Chapter 9

Linear and Logistic Regression

> An unsophisticated forecaster uses statistics as a drunken man uses lamp posts – for support rather than illumination.
>
> – Andrew Lang

Linear regression is the most representative "machine learning" method to build models for value prediction and classification from training data. It offers a study in contrasts:

- Linear regression has a beautiful theoretical foundation yet, in practice, this algebraic formulation is generally discarded in favor of faster, more heuristic optimization.

- Linear regression models are, by definition, linear. This provides an opportunity to witness the limitations of such models, as well as develop clever techniques to generalize to other forms.

- Linear regression simultaneously encourages model building with hundreds of variables, and regularization techniques to ensure that most of them will get ignored.

Linear regression is a bread-and-butter modeling technique that should serve as your baseline approach to building data-driven models. These models are typically easy to build, straightforward to interpret, and often do quite well in practice. With enough skill and toil, more advanced machine learning techniques might yield better performance, but the possible payoff is often not worth the effort. Build your linear regression models first, then decide whether it is worth working harder to achieve better results.

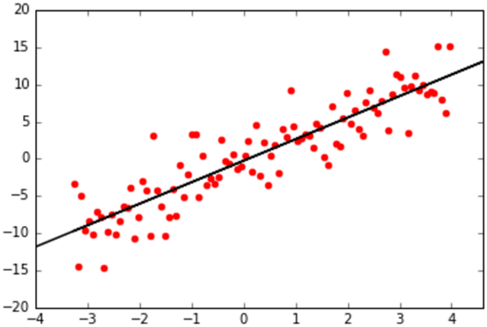

Figure 9.1: Linear regression produces the line which best fits a set of points.

9.1 Linear Regression

Given a collection of n points, linear regression seeks to find the line which best approximates or *fits* the points, as shown in Figure 9.1. There are many reasons why we might want to do this. One class of goals involves simplification and compression: we can replace a large set of noisy data points in the xy-plane by a tidy line that describes them, as shown in Figure 9.1. This regression line is useful for visualization, by showing the underlying trend in the data and highlighting the location and magnitude of outliers.

However, we will be most interested in regression as a method for value forecasting. We can envision each observed point $p = (x, y)$ to be the result of a function $y = f(x)$, where x represents the feature variables and y the independent target variable. Given a collection of n such points $\{p_1, p_2, \ldots, p_n\}$, we seek the $f(x)$ which best explains these points. This function $f(x)$ interpolates or models the points, providing a way to estimate the value y' associated with any possible x', namely that $y' = f(x')$.

9.1.1 Linear Regression and Duality

There is a connection between regression and solving linear equations, which is interesting to explore. When solving linear systems, we seek the single point that lies on n given lines. In regression, we are instead given n points, and we seek the line that lies on "all" points. There are two differences here: (a) the interchange of points for lines and (b) finding the best fit under constraints verses a totally constrained problem ("all" vs. all).

The distinction between points and lines proves trivial, because they both are really the same thing. In two-dimensional space, both points (s, t) and lines $y = mx + b$ are defined by two parameters: $\{s, t\}$ and $\{m, b\}$, respectively. Further, by an appropriate *duality* transformation, these lines are equivalent to

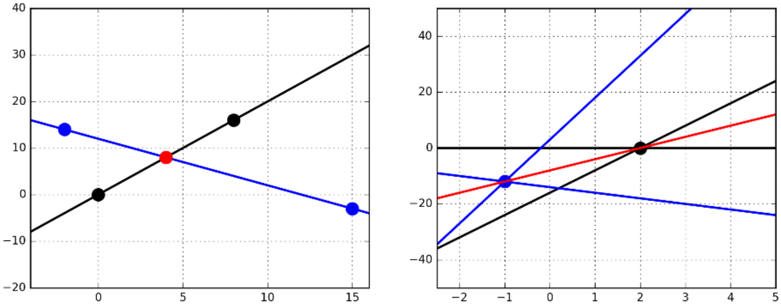

Figure 9.2: Points are equivalent to lines under a duality transform. The point $(4, 8)$ in red (left) maps to the red line $y = 4x - 8$ on right. Both sets of three collinear points on the left correspond to three lines passing through the same point on right.

points in another space. In particular, consider the transform that

$$(s, t) \longleftrightarrow y = sx - t.$$

Now any set of points that lie on a single line get mapped to a set of lines which intersect a singular point – so finding a line that hits all of a set of points is algorithmically the same thing as finding a point that hits all of a set of lines.

Figure 9.2 shows an example. The point of intersection in Figure 9.2 (left) is $p = (4, 8)$, and it corresponds to the red line $y = 4x - 8$ on the right. This red point p is defined by the intersection of black and blue lines. In the dual space, these lines turn into black and blue points lying on the red line. Three collinear points on left (red with either two black or two blue) map to three lines passing through a common point on the right: one red and two of the same color. This duality transformation reverses the roles of points and lines in a way that everything makes sense.

The big difference in defining linear regression is that we seek a line that *comes as close as possible* to hitting all the points. We must be careful about measuring error in the proper way in order to make this work.

9.1.2 Error in Linear Regression

The *residual error* of a fitted line $f(x)$ is the difference between the predicted and actual values. As shown in Figure 9.3, for a particular feature vector x_i and corresponding target value y_i, the residual error r_i is defined:

$$r_i = y_i - f(x_i).$$

This is what we will care about, but note that it is not the only way that error might have been defined. The closest distance to the line is in fact defined

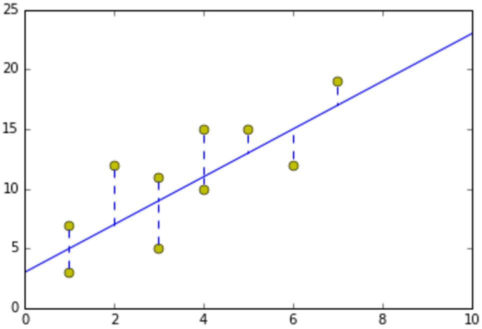

Figure 9.3: The residual error in least squares is the projection of $y_i - f(X)$ down to X, not the shortest distance between the line and the point.

by the perpendicular-bisector through the target point. But we are seeking to forecast the value of y_i from x_i, so the residual is the right notion of error for our purposes.

Least squares regression minimizes the sum of the squares of the residuals of all points. This metric has been chosen because (1) squaring the residual ignores the signs of the errors, so positive and negative residuals do not offset each other, and (2) it leads to a surprisingly nice closed form for finding the coefficients of the best-fitting line.

9.1.3 Finding the Optimal Fit

Linear regression seeks the line $y = f(x)$ which minimizes the sum of the squared errors over all the training points, i.e. the coefficient vector w that minimizes

$$\sum_{i=1}^{n}(y_i - f(x_i))^2, \text{ where } f(x) = w_0 + \sum_{i=1}^{m-1} w_i x_i$$

Suppose we are trying to fit a set of n points, each of which is m dimensional. The first $m-1$ dimensions of each point is the feature vector $(x_1, \ldots, x_{m-1})$, with the last value $y = x_m$ serving as the target or dependent variable.

We can encode these n feature vectors as an $n \times (m-1)$ matrix. We can make it an $n \times m$ matrix A by prepending a column of ones to the matrix. This column can be thought of as a “constant” feature, one that when multiplied by the appropriate coefficient becomes the y-intercept of the fitted line. Further, the n target values can be nicely represented in an $n \times 1$ vector b.

The optimal regression line $f(x)$ we seek is defined by an $m \times 1$ vector of coefficients $w = \{w_0, w_1, \ldots, w_{m-1}\}$. Evaluating this function on these points is exactly the product $A \cdot w$, creating an $n \times 1$ vector of target value predictions. Thus $(b - A \cdot w)$ is the vector of residual values.

How can we find the coefficients of the best fitting line? The vector w is given by:

$$w = (A^T A)^{-1} A^T b.$$

First, let's grok this before we try to understand it. The dimensions of the term on the right are

$$((m \times n)(n \times m))(m \times n)(n \times 1) \rightarrow (m \times 1).$$

which exactly matches the dimensions of the target vector w, so that is good. Further, $(A^T A)$ defines the covariance matrix on the columns/features of the data matrix, and inverting it is akin to solving a system of equations. The term $A^T b$ computes the dot products of the data values and the target values for each of the m features, providing a measure of how correlated each feature is with the target results. We don't understand why this works yet, but it should be clear that this equation is made up of meaningful components.

Take-Home Lesson: That the least squares regression line is defined by $w = (A^T A)^{-1} A^T b$ means that solving regression problems reduces to inverting and multiplying matrices. This formula works fine for small matrices, but the gradient descent algorithm (see Section 9.4) will prove more efficient in practice.

Consider the case of a single variable x, where we seek the best-fitting line of the form $y = w_0 + w_1 x$. The slope of this line is given by

$$w_1 = \sum_{i=1}^{n} \frac{(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2} = r_{xy} \frac{\sigma_x}{\sigma_y}$$

with $w_0 = \bar{y} - w_1 \bar{x}$, because of the nice observation that the best-fitting line passes through $(\bar{x}, \bar{y})$.

The connection with the correlation coefficient (r_{xy}) here is clear. If x were uncorrelated with y $(r_{xy} = 0)$, then w_1 should indeed be zero. Even if they were perfectly correlated $(r_{xy} = 1)$, we must scale x to bring it into the right size range of y. This is the role of σ_y / σ_x.

Now, where does the linear regression formula come from? It should be clear that in the best-fitting line, we cannot change any of the coefficients w and hope to make a better fit. This means that the error vector $(b - Aw)$ has to be orthogonal with the vector associated with each variable x_i, or else there would be a way to change the coefficient to fit it better.

Orthogonal vectors have dot products of zero. Since the ith column of A^T has a zero dot product with the error vector, $(A^T)(b - Aw) = \bar{0}$, where $\bar{0}$ is a vector of all zeros. Straightforward algebra then yields

$$w = (A^T A)^{-1} A^T b.$$

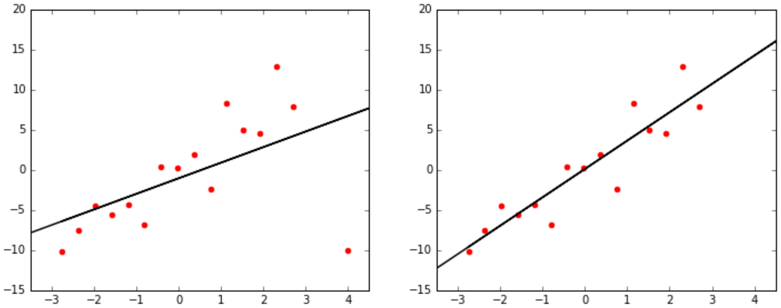

Figure 9.4: Removing outlier points (left) can result in much more meaningful fits (right).

9.2 Better Regression Models

Given a matrix A of n points, each of $m-1$ dimensions, and an $n \times 1$ target array b, we can invert and multiply the appropriate matrices to get the desired coefficient matrix w. This defines a regression model. Done!

However, there are several steps one can take which can lead to better regression models. Some of these involve manipulating the input data to increase the likelihood of an accurate model, but others require more conceptual issues of what our model *should* look like.

9.2.1 Removing Outliers

Linear regression seeks the line $y = f(x)$ which minimizes the sum of the squared errors over all training points, i.e. the coefficient vector w that minimizes

$$\sum_{i=1}^{n} (y_i - f(x_i))^2, \text{ where } f(x) = w_0 + \sum_{i=1}^{m-1} w_i x_i$$

Because of the quadratic weight of the residuals, outlying points can greatly affect the fit. A point at a distance 10 from its prediction has 100 times the impact on training error than a point only 1 unit from the fitted line. One might argue that this is appropriate, but it should be clear that outlier points have a big impact in the shape of the best-fitting line. This creates a problem when the outlier points reflect noise rather than signal, because the regression line goes out of its way to accommodate the bad data instead of fitting the good.

We first encountered this problem back in Figure 6.3, with the Anscombe quartet, a collection of four small data sets with identical summary statistics and regression lines. Two of these point sets achieved their magic because of solitary outlier points. Remove the outliers, and the fit now goes through the heart of the data.

Figure 9.4 shows the best-fitting regression line with (left) and without (right) an outlier point in the lower right. The fit on the right is much better: with an r^2 of 0.917 without the outlier, compared to 0.548 with the outlier.

Therefore identifying outlying points and removing them in a principled way can yield a more robust fit. The simplest approach is to fit the entire set of points, and then use the magnitude of the residual $r_i = (y_i - f(x_i))^2$ to decide whether point p_i is an outlier. It is important to convince yourself that these points *really* represent errors before deleting them, however. Otherwise you will be left with an impressively linear fit that works well only on the examples you didn't delete.

9.2.2 Fitting Non-Linear Functions

Linear relationships are easier to understand than non-linear ones, and grossly appropriate as a default assumption in the absence of better data. Many phenomena *are* linear in nature, with the dependent variable growing roughly proportionally with the input variables:

- Income grows roughly linearly with the amount of time worked.
- The price of a home grows roughly linearly with the size of the living area it contains.
- People's weight increases roughly linearly with the amount of food eaten.

Linear regression does great when it tries to fit data that in fact has an underlying linear relationship. But, generally speaking, no interesting function is *perfectly* linear. Indeed, there is an old statistician's rule that states if you want a function to be linear, measure it at only two points.

We could greatly increase the repertoire of shapes we can model if we move beyond linear functions. Linear regression fits lines, not high-order curves. But we can fit quadratics by adding an extra variable with the value x^2 to our data matrix, in addition to x. The model $y = w_0 + w_1 x + w_2 x^2$ is quadratic, but note that it is a linear function of its non-linear input values. We can fit arbitrarily-complex functions by adding the right higher-order variables to our data matrix, and forming linear combinations of them. We can fit arbitrary polynomials and exponentials/logarithms by explicitly including the right component variables in our data matrix, such as $\sqrt{x}$, $\lg(x)$, x^3, and $1/x$.

Extra features can also be used to capture non-linear interactions between pairs of input variables. The area of a rectangle A is computed $length \times width$, meaning one cannot get an accurate approximation of A as a linear combination of length and width. But, once we add an area feature to our data matrix, this non-linear interaction can be captured with a linear model.

However, explicit inclusion of all possible non-linear terms quickly becomes intractable. Adding all powers x^i for $1 \leq i \leq k$ will blow up the data matrix by a factor of k. Including all product pairs among n variables is even worse, making the matrix $n(n+1)/2$ times larger. One must be judicious about which non-linear terms to consider for a role in the model. Indeed, one of the advantages

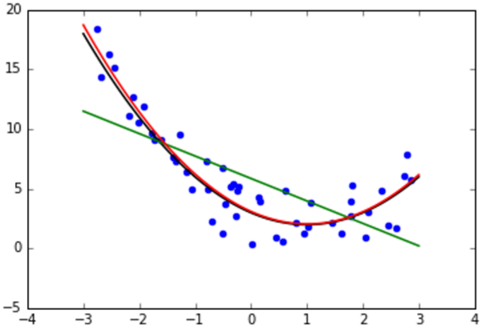

Figure 9.5: Higher-order models (red) can lead to better fits than linear models (green).

of more powerful learning methods, like support vector machines, will be that they can incorporate non-linear terms without explicit enumeration.

9.2.3 Feature and Target Scaling

In principle, linear regression can find the best linear model fitting any data set. But we should do whatever we can to help it find the right model. This generally involves preprocessing the data to optimize for expressibility, interpretability, and numerical stability. The issue here is that features which vary over wide numerical ranges require coefficients over similarly wide ranges to bring them together.

Suppose we wanted to build a model to predict the gross national product of countries in dollars, as a function of their population size x_1 and literacy rate x_2. Both factors seem like reasonable components of such a model. Indeed, both factors may well contribute equally to the amount of economic activity. But they operate on entirely different scales: national populations vary from tens of thousands to over a billion people, while the fraction of people who can read is, by definition, between zero and one. One might imagine the resulting fitted model as looking somewhat like this:

$$GDP = \$10,000x_1 + \$10,000,000,000,000x_2.$$

This is very bad, for several reasons:

- *Unreadable coefficients*: Quick, what is the coefficient of x_2 in the above equation? It is hard for us to deal with the magnitude of such numbers (it is 10 trillion), and hard for us to tell which variable makes a more important contribution to the result given their ranges. Is it x_1 or x_2?
- *Numerical imprecision*: Numerical optimization algorithms have trouble when values range over many orders of magnitude. It isn't just the fact

that floating point numbers are represented by a finite number of bits. More important is that many machine learning algorithms get parameterized by constants that must hold simultaneously for all variables. For example, using fixed step sizes in gradient descent search (to be discussed in Section 9.4.2) might cause it to wildly overshoot in certain directions while undershooting in others.

- *Inappropriate formulations*: The model given above to predict GDP is silly on the face of it. Suppose I decide to form my own country, which would have exactly one person in it, who could read. Do we really think that Skienaland should have a GDP of \$10,000,000,010,000?

  A better model might be something like:

  $$GDP = \$20,000 x_1 x_2$$

  which can be interpreted as each of the x_1 people creating wealth at a rate modulated by their literacy. This generally requires seeding the data matrix with the appropriate product terms. But there is a chance that with proper (logarithmic) target scaling, this model might fall directly out from linear regression.

We will now consider three different forms of scaling, which address these different types of problems.

Feature Scaling: Z-Scores

We have previously discussed Z-scores, which scale the values of each feature individually, so that the mean is zero and the ranges are comparable. Let μ be the mean value of the given feature, and σ the standard deviation. Then the Z-score of x is $Z(x) = (x - \mu)/\sigma$.

Using Z-scores in regression addresses the question of interpretability. Since all features will have similar means and variances, the magnitude of the coefficients will determine the relative importance of these factors towards the forecast. Indeed, in proper conditions, these coefficients will reflect the correlation coefficient of each variable with the target. Further, that these variables now range over the same magnitude simplifies the work for the optimization algorithm.

Sublinear Feature Scaling

Consider a linear model for predicting the number of years of education y that a child will receive as a function of household income. Education levels can vary between 0 and $12 + 4 + 5 = 19$ years, since we consider up to the possible completion of a Ph.D. A family's income level x can vary between 0 and Bill Gates. But observe that no model of the form

$$y = w_1 x + w_0$$

can possibly give sensible answers for both my kids and Bill Gates' kids. The real impact of income on education level is presumably at the lower end: children below the poverty line may not, on average, go beyond high school, while upper-middle-class kids generally go to college. But there is no way to capture this in a linearly-weighted variable without dooming the Gates children to hundreds or thousands of years at school.

An enormous gap between the largest/smallest and median values means that no coefficient can use the feature without blowup on big values. Income level is power law distributed, and the Z-scores of such power law variables can't help, because they are just linear transformations. The key is to replace/augment such features x with sublinear functions like $\log(x)$ and $\sqrt{x}$. Z-scores of these transformed variables will prove much more meaningful to build models from.

Sublinear Target Scaling

Small-scale variables need small-scale targets, in order to be realized using small-scale coefficients. Trying to predict GDP from Z-scored variables will require enormously large coefficients. How else could you get to trillions of dollars from a linear combination of variables ranging from -3 to $+3$?

Perhaps scaling the target value from dollars to billions of dollars here would be helpful, but there is a deeper problem. When your features are normally distributed, you can only do a good job regressing to a similarly distributed target. Statistics like GDP are likely power law distributed: there are many small poor countries compared to very few large rich ones. Any linear combination of normally-distributed variables cannot effectively realize a power law-distributed target.

The solution here is that trying to predict the *logarithm* ($\log_c(y)$) of a power law target y is usually better than predicting y itself. Of course, the value $c^{f(x)}$ can then be used to estimate y, but the potential now exists to make meaningful predictions over the full range of values. Hitting a power law function with a logarithm generally produces a better behaved, more normal distribution.

It also enables us to implicitly realize a broader range of functions. Suppose the "right" function to predict gross domestic product was in fact

$$GDP = \$20,000 x_1 x_2.$$

This could never be realized by linear regression without interaction variables. But observe that

$$\log(GDP) = \log(\$20,000 x_1 x_2) = \log(\$20,000) + \log(x_1) + \log(x_2).$$

Thus the logarithms of arbitrary interaction products could be realized, provided the feature matrix contained the logs of the original input variables as well.

9.2.4 Dealing with Highly-Correlated Features

A final pitfall we will discuss is the problem of highly-correlated features. It is great to have features that are highly correlated with the target: these enable us to build highly-predictive models. However, having multiple features which are highly correlated with *each other* can be asking for trouble.

Suppose you have two perfectly-correlated features in your data matrix, say the subject's height in feet (x_1) as well as their height in meters (x_2). Since 1 meter equals 3.28084 feet, these two variables are perfectly correlated. But having both of these variables can't really help our model, because adding a perfectly correlated feature provides no additional information to make predictions. If such duplicate features really had value for us, it would imply that we could build increasingly accurate models simply by making additional copies of columns from any data matrix!

But correlated features are harmful to models, not just neutral. Suppose our dependent variable is a function of height. Note that equally good models can be built dependent only on x_1, or only on x_2, or on any arbitrary linear combination of x_1 and x_2. Which is the right model to report as the answer?

This is confusing, but even worse things can happen. The rows in the covariance matrix will be mutually dependent, so computing $w = (A^T A)^{-1} A^T b$ now requires inverting a singular matrix! Numerical methods for computing the regression are liable to fail.

The solution here is to identify feature pairs which correlate excessively strongly, by computing the appropriate covariance matrix. If they are lurking, you can eliminate either variable with little loss of power. Better is to eliminate these correlations entirely, by combining the features. This is one of the problems solved by dimension reduction, using techniques like singular value decomposition that we discussed in Section 8.5.1.

9.3 War Story: Taxi Deriver

I am proud of many things in my life, but perhaps most so of being a New Yorker. I live in the most exciting city on earth, the true center of the universe. Astronomers, at least the good ones, will tell you that each new year starts when the ball drops in Times Square, and then radiates out from New York at the speed of light to the rest of the world.

New York cab drivers are respected around the world for their savvy and street smarts. It is customary to tip the driver for each ride, but there is no established tradition of how much that should be. In New York restaurants, the "right" amount to tip the waiter is to double the tax, but I am unaware of any such heuristic for taxi tipping. My algorithm is to round up to the nearest dollar and then toss in a couple of bucks depending upon how fast he got me there. But I have always felt unsure. Am I a cheapskate? Or maybe a sucker?

The taxi data set discussed in Section 1.2.4 promised to hold the answer. It contained over 80 million records, with fields for date, time, pickup and drop off locations, distance traveled, fare, and of course tip. Do people pay disproportionately for longer or shorter trips? Late at night or on weekends? Do others reward fast drivers like I do? It should all be there in the data.

My student, Oleksii Starov, rose to the challenge. We added appropriate features to the data set to capture some of these notions. To explicitly capture conditions like late night and weekends, we set up binary indicator variables, where 1 would denote that the trip was late night and 0 at some other time of day. The coefficients of our final regression equation was:

variable	LR coefficient
(intercept)	0.08370835
duration	0.00000035
distance	0.00000004
fare	0.17503086
tolls	0.06267343
surcharge	0.01924337
weekends	−0.02823731
business day	0.06977724
rush hour	0.01281997
late night	0.04967453
# of passengers	−0.00657358

The results here can be explained simply. Only one variable really matters: the fare on the meter. This model tips 17.5% of the total fare, with very minor adjustments for other things. A single parameter model tipping 18.3% of each fare proved almost as accurate as the ten-factor model.

There were very strong correlations between the fare and both distance traveled (0.95) and trip duration (0.88), but both of these factors are part of the formula by which fares are calculated. These correlations with fare are so strong that neither variable can contribute much additional information. To our disappointment, we couldn't really tease out an influence of time of day or anything else, because these correlations were so weak.

A deeper look at the data revealed that every single tip in the database was charged to a credit card, as opposed to being paid by cash. Entering the tip amount into the meter after each cash transaction is tedious and time consuming, particularly when you are hustling to get as many fares as possible on each 12 hour shift. Further, real New York cabbies are savvy and street-smart enough not to want to pay taxes on tips no one else knows about.

I always pay my fares with cash, but the people who pay by credit card are confronted with a menu offering them the choice of what tip to leave. The data clearly showed most of them mindlessly hitting the middle button, instead of modulating their choice to reflect the quality of service.

Using 80 million fare records to fit a simple linear regression on ten variables is obscene overkill. Better use of this data would be to construct hundreds or even thousands of different models, each designed for a particular class of trips. Perhaps we could build a separate model for trips between each pair of city zip codes. Indeed, recall our map of such tipping behavior, presented back in Figure 1.7.

It took several minutes for the solver to find the best fit on such a large data set, but that it finished at all meant some algorithm faster and more robust than matrix inversion had to be involved. These algorithms view regression as a parameter fitting problem, as we will discuss in the next section.

9.4 Regression as Parameter Fitting

The closed form formula for linear regression, $w = (A^T A)^{-1} A^T b$, is concise and elegant. However, it has some issues which make it suboptimal for computation in practice. Matrix inversion is slow for large systems, and prone to numerical instability. Further, the formulation is brittle: the linear algebra magic here is hard to extend to more general optimization problems.

But there is an alternate way to formulate and solve linear regression problems, which proves better in practice. This approach leads to faster algorithms, more robust numerics, and can be readily adapted to other learning algorithms. It models linear regression as a *parameter fitting* problem, and deploys search algorithms to find the best values that it can for these parameters.

For linear regression, we seek the line that best fits the points, over all possible sets of coefficients. Specifically, we seek the line $y = f(x)$ which minimizes the sum of the squared errors over all training points, i.e. the coefficient vector w that minimizes

$$\sum_{i=1}^{n} (y_i - f(x_i))^2, \text{ where } f(x) = w_0 + \sum_{i=1}^{m-1} w_i x_i$$

For concreteness, let us start with the case where we are trying to model y as a linear function of a single variable or feature x, so $y = f(x)$ means $y = w_0 + w_1 x$. To define our regression line, we seek the parameter pair (w_0, w_1) which minimizes error or cost or *loss*, namely the sum of squares deviation between the point values and the line.

Every possible pair of values for (w_0, w_1) will define *some* line, but we really want the values that minimize the error or *loss function* $J(w_0, w_1)$, where

$$\begin{aligned} J(w_0, w_1) &= \frac{1}{2n} \sum_{i=1}^{n} (y_i - f(x_i))^2 \\ &= \frac{1}{2n} \sum_{i=1}^{n} (y_i - (w_0 + w_1 x_i))^2 \end{aligned}$$

The sum of squares errors should be clear, but where does the $1/(2n)$ come from? The $1/n$ turns this into an average error per row, and the $1/2$ is a common convention for technical reasons. But be clear that the $1/(2n)$ in no way effects the results of the optimization. This multiplier will be the same for each (w_0, w_1) pair, and so has no say in which parameters get chosen.

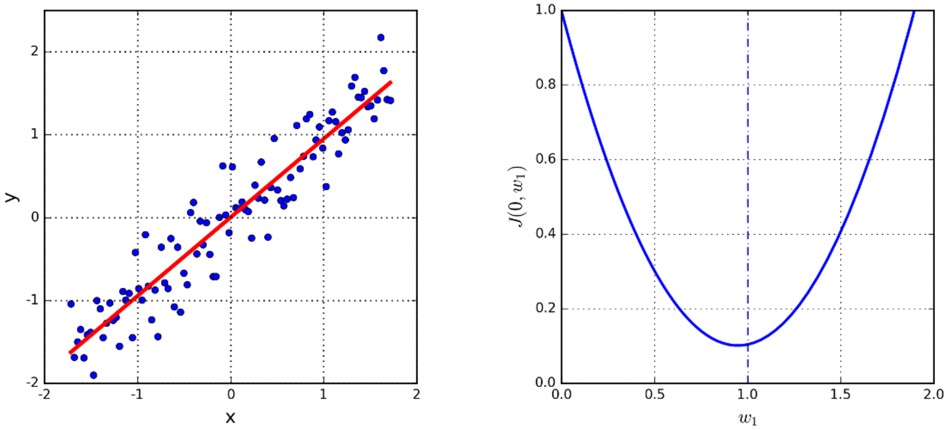

Figure 9.6: The best possible regression line $y = w_1 x$ (left) can be found by identifying the w_1 that minimizes the error of the fit, defined by the minima of a convex function.

So how can we find the right values for w_0 and w_1? We might try a bunch of random value pairs, and keep the one which scores best, i.e. with minimum loss $J(w_0, w_1)$. But it seems very unlikely to stumble on the best or even a decent solution. To search more systematically, we will have to take advantage of a special property lurking within the loss function.

9.4.1 Convex Parameter Spaces

The upshot of the above discussion is that the loss function $J(w_0, w_1)$ defines a surface in (w_0, w_1)-space, with our interest being in the point in this space with smallest z value, where $z = J(w_0, w_1)$.

Let's start by making it even simpler, forcing our regression line to pass through the origin by setting $w_0 = 0$. This leaves us only one free parameter to find, namely the slope of the line w_1. Certain slopes will do a wildly better job of fitting the points shown in Figure 9.6 (left) than others, with the line $y = x$ clearly being the desired fit.

Figure 9.6 (right) shows how the fitting error (loss) varies with w_1. The interesting thing is that the error function is shaped sort of like a parabola. It hits a single minimum value at the bottom of the curve. The x-value of this minimum point defines the best slope w_1 for the regression line, which happens to be $w_1 = 1$.

Any *convex* surface has exactly one local minima. Further, for any convex search space it is quite easy to find this minima: just keep walking in a downward direction until you hit it. From every point on the surface, we can take a small step to a nearby point on the surface. Some directions will take us up to a

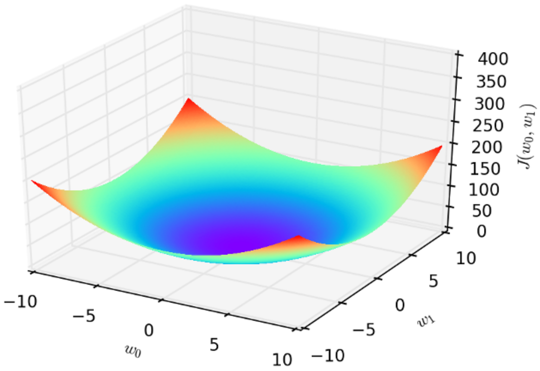

Figure 9.7: Linear regression defines a convex parameter space, where each point represents a possible line, and the minimum point defines the best fitting line.

higher value, but others will take us down. Provided that we can identify which step will take us lower, we will move closer to the minima. And there always is such direction, except when we are standing on the minimal point itself!

Figure 9.7 shows the surface we get for the full regression problem in (w_0, w_1)-space. The loss function $J(w_0, w_1)$ looks like a bowl with a single smallest z-value, which defines the optimal values for the two parameters of the line. The great thing is that this loss function $J(w_0, w_1)$ is again convex, and indeed it remains convex for any linear regression problem in any number of dimensions.

How can we tell whether a given function is convex? Remember back to when you took calculus in one variable, x. You learned how to take the *derivative* $f'(x)$ of a function $f(x)$, which corresponds to the value of the slope of the surface of $f(x)$ at every point. Whenever this derivative was zero, it meant that you had hit some point of interest, be it a local maxima or a minima. Recall the *second derivative* $f''(x)$, which was the derivative function of the derivative $f'(x)$. Depending upon the sign of this second derivative $f''(x)$, you could identify whether you hit a maxima or minima.

Bottom line: the analysis of such derivatives can tell us which functions are and are not convex. We will not delve deeper here. But once it has been established that our loss function is convex, we know that we can trust a procedure like *gradient descent search*, which gets us to the global optima by walking downward.

9.4.2 Gradient Descent Search

We can find the minima of a convex function simply by starting at an arbitrary point, and repeatedly walking in a downward direction. There is only one point

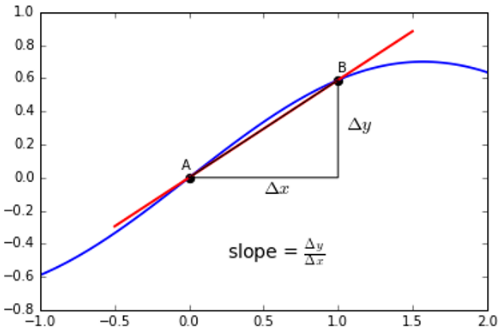

Figure 9.8: The tangent line approximates the derivative at a point.

where there is no way down: the global minima itself. And it is this point that defines the parameters of the best fitting regression line.

But how can we find a direction that leads us down the hill? Again, let's consider the single variable case first, so we seek the slope w_1 of the best-fitting line where $w_0 = 0$. Suppose our current slope candidate is x_0. In this restrictive one-dimensional setting, we can only move to the left or to the right. Try a small step in each direction, i.e. the values $x_0 - \epsilon$ and $x_0 + \epsilon$. If the value of $J(0, x_0 - \epsilon) < J(0, x_0)$, then we should move to the left to go down. If $J(0, x_0 + \epsilon) < J(0, x_0)$, then we should move to the right. If neither is true, it means that we have no place to go to reduce J, so we must have found the minima.

The direction down at $f(x_0)$ is defined by the slope of the *tangent line* at this point. A positive slope means that the minima must lie on the left, while a negative slope puts it on the right. The magnitude of this slope describes the steepness of this drop: by how much will $J(0, x_0 - \epsilon)$ differ from $J(0, x_0)$?

This slope can be approximated by finding the unique line which passes through points $(x_0, J(0, x_0))$ and $(x_0, J(0, x_0 - \epsilon))$, as shown in Figure 9.8. This is exactly what is done in computing the derivative, which at each point specifies the tangent to the curve.

As we move beyond one dimension, we gain the freedom to move in a greater range of directions. Diagonal moves let us cut across multiple dimensions at once. But in principle, we can get the same effect by taking multiple steps, along each distinct dimension in an axis-oriented direction. Think of the Manhattan street grid, where we can get anywhere we want by moving in a combination of north–south and east–west steps. Finding these directions requires computing the *partial derivative* of the objective function along each dimension, namely:

Gradient descent search in two dimensions

```
Repeat until convergence {
        $w_0^{t+1} := w_0^t - \alpha \frac{\partial}{\partial w_0} J(w_0^t, w_1^t)$
        $w_1^{t+1} := w_1^t - \alpha \frac{\partial}{\partial w_1} J(w_0^t, w_1^t)$
}
```

Figure 9.9: Pseudocode for regression by gradient descent search. The variable t denotes the iteration number of the computation.

$$\begin{aligned} \frac{\partial}{\partial w_j} &= \frac{2}{\partial w_j} \frac{1}{2n} \sum_{i=1}^{n} (f(x_i) - b_i)^2 \\ &= \frac{2}{\partial w_j} \frac{1}{2n} \sum_{i=1}^{n} (w_0 + (w_1 x_i) - b_i)^2 \end{aligned}$$

But zig-zagging along dimensions seems slow and clumsy. Like Superman, we want to leap buildings in a single bound. The magnitude of the partial derivatives defines the steepness in each direction, and the resulting vector (say three steps west for every one step north) defines the fastest way down from this point.

9.4.3 What is the Right Learning Rate?

The derivative of the loss function points us in the right direction to walk towards the minima, which specifies the parameters to solve our regression problem. But it doesn't tell us how far to walk. The value of this direction decreases with distance. It is indeed true that the fastest way to drive to Miami from New York is to head south, but at some point you will need more detailed instructions.

Gradient descent search operates in rounds: find the best direction, take a step, and then repeat until we hit the target. The size of our step is called the *learning rate*, and it defines the speed with which we find the minima. Taking tiny baby steps and repeatedly consulting the map (i.e. partial derivatives) will indeed get us there, but only very slowly.

However, bigger isn't always better. If the learning rate is too high, we might jump past the minima, as shown in Figure 9.10 (right). This might mean slow

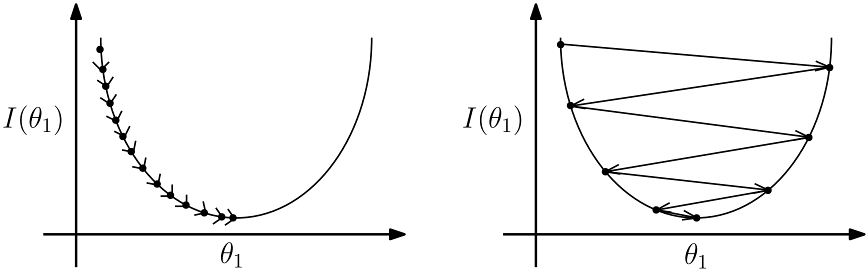

Figure 9.10: The effect of learning rate/step size. Taking too small a step size requires many iterations to converge, while too large a step size causes us to overshoot the minima.

progress towards the hole as we bounce past it on each step, or even negative progress as we end up at a value of $J(w)$ higher than where we were before.

In principle, we want a large learning rate at the beginning of our search, but one which decreases as we get closer to our goal. We need to monitor the value of our loss function over the course of the optimization. If progress becomes too slow, we can increase the step size by a multiplicative factor (say 3) or give up: accepting the current parameter values for our fitting line as good enough. But if the value of $J(w)$ increases, it means that we have overshot our goal. Thus our step size was too large, so we should decrease the learning rate by a multiplicative factor: say by 1/3.

The details of this are messy, heuristic, and ad hoc. But fortunately library functions for gradient descent search have built-in algorithms for adjusting the learning rate. Presumably these algorithms have been highly tuned, and should generally do what they are supposed to do.

But the shape of the surface makes a big difference as to how successfully gradient descent search finds the global minimum. If our bowl-shaped surface was relatively flat, like a plate, the truly lowest point might be obscured by a cloud of noise and numerical error. Even if we do eventually find the minima, it might take us a very long time to get there.

However, even worse things happen when our loss function is not convex, meaning there can be many local minima, as in Figure 9.11. Now this can't be the case for linear regression, but does happen for many other interesting machine learning problems we will encounter.

Local optimization can easily get stuck in local minima for non-convex functions. Suppose we want to reach the top of a ski slope from our lodge in the valley. If we start by walking up to the second floor of the lodge, we will get trapped forever, unless there is some mechanism for taking steps backwards to free us from the local optima. This is the value of search heuristics like simulated annealing, which provides a way out of small local optima to keep us advancing towards the global goal.

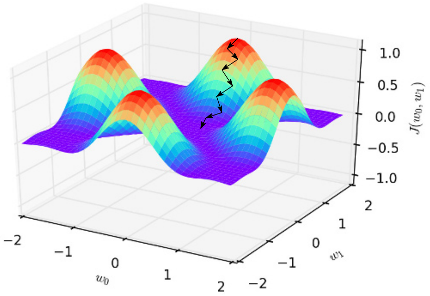

Figure 9.11: Gradient descent search finds local minima for non-convex surfaces, but does not guarantee a globally optimum solution.

> *Take-Home Lesson*: Gradient descent search remains useful in practice for non-convex optimization, although it no longer guarantees an optimal solution. Instead, we should start repeatedly from different initialization points, and use the best local minima we find to define our solution.

9.4.4 Stochastic Gradient Descent

The algebraic definition of our loss function hides something very expensive going on:

$$\frac{\partial}{\partial w_j} = \frac{2}{\partial w_j}\frac{1}{2n}\sum_{i=1}^{n}(f(x_i)-b_i)^2 = \frac{2}{\partial w_j}\frac{1}{2n}\sum_{i=1}^{n}(w_0+(w_1 x_i)-b_i)^2$$

It's that summation. To compute the best direction and rate of change for each dimension j, we must cycle through *all* n of our training points. Evaluating each partial derivative takes time linear in the number of examples, for each step! For linear regression on our lavish taxicab data set, this means 80 million squared-difference computations just to identify the absolute best direction to advance one step towards the goal.

This is madness. Instead, we can try an approximation that uses only a small number of examples to estimate the derivative, and hopes that the resulting direction indeed points down. On average it should, since every point will eventually get to vote on direction.

Stochastic gradient descent is an optimization approach based on sampling a small batch of training points, ideally at random, and using them to estimate the derivative at our current position. The smaller the batch size we use, the faster the evaluation is, although we should be more skeptical that the estimated direction is correct. Optimizing the learning rate and the batch size for gradient descent leads to very fast optimization for convex functions, with the details blessedly concealed by a call to a library function.

It can be expensive to make random choices at every step of the search. Better is to randomize the order of the training examples once, to avoid systematic artifacts in how they are presented, and then build our batches by simply marching down the list. This way we can insure that all n of our training instances eventually do contribute to the search, ideally several times as we repeatedly sweep through all examples over the course of optimization.

9.5 Simplifying Models through Regularization

Linear regression is happy to determine the *best possible* linear fit to any collection of n data points, each specified by $m-1$ independent variables and a given target value. But the "best" fit may not be what we *really* want.

The problem is this. Most of the $m-1$ possible features may be uncorrelated with the target, and thus have no real predictive power. Typically, these will show as variables with small coefficients. However, the regression algorithm will use these values to nudge the line so as to reduce least square error on the given training examples. Using noise (the uncorrelated variables) to fit noise (the residual left from a simple model on the genuinely correlated variables) is asking for trouble.

Representative here is our experience with the taxi tipping model, as detailed in the war story. The full regression model using ten variables had a mean squared error of 1.5448. The single-variable regression model operating only on fare did slightly worse, with an error of 1.5487. But this difference is just noise. The single variable model is obviously better, by Occam's or anybody else's razor.

Other problems arise when using unconstrained regression. We have seen how strongly correlated features introduce ambiguity into the model. If features A and B are perfectly correlated, using both yields the same accuracy as using either one, resulting in more complicated and less interpretable models.

Providing a rich set of features to regression is good, but remember that "the simplest explanation is best." The simplest explanation relies on the smallest number of variables that do a good job of modeling the data. Ideally our regression would select the most important variables and fit them, but the objective function we have discussed only tries to minimize sum of squares error. We need to change our objective function, through the magic of regularization.

9.5.1 Ridge Regression

Regularization is the trick of adding secondary terms to the objective function to favor models that keep coefficients small. Suppose we generalize our loss function with a second set of terms that are a function of the coefficients, not the training data:

$$J(w) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - f(x_i))^2 + \lambda \sum_{j=1}^{m} w_j^2$$

In this formulation, we pay a penalty proportional to the sum of squares of the coefficients used in the model. By squaring the coefficients, we ignore sign and focus on magnitude. The constant λ modulates the relative strength of the regularization constraints. The higher λ is, the harder the optimization will work to reduce coefficient size, at the expense of increased residuals. It eventually becomes more worthwhile to set the coefficient of an uncorrelated variable to zero, rather than use it to overfit the training set.

Penalizing the sum of squared coefficients, as in the loss function above, is called *ridge regression* or *Tikhonov regularization.* Assuming that the dependent variables have all been properly normalized to mean zero, their coefficient magnitude is a measure of their value to the objective function.

How can we optimize the parameters for ridge regression? A natural extension to the least squares formulation does the job. Let Γ be our $n \times n$ "coefficient weight penalty" matrix. For simplicity, let $\Gamma = I$, the identity matrix. The sum-of-squares loss function we seek to minimize then becomes

$$||Aw - b||^2 + ||\lambda \Gamma w||^2$$

The notation $||v||$ denotes the *norm* of v, a distance function on a vector or matrix. The norm of $||\Gamma w||^2$ is exactly the sum of squares of the coefficients when $\Gamma = I$. Seen this way, the closed form to optimize for w is believable as

$$w = (A^T A + \lambda \Gamma^T \Gamma)^{-1} A^T b.$$

Thus the normal form equation can be generalized to deal with regularization. But, alternately, we can compute the partial derivatives of this loss function and use gradient descent search to do the job faster on large matrices. In any case, library functions for ridge regression and its cousin LASSO regression will be readily available to use on your problem.

9.5.2 LASSO Regression

Ridge regression optimizes to select small coefficients. Because of sum-of-squares cost function, it particularly punishes the largest coefficients. This makes it great to avoid models of the form $y = w_0 + w_1 x_1$, where w_0 is a large positive number and w_1 an offsetting large negative number.

Although ridge regression is effective at reducing the magnitude of the coefficients, this criteria does not really push them to zero and totally eliminate the variable from the model. An alternate choice here is to try to minimize the sum of the absolute values of the coefficients, which is just as happy to drive down the smallest coefficients as the big ones.

LASSO regression (for "Least Absolute Shrinkage and Selection Operator") meets this criteria: minimizing the L_1 metric on the coefficients instead of the L_2 metric of ridge regression. With LASSO, we specify an explicit constraint t as to what the sum of the coefficients can be, and the optimization minimizes the sum of squares error under this constraint:

$$J(w,t) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - f(x_i))^2 \text{ subject to } \sum_{j=1}^{m}|w_j| \leq t.$$

Specifying a smaller value of t tightens the LASSO, further constraining the magnitudes of the coefficients w.

As an example of how LASSO zeros out small coefficients, observe what it did to the taxi tipping model for a particular value of t:

variable	LR coefficient	LASSO
(intercept)	0.08370835	0.079601141
duration	0.00000035	0.00000035
distance	0.00000004	0.00000004
fare	0.17503086	0.17804921
tolls	0.06267343	0.00000000
surcharge	0.01924337	0.00000000
weekends	−0.02823731	0.00000000
business day	0.06977724	0.00000000
rush hour	0.01281997	0.00000000
late night	0.04967453	0.00000000
# of passengers	−0.00657358	0.00000000

As you can see, LASSO zeroed out most of the coefficients, resulting in a simpler and more robust model, which fits the data almost as well as the unconstrained linear regression.

But why does LASSO actively drive coefficients to zero? It has to do with the shape of the circle of the L_1 metric. As we will see in Figure 10.2, the shape of the L_1 circle (the collection of points equidistant from the origin) is not round, but has vertices and lower-dimensional features like edges and faces. Constraining our coefficients w to lie on the surface of a radius-t L_1 circle means it is likely to hit one of these lower-dimensional features, meaning the unused dimensions get zero coefficients.

Which works better, LASSO or ridge regression? The answer is that it depends. Both methods should be supported in your favorite optimization library, so try each of them and see what happens.

9.5.3 Trade-Offs between Fit and Complexity

How do we set the right value for our regularization parameter, be it λ or t? Using a small-enough λ or a large-enough t provides little penalty against selecting the coefficients to minimize training error. By contrast, using a very large λ or very small t ensures small coefficients, even at the cost of substantial modeling error. Tuning these parameters enables us to seek the sweet spot between over and under-fitting.

By optimizing these models over a large range of values for the appropriate regularization parameter t, we get a graph of the evaluation error as a function of t. A good fit to the training data with few/small parameters is more robust than a slightly better fit with many parameters.

Managing this trade-off is largely a question of taste. However, several metrics have been developed to help with model selection. Most prominent are the *Akaike Information Criteria* (AIC) and the *Baysian Information Criteria* (BIC). We will not delve deeper than their names, so it is fair for you to think of these metrics as voodoo at this point. But your optimization/evaluation system may well output them for the fitted models they produce, providing a way to compare models with different numbers of parameters.

Even though LASSO/ridge regression punishes coefficients based on magnitude, they do not explicitly set them to zero if you want exactly k parameters. You must be the one to remove useless variables from your model. Automatic feature-selection methods might decide to zero-out small coefficients, but explicitly constructing models from all possible subsets of features is generally computationally infeasible.

The features to be removed first should be those with (a) small coefficients, (b) low correlation with the objective function, (c) high correlation with another feature in the model, and (d) no obvious justifiable relationship with the target. For example, a famous study once showed a strong correlation between the U.S. gross national product and the annual volume of butter production in Bangladesh. The sage modeler can reject this variable as ridiculous, in ways that automated methods cannot.

9.6 Classification and Logistic Regression

We are often faced with the challenge of assigning items the right label according to a predefined set of classes:

- Is the vehicle in the image a car or a truck? Is a given tissue sample indicative of cancer, or is it benign?

- Is a particular piece of email spam, or personalized content of interest to the user?

- Social media analysis seeks to identify properties of people from associated data. Is a given person male or female? Will they tend to vote Democrat or Republican?

Classification is the problem of predicting the right label for a given input record. The task differs from regression in that labels are discrete entities, not continuous function values. Trying to pick the right answer from two possibilities might seem easier than forecasting open-ended quantities, but it is also a lot easier to get dinged for being wrong.

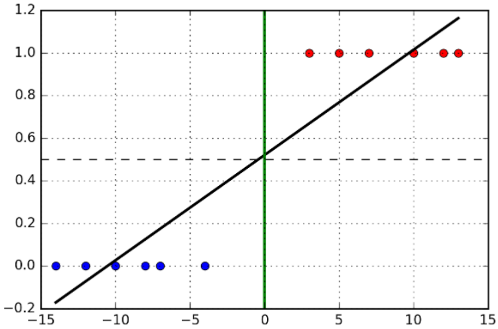

Figure 9.12: The optimal regression line cuts through the classes, even though a perfect separator line ($x = 0$) exists.

In this section, approaches to building classification systems using linear regression will be developed, but this is just the beginning. Classification is a bread-and-butter problem in data science, and we will see several other approaches over the next two chapters.

9.6.1 Regression for Classification

We can apply linear regression to classification problems by converting the class names of training examples to numbers. For now, let's restrict our attention to two class problems, or *binary classification*. We will generalize this to multi-class problems in Section 9.7.2.

Numbering these classes as 0/1 works fine for binary classifiers. By convention, the "positive" class gets 0 and the "negative" one 1:

- male=0 / female=1
- democrat=0 / republican=1
- spam=1 / non-spam=0
- cancer=1 / benign=0

The negative/1 class generally denotes the rarer or more special case. There is no value judgment intended here by positive/negative: indeed, when the classes are of equal size the choice is made arbitrarily.

We might consider training a regression line $f(x)$ for our feature vector x where the target values are these 0/1 labels, as shown in Figure 9.12. There is some logic here. Instances similar to positive training examples should get lower scores than those closer to negative instances. We can threshold the value returned by $f(x)$ to interpret it as a label: $f(x) \leq 0.5$ means that x is positive. When $f(x) > 0.5$ we instead assign the negative label.

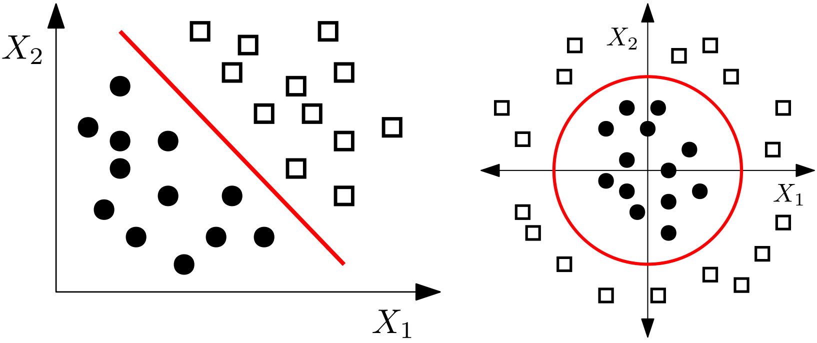

Figure 9.13: A separating line partitions two classes in feature space (left). However, non-linear separators are better at fitting certain training sets (right).

But there are problems with this formulation. Suppose we add a number of "very negative" examples to the training data. The regression line will tilt towards these examples, putting the correct classification of more marginal examples at risk. This is unfortunate, because we would have already properly classified these very negative points, anyway. We really want the line to cut between the classes and serve as a border, instead of through these classes as a scorer.

9.6.2 Decision Boundaries

The right way to think about classification is as carving feature space into regions, so that all the points within any given region are destined to be assigned the same label. Regions are defined by their boundaries, so we want regression to find separating lines instead of a fit.

Figure 9.13 (left) shows how training examples for binary classification can be viewed as colored points in feature space. Our hopes for accurate classification rest on regional coherence among the points. This means that nearby points tend to have similar labels, and that boundaries between regions tend to be sharp instead of fuzzy.

Ideally, our two classes will be well-separated in feature space, so a line can easily partition them. But more generally, there will be outliers. We need to judge our classifier by the "purity" of the resulting separation, penalizing the misclassification of points which lie on the wrong side of the line.

Any set of points can be perfectly partitioned, if we design a complicated-enough boundary that swerves in and out to capture all instances with a given label. See Figure 9.14. Such complicated separators usually reflect overfitting the training set. Linear separators offer the virtue of simplicity and robustness and, as we will see, can be effectively constructed using *logistic regression*.

More generally, we may be interested in non-linear but low-complexity decision boundaries, if they better separate the class boundaries. The ideal sep-

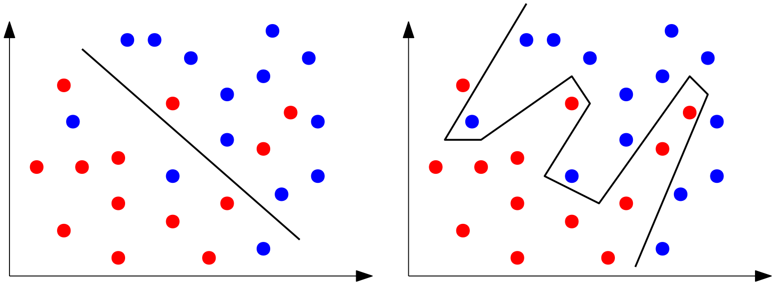

Figure 9.14: Linear classifiers cannot always separate two classes (left). However, perfect separation achieved using complex boundaries usually reflects overfitting more than insight (right).

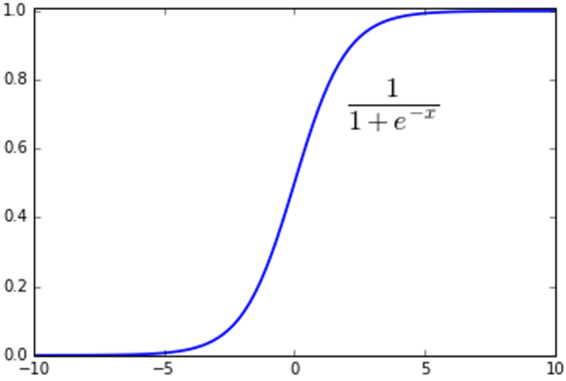

Figure 9.15: The logit function maps a score to a probability.

arating curve in Figure 9.13 (right) is not a line, but a circle. However, it can be found as a linear function of quadratic features like x_1^2 and $x_1 x_2$. We can use logistic regression to find non-linear boundaries if the data matrix is seeded with non-linear features, as discussed in Section 9.2.2.

9.6.3 Logistic Regression

Recall the logit function $f(x)$, which we introduced back in Section 4.4.1:

$$f(x) = \frac{1}{1 + e^{-cx}}$$

This function takes as input a real value $-\infty \leq x \leq \infty$, and produces a value ranging over [0,1], i.e. a probability. Figure 9.15 plots the logit function $f(x)$, which is a sigmoidal curve: flat at both sides but a steep rise in the middle.

The shape of the logit function makes it particularly suited to the interpretation of classification boundaries. In particular, let x be a score that reflects the distance that a particular point p lies above/below or left/right of a line l separating two classes. We want $f(x)$ to measure the probability that p deserves a negative label.

The logit function maps scores into probabilities using only one parameter. The important cases are those at the midpoint and endpoints. Logit says that $f(0) = 1/2$, meaning that the label of a point on the boundary is essentially a coin toss between the two possibilities. This is as it should be. More unambiguous decisions can be made the greater our distance from this boundary, so $f(\infty) = 1$ and $f(-\infty) = 0$.

Our confidence as a function of distance is modulated by the scaling constant c. A value of c near zero makes for a very gradual transition from positive to negative. In contrast, we can turn the logit into a staircase by assigning a large enough value to c, meaning that small distances from the boundary translate into large increases in confidence of classification.

We need three things to use the logit function effectively for classification:

- Extending $f(x)$ beyond a single variable, to a full $(m-1)$-dimensional input vector x.
- The threshold value t setting the midpoint of our score distribution (here zero).
- The value of the scaling constant c regulating the steepness of the transition.

We can achieve all three by fitting a linear function $h(x, w)$ to the data, where

$$h(x, w) = w_0 + \sum_{i=1}^{m-1} w_i \cdot x_i$$

which can then be plugged into the logistic function to yield the classifier:

$$f(x) = \frac{1}{1 + e^{-h(x,w)}}$$

Note that the coefficients of $h(x, w)$ are rich enough to encode the threshold ($t = w_0$) and steepness (c is essentially the average of w_1 through w_{n-1}) parameters.

The only remaining question is how to fit the coefficient vector w to the training data. Recall that we are given a zero/one class label y_i for each input vector x_i, where $1 \leq i \leq n$. We need a penalty function that ascribes appropriate costs to returning $f(x_i)$ as the probability that the class y_i is positive, i.e. $y_i = 1$.

Let us first consider the case where y_i really *is* 1. Ideally $f(x_i) = 1$ in this case, so we want to penalize it for being smaller than 1. Indeed, we want to punish it aggressively when $f(y_i) \to 0$, because that means that the classifier is stating that element i has little chance of being in class-1, when that actually is the case.

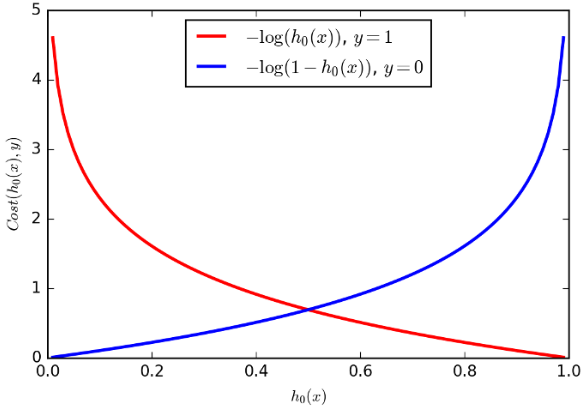

Figure 9.16: Cost penalties for positive (blue) and negative (right) elements. The penalty is zero if the correct label is assigned with probability 1, but increasing as a function of misplaced confidence.

The logarithmic function $cost(x_i, 1) = -\log(f(x_i))$ turns out to be a good penalty function when $y_i = 1$. Recall the definition of the logarithm (or inverse exponential function) from Section 2.4, namely that

$$y = \log_b x \rightarrow b^y = x.$$

As shown in Figure 9.16, $\log(1) = 0$ for any reasonable base, so zero penalty is charged when $f(x_i) = 1$, which is as it should be for correctly identifying $y_i = 1$. Since $b^{\log_b x} = x$, $\log(x) \rightarrow -\infty$ as $x \rightarrow 0$. This makes $cost(x_i, 1) = -\log(f(x_i))$ an increasingly severe penalty the more we misclassify y_i.

Now consider the case where $y_i = 0$. We want to punish the classifier for high values of $f(x_i)$, i.e. more as $f(x_i) \rightarrow 1$. A little reflection should convince you that the right penalty is now $cost(x_i, 0) = -\log(1 - f(x_i))$.

To tie these together, note what happens when we multiply $cost(x_i, 1)$ times y_i. There are only two possible values, namely $y_i = 0$ or $y_i = 1$. This has the desired effect, because the penalty is zeroed out in the case where it does not apply. Similarly, multiplying by $(1 - y_i)$ has the opposite effect: zeroing out the penalty when $y_i = 1$, and applying it when $y_i = 0$. Multiplying the costs by the appropriate indicator variables enables us to define the loss function for logistic regression as an algebraic formula:

$$\begin{aligned} J(w) &= \frac{1}{n} \sum_{i=1}^{n} cost(f(x_i, w), y_i) \\ &= -\frac{1}{n} [\sum_{i=1}^{n} y_i \log f(x_i, w) + (1 - y_i) \log(1 - f(x_i, w))] \end{aligned}$$

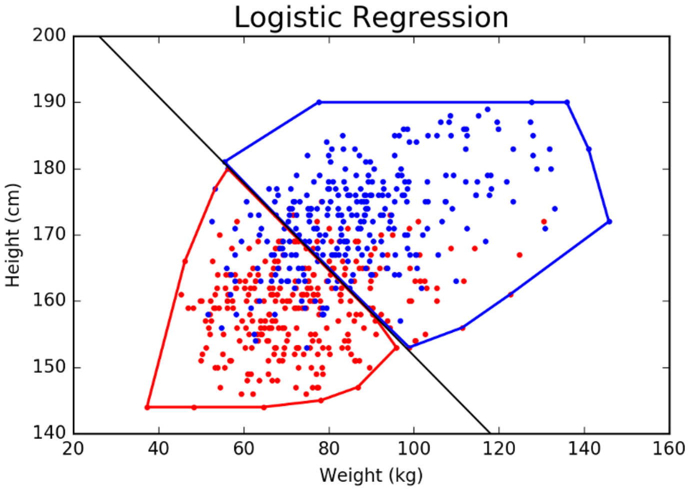

Figure 9.17: The logistic regression classifier best separating men and women in weight–height space. The red region contains 229 women and only 63 men, while the blue region contains 223 men to 65 women.

The wonderful thing about this loss function is that it is convex, meaning that we can find the parameters w which best fit the training examples using gradient descent. Thus we can use logistic regression to find the best linear separator between two classes, providing a natural approach to binary classification.

9.7 Issues in Logistic Classification

There are several nuances to building effective classifiers, issues which are relevant both to logistic regression and the other machine learning methods that we will explore over the next two chapters. These include managing unbalanced class sizes, multi-class classification, and constructing true probability distributions from independent classifiers.

9.7.1 Balanced Training Classes

Consider the following classification problem, which is of great interest to law enforcement agencies in any country. Given the data you have on a particular person p, decide whether p is a terrorist or is no particular threat.

The quality of the data available to you will ultimately determine the accuracy of your classifier, but regardless, there is something about this problem that makes it very hard. It is the fact that there are not enough terrorists available in the general population.

In the United States, we have been blessed with general peace and security. It would not surprise me if there were only 300 or so genuine terrorists in the entire country. In a country of 300 million people, this means that only one out of every million people are active terrorists.

There are two major consequences of this imbalance. The first is that *any* meaningful classifier is doomed to have a lot of false positives. Even if our classifier proved correct an unheard of 99.999% of the time, it would classify 3,000 innocent people as terrorists, ten times the number of bad guys we will catch. Similar issues were discussed in Section 7.4.1, concerning precision and recall.

But the second consequence of this imbalance is that there cannot be many examples of actual terrorists to train on. We might have tens of thousands of innocent people to serve as positive/class-0 examples, but only a few dozen known terrorists to be negative/class-1 training instances.

Consider what the logistic classifier is going to do in such an instance. Even misclassifying *all* of the terrorists as clean cannot contribute too much to the loss function, compared with the cost of how we treat the bigger class. It is more likely to draw a separating line to clear everybody than go hunting for terrorists. The moral here is that it is generally best to use equal numbers of positive and negative examples.

But one class may be hard to find examples for. So what are our options to produce a better classifier?

- *Force balanced classes by discarding members of the bigger class*: This is the simplest way to realize balanced training classes. It is perfectly justified if you have enough rare-class elements to build a respectable classifier. By discarding the excess instances we don't need, we create a harder problem that does not favor the majority class.

- *Replicate elements of the smaller class, ideally with perturbation*: A simple way to get more training examples is to clone the terrorists, inserting perfect replicas of them into the training set under different names. These repeated examples *do* look like terrorists, after all, and adding enough of them will make the classes balanced.

  This formulation is brittle, however. These identical data records might create numerical instabilities, and certainly have a tendency towards overfitting, since moving one extra real terrorist to the right side of the boundary moves all her clones as well. It might be better to add a certain amount of random noise to each cloned example, consistent with variance in the general population. This makes the classifier work harder to find them, and thus minimizes overfitting.

- *Weigh the rare training examples more heavily than instances of the bigger class*: The loss function for parameter optimization contains a separate term for the error of each training instance. Adding a coefficient to ascribe more weight to the most important instances leaves a convex optimization problem, so it can still be optimized by stochastic gradient descent.

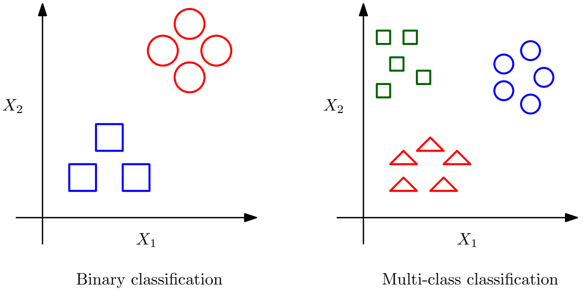

Figure 9.18: Multi-class classification problems are a generalization of binary classification.

The problem with all three of these solutions is that we bias the classifier, by changing the underlying probability distribution. It is important for a classifier to know that terrorists are extremely rare in the general population, perhaps by specifying a Baysian prior distribution.

Of course the best solution would be to round up more training examples from the rarer class, but that isn't always possible. These three techniques are about the best we can muster as an alternative.

9.7.2 Multi-Class Classification

Often classification tasks involve picking from more than two distinct labels. Consider the problem of identifying the genre of a given movie. Logical possibilities include *drama*, *comedy*, *animation*, *action*, *documentary*, and *musical*.

A natural but misguided approach to represent k-distinct classes would add class numbers beyond 0/1. In a hair-color classification problem, perhaps we could assign *blond* = 0, *brown* = 1, *red* = 2, *black* = 4, and so on until we exhaust human variation. Then we could perform a linear regression to predict class number.

But this is generally a bad idea. *Ordinal* scales are defined by either increasing or decreasing values. Unless the ordering of your classes reflects an ordinal scale, the class numbering will be a meaningless target to regress against.

Consider the hair-color numbering above. Should *red* hair lie between *brown* and *black* (as currently defined), or between *blond* and *brown*? Is *grey* hair a lighter shade of *blond*, say class −1, or is it an incomparable condition due principally to aging? Presumably the features that contribute to grey hair (age and the number of teen-age children) are completely orthogonal to those of blond hair (hair salon exposure and Northern European ancestry). If so, there is no way a linear regression system fitting hair color as a continuous variable would be destined to separate these colors out from darker hair.

Certain sets of classes are properly defined by ordinal scales. For example, consider classes formed when people grade themselves on survey questions like "Skiena's class is too much work" or "How many stars do you give this movie?"

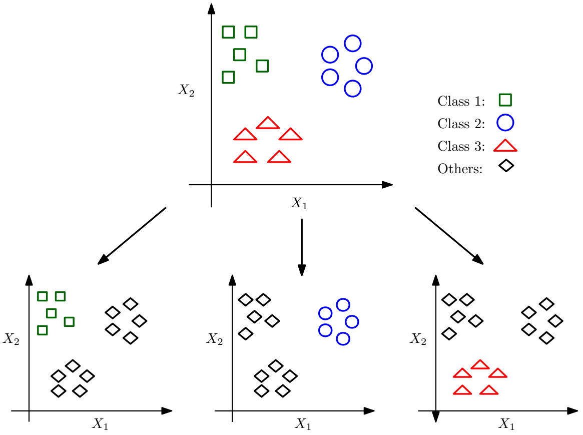

Figure 9.19: Voting among multiple one-vs.-rest classifiers is generally the best way to do multi-class classification.

Completely Agree ↔ Mostly Agree ↔ Neutral ↔ Mostly Disagree ↔ Completely Disagree

Four stars ↔ Three stars ↔ Two stars ↔ One stars ↔ Zero stars

Classes defined by such *Likert scales* are ordinal, and hence such class numbers are a perfectly reasonable thing to regress against. In particular, mistakenly assigning an element to an adjacent class is much less of a problem than assigning it to the wrong end of the scale.

But generally speaking, class labels are not ordinal. A better idea for multi-class discrimination involves building many one-vs.-all classifiers, as shown in Figure 9.19. For each of the possible classes C_i, where $1 \leq i \leq c$, we train a logistic classifier to distinguish elements of C_i against the union of elements from all other classes combined. To identify the label associated with a new element x, we test it against all c of these classifiers, and return the label i which has the highest associated probability.

This approach should seem straightforward and reasonable, but note that the classification problem gets harder the more classes you have. Consider the monkey. By flipping a coin, a monkey should be able to correctly label 50% of the examples in any binary classification problem. But now assume there are a hundred classes. The monkey will only guess correctly 1% of the time. The task is now very hard, and even an excellent classifier will have a difficult time producing good results on it.

9.7.3 Hierarchical Classification

When your problem contains a large number of classes, it pays to group them into a tree or hierarchy so as to improve both accuracy and efficiency. Suppose we built a binary tree, where each individual category is represented by a leaf node. Each internal node represents a classifier for distinguishing between the left descendants and the right descendants.

To use this hierarchy to classify a new item x, we start at the root. Running the root classifier on x will specify it as belonging to either the left or right subtree. Moving down one level, we compare x with the new node's classifier and keep recurring until we hit a leaf, which defines the label assigned to x. The time it takes is proportional to the height of the tree, ideally logarithmic in the number of classes c, instead of being linear in c if we explicitly compare against every class. Classifiers based on this approach are called *decision trees*, and will be discussed further in Section 11.2.

Ideally this hierarchy can be built from domain knowledge, ensuring that categories representing similar classes are grouped together. This has two benefits. First, it makes it more likely that misclassifications will still produce labels from similar classes. Second, it means that intermediate nodes can define higher-order concepts, which can be more accurately recognized. Suppose that the one hundred categories in a image classification problem included "car," "truck," "boat," and "bicycle". When all of these categories are descendants of an intermediate node called "vehicle," we can interpret the path to this node as a lower-resolution, higher-accuracy classifier.

There is another, independent danger with classification that becomes more acute as the number of classes grows. Members of certain classes (think "college students") are much more plentiful than others, like "rock stars." The relative disparity between the size of the largest and smallest classes typically grows along with number of classes.

For this example, let's agree that "rock stars" tend to be sullen, grungy-looking males, providing useful features for any classifier. However, only a small fraction of sullen, grungy-looking males are rock stars, because there are extremely few people who have succeeded in this demanding profession. Classification systems which do not have a proper sense of the prior distribution on labels are doomed having many false positives, by assigning rare labels much too frequently.

This is the heart of *Baysian analysis*: updating our current (prior) understanding of the probability distribution in the face of new evidence. Here, the evidence is the result is from a classifier. If we incorporate a sound prior distribution into our reasoning, we can ensure that items require particularly strong evidence to be assigned to rare classes.

9.7.4 Partition Functions and Multinomial Regression

Recall that our preferred means of multi-class classification involved training independent single-class vs. all logistic classifiers $F_i(x)$, where $1 \leq i \leq c$ and c is the number of distinct labels. One minor issue remains. The probabilities we get from logistic regression aren't really probabilities. Turning them into real probabilities requires the idea of a *partition function*.

For any particular item x, summing up the "probabilities" over all possible labels for x *should* yield $T = 1$, where

$$T = \sum_{i=1}^{c} F_i(x).$$

But should doesn't mean is. All of these classifiers were trained independently, and hence there is nothing forcing them to sum to $T = 1$.

A solution is to divide all of these probabilities by the appropriate constant, namely $F'(x) = F(x)/T$. This may sound like a kludge, because it is. But this is essentially what physicists do when they talk about *partition functions*, which serve as denominators turning something proportional to probabilities into real probabilities.

Multinomial regression is a more principled method of training independent single-class vs. all classifiers, so that the probabilities work out right. This involves using the correct partition function for log odds ratios, which are computed with exponentials of the resulting values. More than this I will not say, but it is reasonable to look for a multinomial regression function in your favorite machine learning library and see how it does when faced with a multi-class regression problem.

A related notion to the partition function arises in Baysian analysis. We are often faced with a challenge of identifying the most likely item label, say A, as a function of evidence E. Recall that Bayes' theorem states that

$$P(A|E) = \frac{P(E|A)P(A)}{P(E)}$$

Computing this as a real probability requires knowing the denominator $P(E)$, which can be a murky thing to compute. But comparing $P(A|E)$ to $P(B|E)$ in order to determine whether label A is more likely than label B does not require knowing $P(E)$, since it is the same in both expressions. Like a physicist, we can waive it away, mumbling about the "partition function."

9.8 Chapter Notes

Linear and logistic regression are standard topics in statistics and optimization. Textbooks on linear/logistic regression and its applications include [JWHT13, Wei05].

The treatment of the gradient descent approach to solving regression here was inspired by Andrew Ng, as presented in his Coursera machine learning course. I strongly recommend his video lectures to those interested in a more thorough treatment of the subject.

The discovery that butter production in Bangladesh accurately forecasted the S&P 500 stock index is due to Leinweber [Lei07]. Unfortunately, like most spurious correlations it broke down immediately after its discovery, and no longer has predictive power.

9.9 Exercises

Linear Regression

9-1. *[3]* Construct an example on $n \geq 6$ points where the optimal regression line is $y = x$, even though none of the input points lie directly on this line.

9-2. *[3]* Suppose we fit a regression line to predict the shelf life of an apple based on its weight. For a particular apple, we predict the shelf life to be 4.6 days. The apples residual is -0.6 days. Did we over or under estimate the shelf-life of the apple? Explain your reasoning.

9-3. *[3]* Suppose we want to find the best-fitting function $y = f(x)$ where $y = w^2 x + wx$. How can we use linear regression to find the best value of w?

9-4. *[3]* Suppose we have the opportunity to pick between using the best fitting model of the form $y = f(x)$ where $y = w^2 x$ or $y = wx$, for constant coefficient w. Which of these is more general, or are they identical?

9-5. *[5]* Explain what a long-tailed distribution is, and provide three examples of relevant phenomena that have long tails. Why are they important in classification and regression problems?

9-6. *[5]* Using a linear algebra library/package, implement the closed form regression solver $w = (A^T A)^{-1} A^T b$. How well does it perform, relative to an existing solver?

9-7. *[3]* Establish the effect that different values for the constant c of the logit function have on the probability of classification being 0.01, 1, 2, and 10 units from the boundary.

Experiments with Linear Regression

9-8. *[5]* Experiment with the effects of fitting non-linear functions with linear regression. For a given (x, y) data set, construct the best fitting line where the set of variables are $\{1, x, \ldots, x^k\}$, for a range of different k. Does the model get better or worse over the course of this process, both in terms of fitting error and general robustness?

9-9. *[5]* Experiment with the effects of feature scaling in linear regression. For a given data set with at least two features (dimensions), multiply all the values of one feature by 10^k, for $-10 \leq k \leq 10$. Does this operation cause a loss of numerical accuracy in fitting?

9-10. *[5]* Experiment with the effects of highly correlated features in linear regression. For a given (x, y) data set, replicate the value of x with small but increasing amounts of random noise. What is returned when the new column is perfectly correlated with the original? What happens with increasing amounts of random noise?

9-11. *[5]* Experiment with the effects of outliers on linear regression. For a given (x, y) data set, construct the best fitting line. Repeatedly delete the point with the largest residual, and refit. Is the sequence of predicted slopes relatively stable for much of this process?

9-12. *[5]* Experiment with the effects of regularization on linear/logistic regression. For a given multi-dimensional data set, construct the best fitting line with (a) no regularization, (b) ridge regression, and (c) LASSO regression; the latter two with a range of constraint values. How does the accuracy of the model change as we reduce the size and number of parameters?

Implementation Projects

9-13. *[5]* Use linear/logistic regression to build a model for one of the following *The Quant Shop* challenges:

(a) *Miss Universe.*

(b) *Movie gross.*

(c) *Baby weight.*

(d) *Art auction price.*

(e) *White Christmas.*

(f) *Football champions.*

(g) *Ghoul pool.*

(h) *Gold/oil prices.*

9-14. *[5]* This story about predicting the results of the NCAA college basketball tournament is instructive:

`http://www.nytimes.com/2015/03/22/opinion/sunday/making-march-madness-eas`
`html`.

Implement such a logistic regression classifier, and extend it to other sports like football.

Interview Questions

9-15. *[8]* Suppose we are training a model using stochastic gradient descent. How do we know if we are converging to a solution?

9-16. *[5]* Do gradient descent methods always converge to the same point?

9-17. *[5]* What assumptions are required for linear regression? What if some of these assumptions are violated?

9-18. *[5]* How do we train a logistic regression model? How do we interpret its coefficients?

Kaggle Challenges

9-19. Identify what is being cooked, given the list of ingredients.

`https://www.kaggle.com/c/whats-cooking`

9-20. Which customers are satisfied with their bank?

`https://www.kaggle.com/c/santander-customer-satisfaction`

9-21. What does a worker need access to in order to do their job?

`https://www.kaggle.com/c/amazon-employee-access-challenge`