 0$ to (b) $\lambda_1 = 0.1$.

systems, the ℓ_1 norm promotes sparsity of the solution.

We again use the convex optimization package `cvx` for computing our solution to (4.42). The following code considers both ℓ_2 and ℓ_1 penalization in producing solutions to an under-determined systems with 20 constraints and 100 unknowns.

Code 4.12: Solutions for under-determined system

```
n=20; m=100
A=rand(n,m); b=rand(n,1);

cvx_begin;
    variable x2(m)
    minimize( norm(x2,2) );
    subject to
        A*x2 == b;
cvx_end;

cvx_begin;
    variable x1(m)
    minimize( norm(x1,1) );
```

$$\begin{array}{ccc}
 \text{Model terms} & & \text{Loadings} \\
 \mathbf{A} & & \mathbf{x} \\
 & & = \\
 & & \mathbf{b} \\
 & & \text{Outcomes}
 \end{array}$$

Figure 4.9: Regression framework for underdetermined systems. In this case, $\mathbf{Ax} = \mathbf{b}$ can be satisfied. In fact, there are an infinite number of solutions. Thus pinning down a unique solution for this system involves minimizing a constraint. For instance, from an infinite number of solutions, we choose the one that minimizes the ℓ_2 norm $\min \|x\|_2$, which is subject to the constraint $\mathbf{Ax} = \mathbf{b}$.

```

|| subject to
||     A*x1 == b;
|| cvx_end;

```

This code produces two solution vectors \mathbf{x}_2 and \mathbf{x}_1 which minimize the ℓ_2 and ℓ_1 norm respectively. Note the way that `cvx` allows one to impose constraints in the optimization routine. Figure 4.10 shows a bar plot and histogram of the two solutions produced. As before, the sparsity promoting ℓ_1 norm yields a solution vector dominated by zeros. In fact, for this case, there are exactly 80 zeros for this linear system since there are only 20 constraints for the 100 unknowns

As with the over-determined system, the optimization can be easily modified to handle more general under-determined matrix equations as shown in Fig. 4.9. The `cvx` optimization engine is easily modified for this case like it was for the over-determined systems. The software engine can also work with more general p -norms as well as minimize with both ℓ_1 and ℓ_2 penalties simulta-

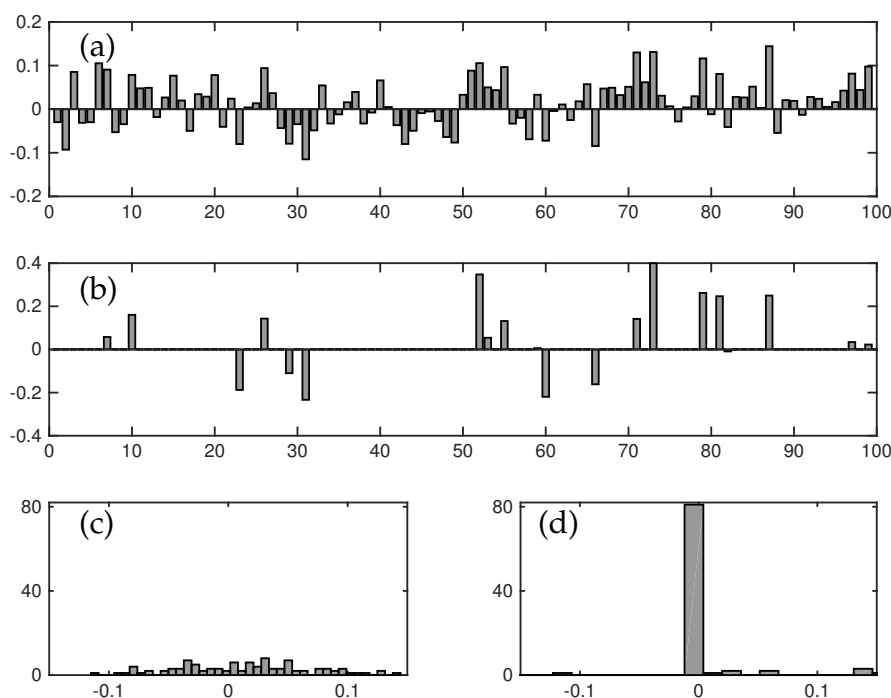


Figure 4.10: Solutions to an under-determined system with 20 constraints and 100 unknowns. Panels (a) and (b) show a bar plot of the values of the loadings of the vectors x . In the former panel, the optimization is subject to minimizing the ℓ_2 norm of the solution, while the latter panel is subject to minimizing the ℓ_1 norm. Note that the ℓ_1 penalization produces a sparse solution vector. A histogram of the loading values for (a) and (b) is shown in the panels (c) and (d) respectively.

neously. For instance, a common optimization modifies (4.42) to the following

$$\min (\lambda_1 \|x\|_1 + \lambda_2 \|x\|_2) \quad \text{subject to} \quad \mathbf{Ax} = \mathbf{b} \quad (4.42)$$

where the weighting between λ_1 and λ_2 can be used to promote a desired sparsification of the solution. These different optimization strategies are common and will be considered further in what follows.

4.4 Optimization as the cornerstone of regression

In the previous two sections of this chapter, the fitting function $f(x)$ was specified. For instance, it may be desirable to produce a line fit so that $f(x) = Ax + B$. The coefficients are then found by the regression and optimization methods already discussed. In what follows, our objective is to develop techniques which allow us to objectively select a good model for fitting the data, i.e. should one

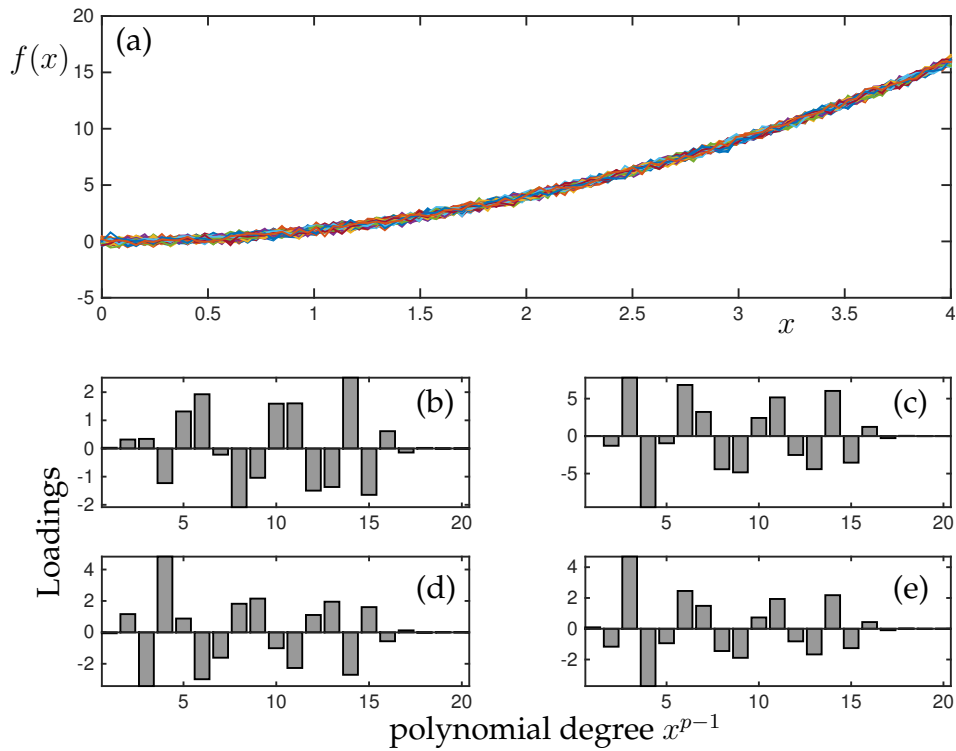


Figure 4.11: (a) One hundred realizations of the parabolic function (4.43) with additive white noise parametrized by $\sigma = 0.1$. Although the noise is small, the least-square fitting procedure produces significant variability when fitting to a polynomial of degree twenty. Panels (b)-(e) demonstrate the loadings (coefficients) for the various polynomial coefficients for four different noise realizations. This demonstrated model variability frames the model selection architecture.

use a quadratic or cubic fit? The error metric alone does not dictate a good model selection as the more terms that are chosen for fitting, the more parameters are available for lowering the error regardless of whether the additional terms have any meaning or interpretability.

Optimization strategies will play a foundational role in extracting interpretable results and meaningful models from data. As already shown in previous sections, the interplay of the ℓ_2 and ℓ_1 norms has a critical impact on the optimization outcomes. To illustrate further the role of optimization and the variety of outcomes possible, consider the simple example of data generated from noisy measurements of a parabola

$$f(x) = x^2 + W(0, \sigma) \quad (4.43)$$

where $W(0, \sigma)$ is a normally distributed random variable with mean zero and standard deviation σ . Figure 4.11(a) shows an example of 100 random measurements of (4.43). The parabolic structure is clearly evident despite the noise

added to the measurement. Indeed, a parabolic fit is trivial to compute using classic least-square fitting methods outlined in the first section of this chapter.

The goal is to *discover* the best model for the data given. So instead of specifying a simple model, in practice, we do not know what the function is and need to extract it if possible. We can begin by positing a regression to a set of polynomial models. In particular, consider framing the model selection problem $\mathbf{Y} = f(\mathbf{X}, \boldsymbol{\beta})$ of (4.4) as the following $\mathbf{Ax} = \mathbf{b}$ problem:

$$\begin{bmatrix} | & | & | & \cdots & | \\ 1 & x_j & x_j^2 & \cdots & x_j^{p-1} \\ | & | & | & \cdots & | \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_p) \end{bmatrix} \quad (4.44)$$

where the matrix \mathbf{A} contains polynomial models up to degree $p - 1$ with each row representing a measurement, the β_k are the coefficients for each polynomial, and the matrix \mathbf{b} contains the outcomes (data) $f(x_j)$. In what follows, we will consider a scenario where 100 measurements are taken and 20 term (19th order) polynomial is fit. Thus the matrix system $\mathbf{Ax} = \mathbf{b}$ results in an over-determined system as illustrated in Fig. 4.6.

The following code solves the over-determined system (4.44) using least-square regression via the `pinv` function. For this case, four realizations are run in order to illustrate the impact that a small amount of noise has on the regression procedure.

Code 4.13: Least-square polynomial fit to parabola with noise

```
n=100; L=4;
x=linspace(0,L,n);
f=(x.^2).'; % parabola with 100 data points

M=20; % polynomial degree
for j=1:M
    phi(:,j)=(x.').^(j-1); % build matrix A
end

for j=1:4
    fn=(x.^2+0.1*randn(1,n)).';
    an=pinv(phi)*fn; fna=phi*an; % least-square fit
    En=norm(f-fna)/norm(f);
    subplot(4,2,4+j),bar(an,'FaceColor',[.6 .6 .6],'
        EdgeColor','k')
end
```

Figure 4.11(b)-(e) shows four typical loadings $\boldsymbol{\beta}$ computed from the regression procedure. Note that despite the low-level of noise added, the loadings are

significantly different from one another. Thus each noise realization produces a very different model to explain the data.

The variability of the regression results are problematic for model selection. It suggests that even a small amount of measurement noise can lead to significantly different conclusions about the underlying model. In what follows, we quantify this variability while also considering various regression procedures for solving the over-determined linear system $\mathbf{Ax} = \mathbf{b}$. Highlighted here are five standard methods: least-square regression (**pinv**), the backslash operator (`\`), LASSO (least absolute shrinkage and selection operator) (**lasso**), robust fit (**robustfit**), and ridge regression (**ridge**). Returning to the last section, and specifically (4.40), helps frame the mathematical architecture for these various $\mathbf{Ax} = \mathbf{b}$ solvers. Specifically, the Moore-Penrose pseudo-inverse (**pinv**) solves (4.40) with $\lambda_1 = \lambda_2 = 0$. The backslash command (`\`) for over-determined systems solves the linear system via a QR decomposition [479]. The LASSO (**lasso**) solves (4.40) with $\lambda_1 > 0$ and $\lambda_2 = 0$. Ridge regression (**ridge**) solves (4.40) with $\lambda_1 = 0$ and $\lambda_2 > 0$. However, the standard implementation of ridge in MATLAB is a bit more nuanced. Note that the popular elastic net algorithm weights both the ℓ_2 and ℓ_1 penalty, thus providing a tunable hybrid model regression between ridge and LASSO regression. Robust fit (**robustfit**) solves (4.40) by a weighted least-square fitting. It moreover allows one to leverage robust statistics methods and penalize according to the Huber norm so as to promote outlier rejection [239]. In the data considered here, no outliers are imposed on the data so that the power of robust fit is not properly leveraged. Regardless, it is an important technique one should consider.

Figure 4.12 shows a series of box plots for 100 realizations of data that illustrate the differences with the various regression techniques considered. It also highlights critically important differences with optimization strategies based upon the ℓ_2 and ℓ_1 norm. From a model selection point of view, the least-square fitting procedure produces significant variability in the loading parameters β as illustrated in Fig. 4.12, panels (a), (b) and (e). The least-square fitting was produced by the Moore-Penrose pseudo-inverse or QR decomposition respectively. If some ℓ_1 penalty (regularization) is allowed, then Fig. 4.12, panels (d), (d) and (f), show that a more parsimonious model is selected with low variability. This is expected as the ℓ_1 norm sparsifies the solution vector of loading values β . Indeed, the standard LASSO regression nicely selects the quadratic polynomial as the dominant contribution to the data. The following code was used to generate this data.

Code 4.14: Comparison of regression methods

```
subplot(2,1,1)
lambda=0.1; phi2=phi(:,2:end);
for jj=1:100
    f=(x.^2+0.2*randn(1,n)).';
```

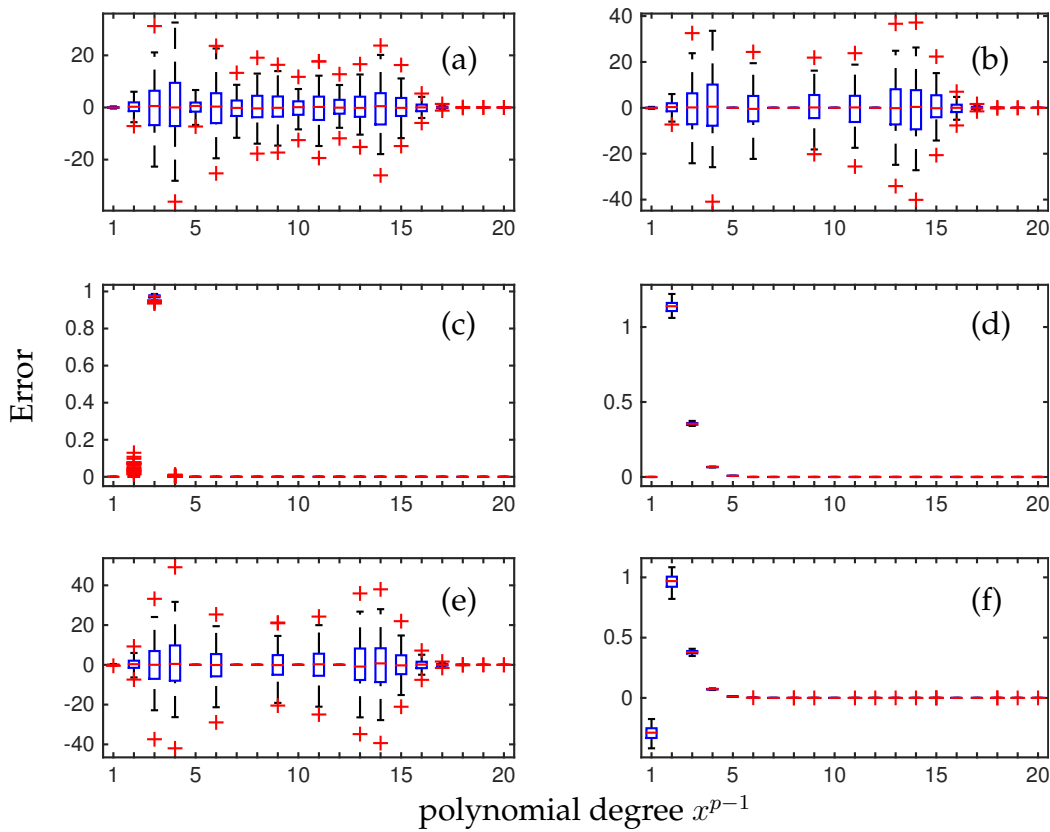



Figure 4.12: Comparison of regression methods for $\mathbf{Ax} = \mathbf{b}$ for an over-determined system of linear equations. The 100 realizations of data are generated from a simple parabola (4.43) that is fit to a 20th degree polynomial via (4.44). The box plots show (a) least-square regression via the Moore-Penrose pseudo-inverse (**pinv**), (b) the backslash command (****), (c) LASSO regression (**lasso**), (d) LASSO regression with different ℓ_2 versus ℓ_1 penalization, (e) robust fit, and (f) ridge regression. Note the significant variability in the loading values for the strictly ℓ_2 based methods ((a), (b) and (e)), and the low-variability for ℓ_1 weighted methods ((c), (d) and (f)). Only the standard LASSO (c) identifies the dominance of the parabolic term.

```

a1=pinv(phi)*f; f1=phi*a1; E1(jj)=norm(f-f1)/norm(f);
a2=phi\f; f2=phi*a2; E2(jj)=norm(f-f2)/norm(f);
[a3,stats]=lasso(phi,f,'Lambda',lambda); f3=phi*a3; E3(jj)
    =norm(f-f3)/norm(f);
[a4,stats]=lasso(phi,f,'Lambda',lambda,'Alpha',0.8); f4=
    phi*a4; E4(jj)=norm(f-f4)/norm(f);
a5=robustfit(phi2,f); f5=phi*a5; E5(jj)=norm(f-f5)/norm(f)
;

```

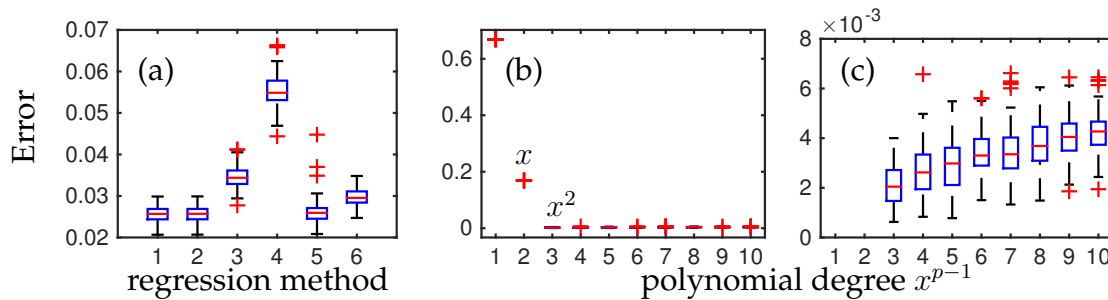


Figure 4.13: (a) Comparison of the error for the six regression methods used in Fig. 4.12. Despite the variability across the optimization methods, all of them produce low-error solutions. (b) Error using least-square regression as a function of increasing degree of polynomial. The error drops rapidly until the quadratic term is used in the regression. (c) Detail of the error showing that the error actually increases slightly by using a higher-degree of polynomial to fit the data.

```

a6=ridge(f,phi2,0.5,0); f6=phi*a6; E6(jj)=norm(f-f6)/norm(f);
    f);

A1(:,jj)=a1;
A2(:,jj)=a2;
A3(:,jj)=a3;
A4(:,jj)=a4;
A5(:,jj)=a5;
A6(:,jj)=a6;
plot(x,f), hold on
end
Err=[E1; E2; E3; E4; E5; E6];
Err2=[E1; E2; E3; E4; E5];

```

Note that this code also produces the 100 realizations that are visualized in Fig. 4.11(a).

Despite the significant variability exhibited in Fig. 4.12 for most of the loading values by the different regression techniques, the error produced in the fitting procedure has little variability. Moreover, the various methods all produce regressions that have comparable error. Thus despite their differences in optimization frameworks, the error from fitting is relatively agnostic to the underlying method. This suggests that using the error alone as a metric for model selection is potentially problematic since almost any method can produce a reliable, low-error model. Figure 4.13(a) shows a box plot of the error produced using the regression methods of Fig. 4.12. All of the regression techniques produce comparably low error and low variability results using significantly different strategies.

As a final note to this section and the code provided, we can consider in-

stead the regression procedure as a function of the number of polynomials in (4.44). In our example of Fig. 4.12, polynomials up to degree 20 were considered. If instead, we sweep through polynomial degrees, then something interesting and important occurs as illustrated in Fig. 4.13(b)-(c). Specifically, the error of the regression collapses to 10^{-3} after the quadratic term is added as shown in panel (b). This is expected since the original model was a quadratic function with a small amount of noise. Remarkably however, as more polynomial terms are added, the error actually increases in the regression procedure as highlighted in panel (c). Thus simply adding more terms does not improve the error, which is counter-intuitive at first. The code to produce these results are given by the following:

Code 4.15: Model fitting

```
M=10;
En=zeros(100,M);
for jj=1:M

    for j=1:jj
        phi(:,j)=(x.').^(j-1);
    end

    f=(x.^2).';
    for j=1:100
        fn=(x.^2+0.1*randn(1,n)).';
        an=pinv(phi)*fn; fna=phi*an;
        En(j,jj)=norm(f-fna)/norm(f);
    end
end
```

Note that we have only swept through polynomials up to degree 10. Note further that panel (c) of Fig. 4.13 is a detail of panel (b). The error produced by a simple parabolic fit is approximately twice as good as a polynomial with degree 10. These results will help frame our model selection framework of the remaining sections.

4.5 The Pareto front and *Lex Parsimoniae*

The preceding chapters show that regression is more nuanced than simply choosing a model and performing a least-square fit. Not only are there numerous metrics for constraining the solution, the model itself should be carefully selected in order to achieve a better, more interpretable understanding of the data. Such considerations on an appropriate model date back to William of

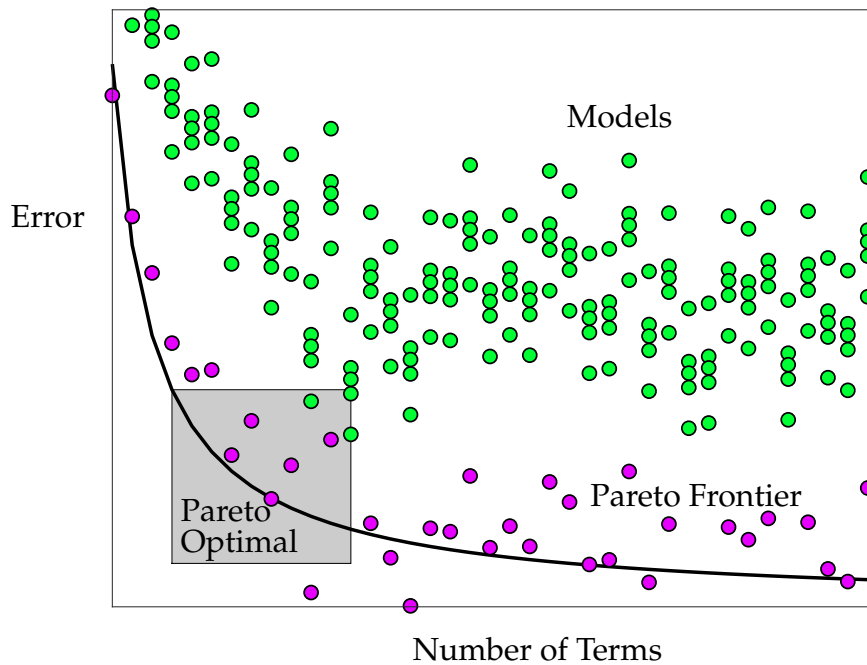


Figure 4.14: For model selection, the criteria of accuracy (low error) is balanced against parsimony. There can be a variety of models with the same number of terms (green and magenta points), but the *Pareto Frontier* (magenta points) is defined by the envelope of models that produce the lowest error for a given number of terms. The solid line provides an approximation to the Pareto frontier. The *Pareto optimal* solutions (shaded region) are those models that produce accurate models while remaining parsimonious.

Occam (c. 1287–1347), who was an English Franciscan friar, scholastic philosopher, and theologian. Occam proposed his law of parsimony (in latin *lex parsimoniae*), commonly known as Occam’s razor, whereby he stated that among competing hypotheses, the one with the fewest assumptions should be selected, or when you have two competing theories that make exactly the same predictions, the simpler one is the better. The philosophy of Occam’s razor has been used extensively throughout the physical and biological sciences for developing governing equations to model observed phenomena.

Parsimony also plays a central role in the mathematical work of Vilfredo Pareto (c. 1848–1923). Pareto was an Italian engineer, sociologist, economist, political scientist, and philosopher. He made several important contributions to economics, specifically in the study of income distribution and in the analysis of individuals’ choices. He was also responsible for popularizing the use of the term *elite* in social analysis. In more recent times, he has become known for the popular 80/20 rule which is qualitatively illustrated in Fig. 4.14, named after him as the Pareto principle by management consultant Joseph M. Juran in

1941. Stated simply, it is a common principle in business and consulting management that, for instance, observes that 80% of sales come from 20% of clients. This concept was popularized by Richard Koch's book *The 80/20 Principle* [270] (along with several follow-up books [271, 272, 273]), which illustrated a number of practical applications of the Pareto principle in business management and life, ultimately advocated a way to maximize business efficiency.

Pareto and Occam ultimately advocated the same philosophy: explain the majority of observed data with a parsimonious model. Importantly, model selection is not simply about reducing error, it is about producing a model that has a high degree of interpretability, generalization and predictive capabilities. Figure 4.14 shows the basic concept of the *Pareto Frontier* and *Pareto Optimal* solutions. Specifically, for each model considered, the number of terms and the error in matching the data is computed. The solutions with the lowest error for a given number of terms define the Pareto frontier. Those solutions that optimally balance error and parsimony are in the shaded region and represent the Pareto optimal solutions. In game theory, the Pareto optimal solution is thought of as a strategy that cannot be made to perform better against one opposing strategy without performing less well against another (in this case error and parsimony). In economics, it describes a situation in which the profit of one party cannot be increased without reducing the profit of another. Our objective is to select, in a principled way, the best model from the space of Pareto optimal solutions. Towards this end, information criteria (which will be discussed in subsequent sections) will be used to select from candidate modes in the Pareto optimal region.

Overfitting

The Pareto concept needs amending when considering application to real data. Specifically, when building models with many free parameters, as is often the case in machine learning applications with high-dimensional data, it is easy to overfit the data. Indeed, the increase in error illustrated in Fig. 4.13(c) as a function of increasing model complexity illustrates this point. Thus unlike what is depicted in Fig. 4.14 where the error goes towards zero as the number of model terms (parameters) is increased, the error actually increases when considering models with a higher number of terms and/or parameters. To determine the (best) correct model, various cross-validation and model selection algorithms are necessary.

To illustrate the overfitting that occurs real data, consider the simple example of the last section. In this example, we are simply trying to find the correct parabolic model measured with additive noise (4.43). The results of Figs. 4.13(b) and 4.13(c) already indicate that overfitting is occurring for polynomial models beyond second order. The following MATLAB example will

highlight the effects of overfitting. Consider the following code that produces a training and test set for the parabola of (4.43). The training set is on the region $x \in [0, 4]$ while the test set (extrapolation region) will be for $x \in [4, 8]$.

Code 4.16: Parabolic model with training and test data

```
n=200; L=8;
x=linspace(0,L,n);
x1=x(1:100); % train
x2=x(101:200); % test
n1=length(x1);
n2=length(x2);
ftrain=(x1.^2).'; % train parabola x=[0,4]
ftest=(x2.^2).'; % test parabola x=[4,8]
figure(1), subplot(3,1,1),
plot(x1,ftrain,'r',x2,ftest,'b','Linewidth',[2])
```

This code produces the ideal model on two distinct regions: $x \in [0, 4]$ and $x \in [4, 8]$. Once measurement noise is added to the model, then the parameters for a polynomial fit no longer produce the perfect parabolic model. We can compute for given noisy measurements both an interpolation error, where measurements are taken in the data regime of $x \in [0, 4]$, and extrapolation error, where measurements are taken in the data regime of $x \in [4, 8]$. For this example, a least squares regression is performed using the pseudo-inverse (**pinv**) from MATLAB.

Code 4.17: Overfitting a quadratic model

```
M=30; % number of model terms
Eni=zeros(100,M); Ene=zeros(100,M);
for jj=1:M
    for j=1:jj
        phi_i(:,j)=(x1.').^(j-1); % interpolation key
        phi_e(:,j)=(x2.').^(j-1); % extrapolation key
    end

    f=(x.^2).';
    for j=1:100
        fni=(x1.^2+0.1*randn(1,n1)).'; % interpolation
        fne=(x2.^2+0.1*randn(1,n2)).'; % extrapolation

        ani=pinv(phi_i)*fni; fnai=phi_i*ani;
        Eni(jj,jj)=norm(ftrain-fnai)/norm(ftrain);

        fnae=phi_e*ani; % use loadings from x in [0,4]
        Ene(jj,jj)=norm(ftest-fnae)/norm(ftest);
    end
end
```

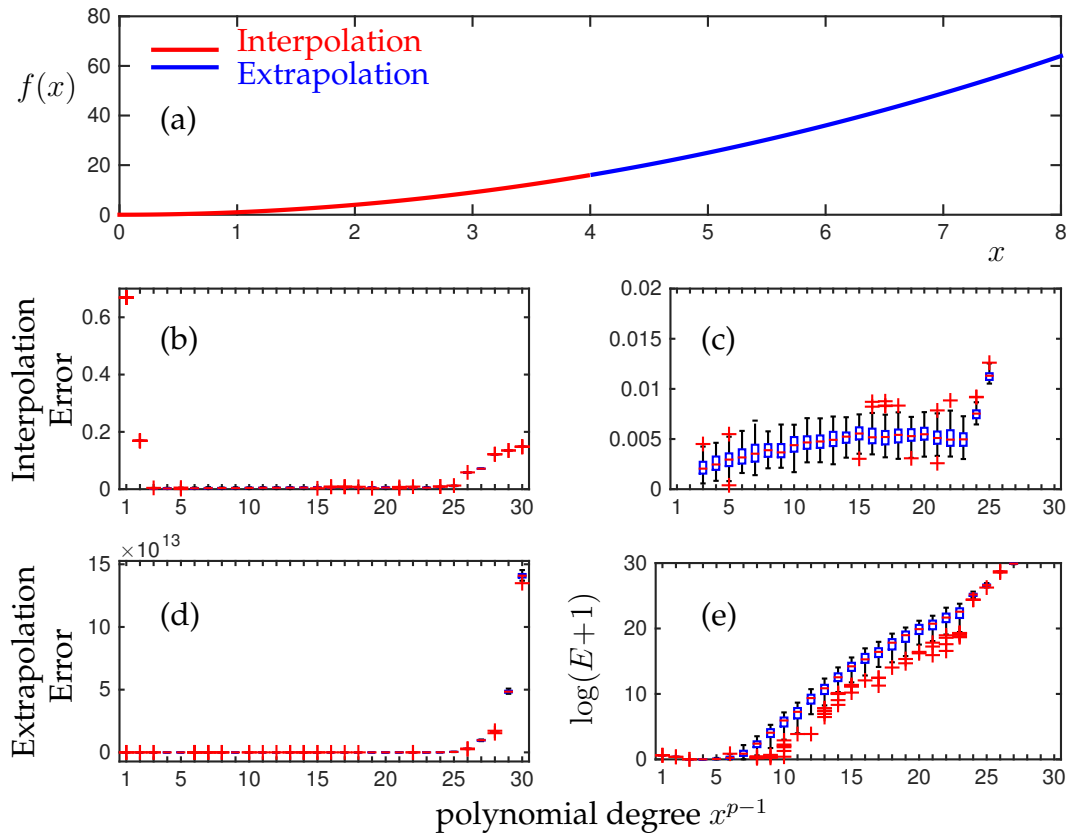


Figure 4.15: (a) The ideal model $f(x) = x^2$ over the domain $x \in [0, 8]$. Data is collected in the region $x \in [0, 4]$ in order to build a polynomial regression model (4.44) with increasing polynomial degree. In the interpolation regime $x \in [0, 4]$, the model error stays constrained, with increasing error due to overfitting for polynomials of degree greater than 2. The error is shown in panel (b) with a zoom in of the error in panel (c). For extrapolation, $x \in [4, 8]$, the error grows exponentially beyond a parabolic fit. In panel (d), the error is shown to grow to 10^{13} . A zoom in of the region on a logarithmic scale of the error ($\log(E+1)$ where unity is added so that zero error produces a zero score) shows the exponential growth of error. This clearly shows that the model trained on the interval $x \in [0, 4]$ does not generalize (extrapolate) to the region $x \in [4, 8]$. This example should serve as a serious warning and note of caution in model fitting.

|| end

This simple example shows some of the most basic and common features associated with overfitting of models. Specifically, overfitting does not allow for generalization. Consider the results of Fig. 4.15 generated from the above

code. In this example, the least-square loadings (4.44) for an M th degree polynomial are computed using the pseudo-inverse for data in the range $x \in [0, 4]$. The interpolation error for these loadings are demonstrated in Figs. 4.15(b) and (c). Note the impact of overfitting by polynomials for this interpolation of the data. Specifically, the error of the interpolated fit increases from beyond a second degree polynomial. Extrapolation for an overfit model produces significant errors. Figures 4.15(d) and (e) show the error growth as a function of the least-square fit M th degree polynomial model. The error in Fig. 4.15(d) is on a logarithmic plot since it grows to 10^{13} . This demonstrates a clear inability of the overfit model to generalize the model to the range $x \in [4, 8]$. Indeed, only a parsimonious model with a 2nd degree polynomial can easily generalize to the range $x \in [4, 8]$ while keeping the error small.

The above example shows that some form of model selection to systematically deduce a parsimonious model is critical for producing viable models that can generalize outside of where data is collected. Much of machine learning revolves around (i) using data to generate predictive models, and (ii) cross-validation techniques to remove the most deleterious effects of overfitting. Without a cross-validation strategy, one will almost certainly produce a non-generalizable model such as that exhibited in Fig. 4.15. In what follows, we will consider some standard strategies for producing reasonable models.

4.6 Model selection: Cross validation

The previous section highlights many of the fundamental problems with regression. Specifically, it is easy to overfit the data with a model, thus leading to a perfectly good model for interpolation purposes, but incapable of generalizing for extrapolation. This is an especially pernicious issue in training deep neural nets. To overcome the consequences of overfitting, various techniques have been proposed to more appropriately select a parsimonious model with only a few parameter terms, thus balancing the error with a model that can more easily generalize, or extrapolate. This provides a reinterpretation of the Pareto front in Fig. 4.14. Specifically, the error as the number of terms grows increases dramatically due to overfitting, especially when tested against extrapolation.

There are two common mathematical strategies for circumventing the effects of overfitting in model selection: *cross-validation* and computing *information criteria*. This section considers the former, while the later method is considered in the next section. Cross-validation strategies are perhaps the most common technique in the computer science literature, and they are critical in almost all machine learning algorithms. Indeed, one should never trust a model unless properly cross-validated. Cross-validation can be stated quite simply:

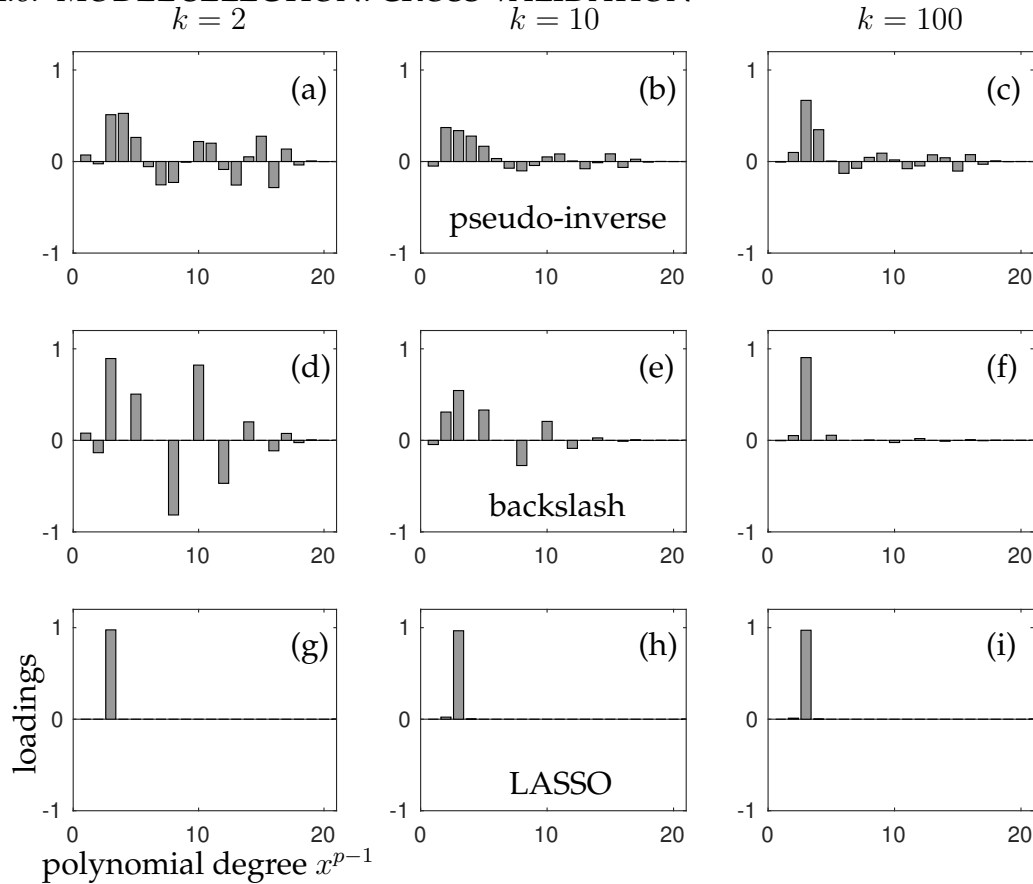


Figure 4.16: Cross-validation using k -fold strategy with $k = 2, 20$ and 100 (left, middle and right columns respectively). Three different regression strategies are cross-validated: least-square fitting of pseudo-inverse, the QR-based backslash, and the sparsity promoting LASSO. Note that the LASSO for this example produces the quadratic model within even a one or two fold validation. The backslash based QR algorithm has a strong signature after 100-fold cross-validation, while the least-square fitting suggests that the quadratic and cubic terms are both important even after 100-fold cross-validation.

Take random portions of your data and build a model. Do this k times and average the parameter scores (regression loadings) to produce the cross-validated model. Test the model predictions against withheld (extrapolation) data and evaluate whether the model is actually any good. This commonly used strategy is called k -fold cross-validation. It is simple, intuitively appealing, and the k -fold model building procedure produces a statistically based model for evaluation.

To illustrate the concept of cross-validation, we will once again consider fitting polynomial models to the simple function $f(x) = x^2$ (See Fig. 4.16). The previous sections of this chapter have already considered this problem in de-

tail, both from the various regression frameworks available (pseudo-inverse, LASSO, robust fit, etc.), as well as their ability to accurately produce a model for interpolating and extrapolating data. The following MATLAB code considers three regression techniques (least-square fitting of pseudo-inverse, the QR-based backslash, and the sparsity promoting LASSO) for k -fold cross-validation ($k = 2, 20$ and 100). In this case, one can think of the k snapshots of data as trial measurements. As one might expect, there would be an advantage as more trials are taken and $k = 100$ models are averaged for a final model.

Code 4.18: k -means

```
n=100; L=4;
x=linspace(0,L,n);
f=(x.^2).'; % parabola with 100 data points

M=21; % polynomial degree
for j=1:M
    phi(:,j)=(x.').^(j-1); % build matrix A
end

trials=[2 10 100];
for j=1:3

    for jj=1:trials(j)
        f=(x.^2+0.2*randn(1,n)).';
        a1=pinv(phi)*f; f1=phi*a1; E1(jj)=norm(f-f1)/norm(f);
        a2=phi\f; f2=phi*a2; E2(jj)=norm(f-f2)/norm(f);
        [a3,stats]=lasso(phi,f,'Lambda',0.1); f3=phi*a3; E3(jj)=
            norm(f-f3)/norm(f);

        A1(:,jj)=a1;
        A2(:,jj)=a2;
        A3(:,jj)=a3;
    end
    A1m=mean(A1. ');
    A2m=mean(A2. ');
    A3m=mean(A3. ');
    Err=[E1; E2; E3];

    subplot(3,3,j), bar(A1m), axis([0 21 -1 1.2])
    subplot(3,3,3+j), bar(A2m), axis([0 21 -1 1.2])
    subplot(3,3,6+j), bar(A3m), axis([0 21 -1 1.2])

end
```

Figure 4.16 shows the results of the k -fold cross-validation computations.

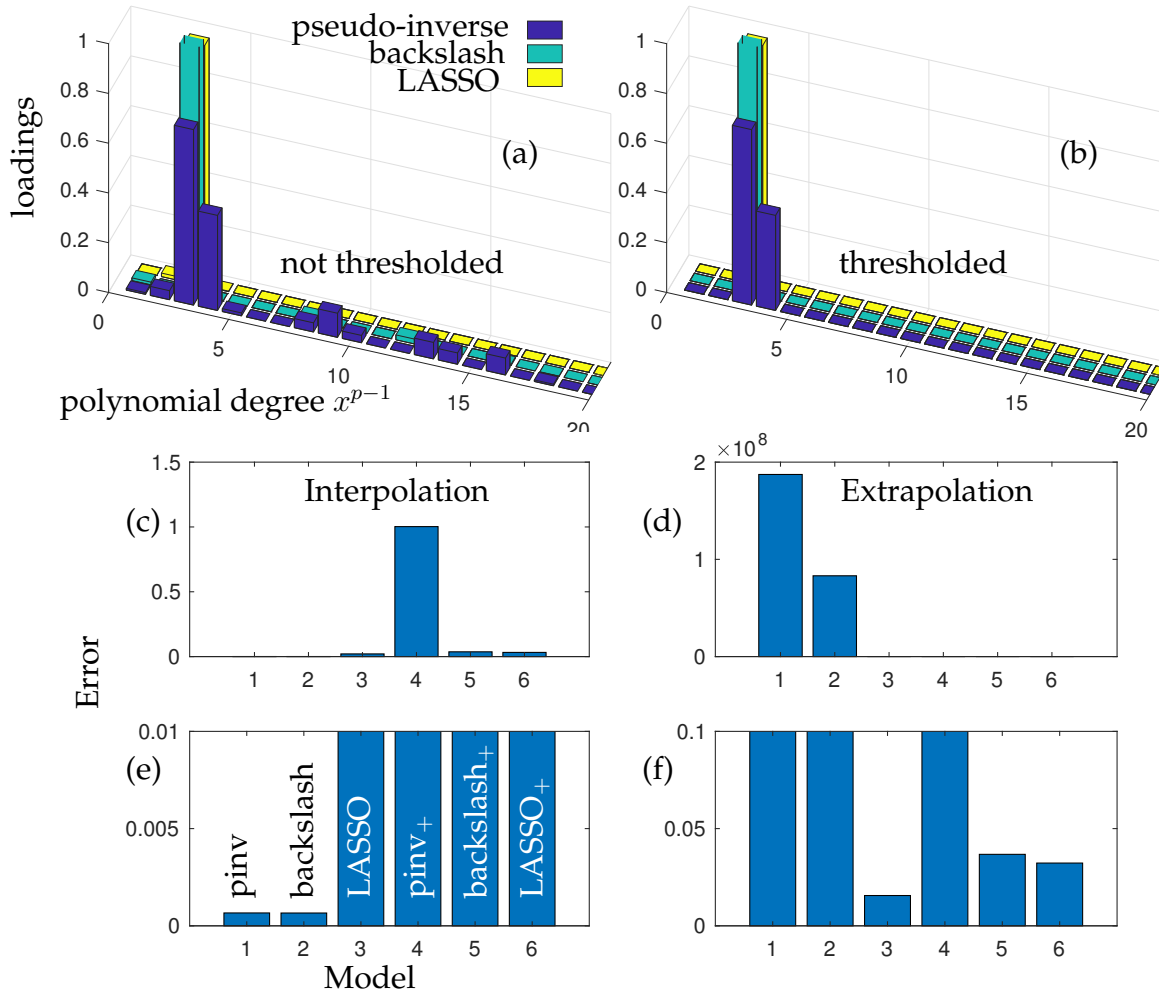


Figure 4.17: Error and loading results for $k = 100$ fold cross-validation. The loadings for the k -fold validation (with thresholding denoted by subscript +, (b) and without (a) thresholding) are shown for least-square fitting of pseudo-inverse, the QR-based backslash, and the sparsity promoting LASSO (See Fig. 4.16). Both the (c) interpolation error (and detail in (e)) and (d) extrapolation error (and detail in (f)) are computed. The LASSO performs well for both interpolation and extrapolation while a least-square fit gives poor performance under extrapolation. The 6 models considered are: 1. pseudo-inverse, 2. backslash, 3. LASSO, 4. thresholded pseudo-inverse, 5. thresholded backslash, and 6. thresholded LASSO.

By promoting sparsity (parsimony), the LASSO achieves the desired quadratic model after even a single $k = 1$ fold (i.e. thus this is not even cross-validated). In contrast the least-square regression (pseudo-inverse) and QR-based regression both require a significant number of folds to produce the dominant quadratic

term. The least-square regression, even after $k = 100$ folds, still advocates for the importance of both a quadratic and cubic term.

The final model selection process under k -fold cross-validation often can involve a *thresholding* of terms that are small in the regression. The above code demonstrates the regression on three regression strategies. Although the LASSO looks almost ideal, it still has a small contributing linear component. The QR strategy of backslash produces a number of small components scattered among the polynomials used in the fit. The least-square regression has the dominant quadratic and cubic terms with a large number of non-zero coefficients scattered across the polynomials. If one thresholds the loadings, then the LASSO and backslash will produce exactly the quadratic model, while the least-square fit produces a quadratic-cubic model. The following code thresholds the loading coefficients and then produces the final cross-validated model. This model can then be evaluated against both the interpolated and extrapolated data regions as in Fig. 4.17.

Code 4.19: Comparison of cross-validated models

```
Atot=[A1m; A2m; A3m]; % average loadings of three methods
Atot2=(Atot>0.2).*Atot; % threshold
Atot3=[Atot; Atot2]; % combine both thresholded and not

figure(3), bar3(Atot.'), axis([0 4 0 20 0 1]), view(-70,38),
    set(gca,'Xtick',[0,2,4],'FontSize',[18])
figure(4), bar3(Atot2.'), axis([0 4 0 20 0 1]), view(-70,38),
    set(gca,'Xtick',[0,2,4],'FontSize',[18])

n=200; L=8;
x=linspace(0,L,n);
x1=x(1:100); % train (interpolation)
x2=x(101:200); % test (extrapolation)

ftrain=(x1.^2).'; % interpolated parabola x=[0,4]
ftest=(x2.^2).'; % extrapolated parabola x=[4,5]

for j=1:M
    phi_i(:,j)=(x1.^j-1).'; % interpolation key
    phi_e(:,j)=(x2.^j-1).'; % extrapolation key
end

for jj=1:6 % compute inter/extra-polation scores
    ani=Atot3(jj,:).';
    fnai=phi_i*ani;
    Eni(jj)=norm(ftrain-fnai)/norm(ftrain);
    fnae=phi_e*ani;
```

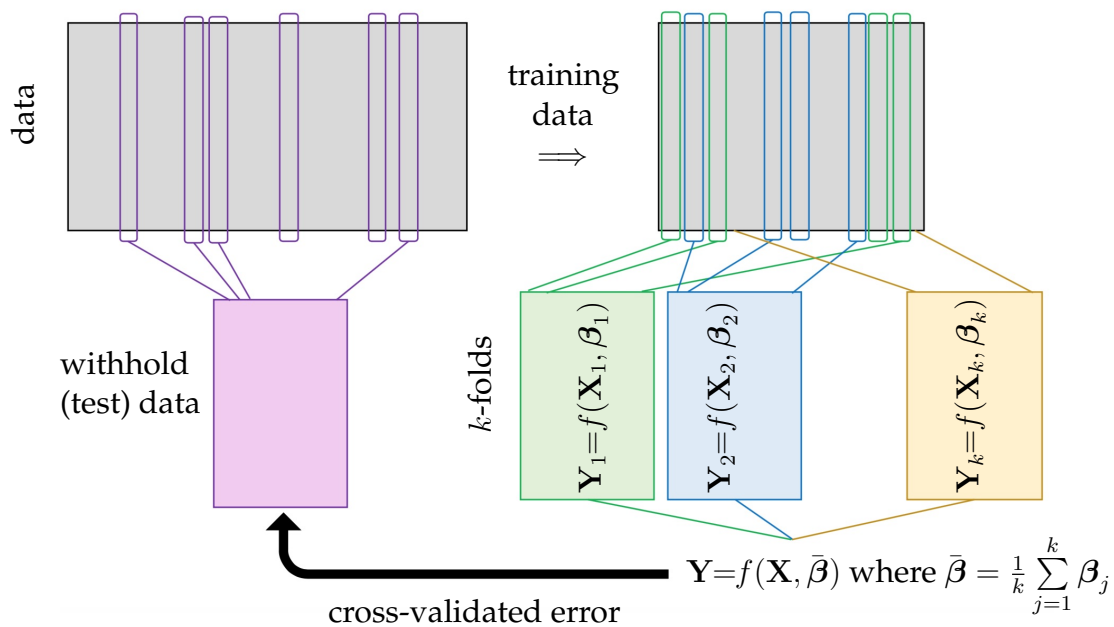


Figure 4.18: Procedure for k -fold cross-validation of models. The data is initially partitioned into a training and test (withhold) set. Typically the withhold set is generated from a random sample of the overall data. The training data is partitioned into k -folds whereby a random sub-selection of the training data is collected in order to build a regression model $Y_j = f(X_j, \beta_j)$. Importantly, each model generates the loading parameters β_j . After the k -fold models are generated, the average model (or leading) is produced $Y = f(X, \bar{\beta})$ where $\bar{\beta} = (1/k) \sum_{j=1}^k \beta_j$. This model is then tested on the withheld data to evaluate its viability.

```

|| En e ( j j ) = norm ( ftest - fnae ) / norm ( ftest ) ;
|| end

```

The results of Fig. 4.17 show that the model selection process, and the regression technique used, makes a critical difference in producing a viable model. It further shows that despite a k -fold cross-validation, the extrapolation error (or generalizability of the model) can still be poor. A good model is one that keeps errors small and also generalizes well, something that the LASSO does well in producing a model for the example considered in this chapter.

k -fold cross-validation

The process of k -fold cross validation is highlighted in Fig. 4.18. The concept is to partition a data set into a training set and a test set. The test set, or withhold set, is kept separate from any training procedure for the model. Importantly,

the test set is where the model produces an extrapolation approximation, which the figures of the last two sections show to be challenging. In k -fold cross-validation, the training data is further partitioned into k -folds, which are typically randomly selected portions of the data. For instance, in standard 10-fold cross validation, the training data is randomly partitioned into 10 partitions (or folds). Each partition is used to construct a regression model $\mathbf{Y}_j = f(\mathbf{X}_j, \beta_j)$ for $j = 1, 2, \dots, 10$. The average of the loading values $\bar{\beta} = (1/k) \sum_{j=1}^k \beta_j$ are then used for the final, cross-validated regression model $\mathbf{Y} = f(\mathbf{X}, \bar{\beta})$. This model is then used on the withhold data to test its extrapolation power, or generalizability. The error on this withhold test set is what determines the efficacy of the model.

Leave p -out cross-validation

Another standard technique for cross-validation involves the so-called *leave p -out cross validation* (LpO CV). In this case, p -samples of the data are removed from the data and kept as the withhold set. A model is built on the remaining data and the accuracy of the model is tested in the p withheld samples. This is repeated with a new selection of p samples until all the data has been part of the withheld data set. The accuracy of the model is then evaluated from averaging the accuracy of the models and the loadings produced from the various partitions of the data.

4.7 Model selection: Information criteria

There is a different approach to model selection than the cross-validation strategies outlined in the previous section. Indeed, model selection has a rigorous set of mathematical innovations starting from the early 1950s. The Kullback-Leibler (KL) divergence [288] measures the distance between two probability density distributions (or data sets which represent the truth and a model) and is the core of modern information theory criteria for evaluating the viability of a model. The KL divergence has deep mathematical connections to statistical methods characterizing entropy as developed by Ludwig E. Boltzmann (c. 1844-1906), as well as a relation to information theory developed by Claude Shannon [442]. Model selection is a well developed field with a large body of literature, most of which is exceptionally well reviewed by Burnham and Anderson [96]. In what follows, only brief highlights will be given as relates to demonstrating some of the standard methods.

The KL divergence between two models $f(\mathbf{X}, \beta)$ and $g(\mathbf{X}, \mu)$ is defined as

$$I(f, g) = \int f(\mathbf{X}, \beta) \log \left[\frac{f(\mathbf{X}, \beta)}{g(\mathbf{X}, \mu)} \right] d\mathbf{X} \quad (4.45)$$

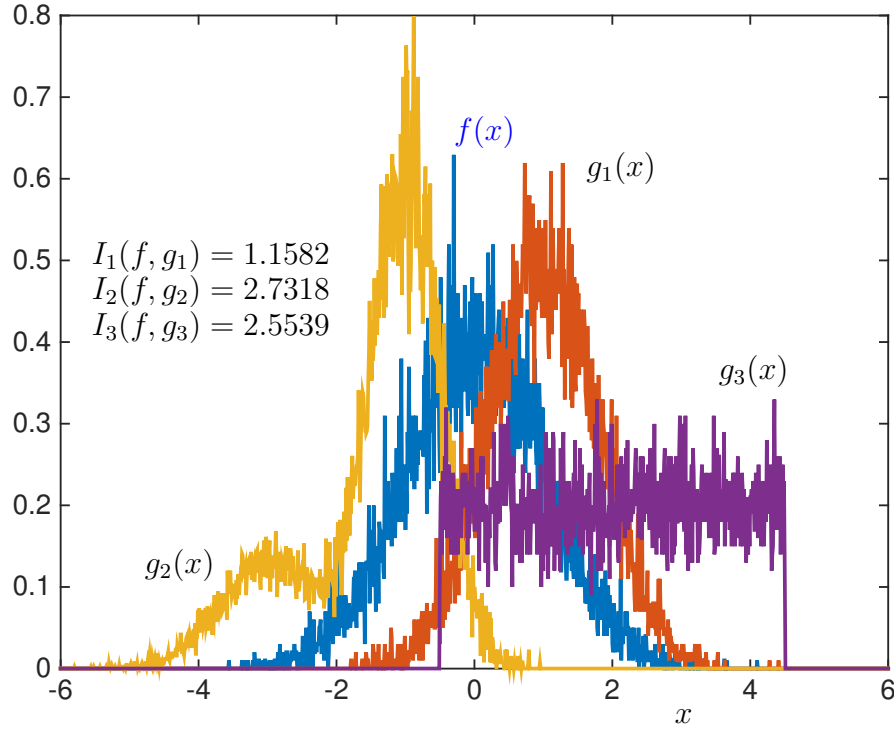


Figure 4.19: Comparison of three models $g_1(x)$, $g_2(x)$ and $g_3(x)$ against the truth model $f(x)$. The KL divergence $I_j(f, g_j)$ for each model is computed, showing that the model $g_1(x)$ is closest to statistically representing the true data.

where β and μ are parameterizations of the models $f(\cdot)$ and $g(\cdot)$ respectively. From an information theory perspective, the quantity $I(f, g)$ measures the information lost when g is used to represent f . Note that if $f = g$, then the log term is zero (i.e. $\log(1) = 0$) and $I(f, g) = 0$ so that there is no information lost. In practice, f will represent the *truth*, or measurements of an experiment, while g will be a model proposed to model f .

Unlike the regression and cross-validation performed previously, when computing KL divergence a model must be specified. Recall that we used cross-validation previously to generate a models using different regression strategies (See Fig. 4.18 for instance). Here a number of models will be posited and the loss of information, or KL divergence, of each model will be computed. The model with the lowest loss of information is generally regarded as the best model. Thus given M proposed models $g_j(\mathbf{X}, \mu_j)$ where $j = 1, 2, \dots, M$, we can compute $I_j(f, g_j)$ for each model. The correct model, or best model, is the one that minimizes the information loss $\min_j I_j(f, g_j)$.

As a simple example, consider Fig. 4.19 which shows three different models that are compared to the truth data. To generate this figure, the following code was used. The computation of the KL divergence score is also illustrated.

Note that in order to avoid a divide by zero, a constant offset is added to each probability distribution. The truth data generated, $f(x)$, is a simple normally distributed variable. The three models shown are variants of normally (and uniformly) distributed functions.

Code 4.20: Computation of KL divergence

```
n=10000;
x1=randn(n,1); % "truth" model (data)
x2=0.8*randn(n,1)+1; % model 1
x3=0.5*randn(n,1)-1; % model 3 components
x4=0.7*randn(n,1)-3;
x5=5*rand(n,1)-0.5;
x=-6:0.01:6; % range for data

f=hist(x1,x)+0.01; % generate PDFs
g1=hist(x2,x)+0.01;
g2a=hist(x3,x);
g2b=hist(x4,x);
g2=g2a+0.3*g2b+0.01;
g3=hist(x5,x)+0.01;

f=f/trapz(x,f); % normalize data
g1=g1/trapz(x,g1);
g2=g2/trapz(x,g2);
g3=g3/trapz(x,g3);

plot(x,f,x,g1,x,g2,x,g3,'Linewidth',[2])
Int1=f.*log(f./g1); % compute integrand
Int2=f.*log(f./g2);
Int3=f.*log(f./g3);

% use if needed
%Int1(isinf(Int1))=0; Int1(isnan(Int1))=0;
%Int2(isinf(Int2))=0; Int2(isnan(Int2))=0;

I1=trapz(x,Int1) % KL divergence
I2=trapz(x,Int2)
I3=trapz(x,Int3)
```

Information Criteria: AIC and BIC

This simple example shows the basic ideas behind model selection: compute a distance between a proposed model output $g_j(x)$ and the measured truth $f(x)$. In the early 1970s, Hirotugu Akaike combined Fisher's maximum like-

likelihood computation [169] with the KL divergence score to produce what is now called the *Akaike Information Criterion* (AIC) [6]. The was later modified by Gideon Schwarz to the so-called *Bayesian Information Criterion* BIC [436] which provided an information score that was guaranteed to converge to the correct model provided the model was included in the evaluation and a sufficiently large amount of data was included.

To be more precise, we turn to Akaike's seminal contribution [6]. Akaike was aware that KL divergence in practice cannot be computed since it requires full knowledge of the statistics of the truth model $f(x)$ and of all the parameters in the proposed models $g_j(x)$. Thus Akaike instead proposed a way to estimate KL divergence based upon the empirical log-likelihood function at its maximum point. This is computable in practice and was a critically enabling insight for rigorous methods of model selection. The technical aspects of Akaike's work connecting log-likelihood estimates and KL divergence [6, 96] was a paradigm shifting mathematical achievement, and thus led to the development of the AIC score

$$AIC = 2K - 2 \log [\mathcal{L}(\hat{\mu}|\mathbf{x})], \quad (4.46)$$

where K is the number of parameters used in the model, $\hat{\mu}$ is an estimate of the best parameters used (i.e. lowest KL divergence) in $g(\mathbf{X}, \mu)$ computed from a *maximum likelihood estimate* (MLE), and \mathbf{x} are independent samples of the data to be fit. Thus instead of a direct measure of the distance between two models, the AIC provides an estimate of the relative distance between the approximating model and the true model (or data). Note that as the number of terms gets large in a proposed model, its AIC score increases with slope $2K$, thus providing a penalty for non-parsimonious models. Importantly, due to its relative measure, it will always select a best model with the lowest AIC score, but this best model may still be quite poor in prediction and reconstruction of the data.

AIC is one of the standard model selection criteria used today. However, there are others. Highlighted here is the modification of AIC by Gideon Schwarz to construct BIC [436]. BIC is almost identical to AIC aside from the penalization of the information criteria by the number of terms. Specifically, BIC is defined as

$$BIC = \log(n)K - 2 \log [\mathcal{L}(\hat{\mu}|\mathbf{x})], \quad (4.47)$$

where n is the number of data points, or sample size, considered. This slightly different version of the information criteria has one significant consequence. The seminal contribution of Schwarz was to prove that if the correct (truth) model was included along with a set of candidate models, then it would be theoretically guaranteed to be selected as the best model based upon BIC for sufficiently large set of data \mathbf{x} . This is in contrast to AIC for which in certain pathological cases, it can select the wrong model.

Suggested reading

Texts

- (1) **Model Selection and Multimodel Inference**, by K. P. Burnham and D. R. Anderson [96].
- (2) **Multivariate analysis**, by R. A. Johnson and D. Wichern, 2002 [244].
- (3) **An Introduction to Statistical Learning**, by G. James, D. Witten, T. Hastie and R. Tibshirani, 2013 [242].

Papers and reviews

- (1) **On the mathematical foundations of theoretical statistics.**, by R. A. Fisher, *Philosophical Transactions of the Royal Society of London*, 1922 [169].
- (2) **A new look at the statistical model identification.**, by H. Akaike, *IEEE Transactions on Automatic Control*, 1974 [6].
- (3) **Estimating the dimension of a model.**, by G. Schwarz et al., *The annals of statistics*, 1978 [436].
- (4) **On information and sufficiency.**, by S. Kullback and R. A. Leibler, *The annals of statistics*, 1951 [288].
- (5) **A mathematical theory of communication.**, by C. Shannon, *ACM SIGMOBILE Mobile Computing and Communications Review*, 2001 [436].