CHAPTER

4 Linear Regression

In this chapter, we focus on the regression task, specifically applying one of the simplest possible regression models: the linear model. We use this class of functions to explore a number of fundamental tools that will be useful in the sequel, including matrix-based representations of the data, gradient-based optimization of our model, and further exploration of the notions of model complexity.

Suppose that we have observed a feature vector $x = [x_1, \ldots, x_n]$, and our prediction takes the form of a linear function:

$$f(x\ ;\ \theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots$$

It is helpful for us to define our variables, features, etc. in the form of matrices, which will allow us to write some equations compactly and also translate well into code. To do so, let us define a "zero-th" feature $x_0^{(i)} = 1$ for all data i; then, we can express

$$f(x\ ;\ \theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots = \theta \cdot x^T$$

where $\theta = [\theta_0, \ldots, \theta_n]$ and $x = [x_0, \ldots, x_N]$ are vector representations of the model parameters and features (with $x_0 = 1$), and "$\cdot$" is the usual vector-vector dot product.

Recall that our data for training, $D = \{(x^{(i)}, y^{(i)}\}$ consist of pairs of observed feature vectors $x^{(i)}$ and target values $y^{(i)}$, for $i = 1 \ldots m$. Let us stack these up vertically, giving a data matrix $\mathbf{X}$ and target vector $\mathbf{y}$:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \qquad \mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \ldots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \ldots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \ldots & x_n^{(m)} \end{bmatrix}$$

Where possible, we will try to be consistent that the horizontal axis of vectors and matrices corresponds to the feature index, and the vertical axis corresponds to the data index.[1] Then, the (i, j)th entry of $\mathbf{X}$ is the jth feature of the ith data point: $\mathbf{X}_{ij} = x_j^{(i)}$. Again, here we have taken "feature zero" to be the constant $x_0^{(i)} = 1$.

[1]Note that the orientation of features along rows, and data along columns, is completely arbitrary, and many texts or code may use the opposite convention, leading to transposition differences between equations.

DRAFT

4.1 Optimization

The process of learning involves selecting the predictor out of our hypothesis class, i.e., here the set of linear models, that best fits our observed data D. To this end, we first select a loss function J, and then find a predictor (corresponding to a value of the parameter vector θ) that minimizes this loss function.

Visualizing the loss function

For a given observation (x,y), we can compute the error in our prediction (also called the residual) as $y - f(x;\theta)$:

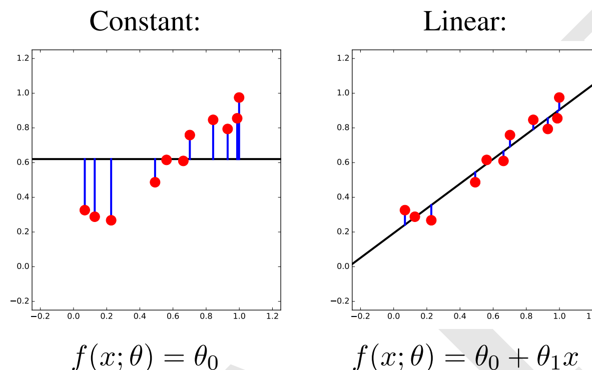


Figure 4.1: Error residuals (blue lines) on a particular data set (red) for (a) a constant predictor and (b) a linear predictor.

In general, we would like to minimize the overall size of these errors. A common, and as we shall see computationally convenient choice is the Euclidean norm of the vector of residuals, or the sum of squared errors (SSE) cost: The cost function J(.) tells us the accuracy of a given parameter vector at predicting our training data. This is a function defined over the space of parameters ; for a two-dimensional parameter vector, we can visualize it as a surface, or use colors or contours to suggest the three-dimensional height (cost) on a 2D plot.

visualalization: a predictor is a point in "parameter space" (a specific value of the parameter vector), and we are visualizing the loss as a function of that parameter vector, $J(\theta)$. The predictor with the least average squared error, then, is the minimum of this function.

Gradient descent

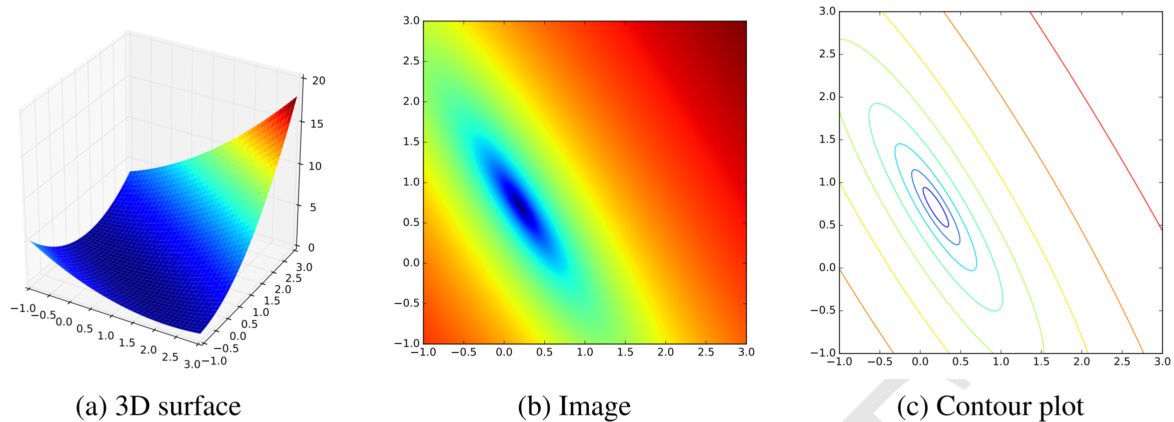
One option is to follow the local slope of $J(\theta)$ downward towards a local minimum. We can evaluate the gradient direction, i.e., the direction in which our cost function $J(\theta)$ has the greatest increase; going in the opposite direction gives us the direction of greatest decrease of our cost J. This gradient will be a vector of the same dimension as θ:

$$\nabla J(\theta) = \left[\frac{\partial J}{\partial \theta_0} \; \frac{\partial J}{\partial \theta_1} \; \cdots \; \frac{\partial J}{\partial \theta_n} \right]$$

In gradient descent, we simply initialize to some starting value of θ and repeatedly update by choosing a new θ by moving in the direction of steepest descent, e.g.,

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$

DRAFT



(a) 3D surface (b) Image (c) Contour plot

Figure 4.2: Three visualizations of the loss function $J(\theta)$, for the linear model in Figure 4.1(b), with $\theta = [\theta_0, \theta_1]$. Since J is a function of two parameters, we can visualize it as (a) a 3D surface; typically we will simply project such surfaces onto the feature space and display their value with (b) color, or simply show (c) the contours (topography) of the resulting surface.

Here, "$leftarrow$" indicates that we are updating the value of θ using the quantity on the right. The value α is a step-size parameter that tells us how far to move in the direction of the gradient. The choice of step size can be very important in gradient descent methods, as it often controls how fast we converge to a local minimum. Values that are too small move very slowly; values that are too large can skip over or even oscillate around local minima.[2]

A common heuristic approach to setting the step size is to let α decrease with each iteration (step), for example choosing α to be inversely proportional to the iteration number: $\alpha = \frac{C}{C+s}$ at iteration s, where C is some constant (e.g., $C = 2$). **cite; convergence properties; rate?**

We also need a criterion for stopping our updates. Typical stopping criteria measure the amount of change applied to θ at the current step, $\alpha \|\nabla J(\theta)\|$ (the vector length of the step), or alternatively the change in the value of the objective $J(\theta)$, after each step. If these values are small, our gradient steps are not significantly updating the solution θ, or not changing its value, and we may stop. Alternatively, stopping criteria based on number of iterations or execution time are also common.

We can see the behavior of gradient descent on the cost function J defined over parameter space (top; each point corresponds to a vector $\theta = [\theta_0 \ \theta_1]$), and the induced predictors $\hat{y}(x) = \theta x^T$ (bottom; each value of (θ) corresponds to a different linear predictor for y):

Stochastic gradient descent

TBD; decomposable loss $J(\theta) = \frac{1}{m}\sum_i J_i(\theta)$; update model θ using the gradient of J_i, i.e., with respect to a single data point i at a time.

Closed form optimum for squared error

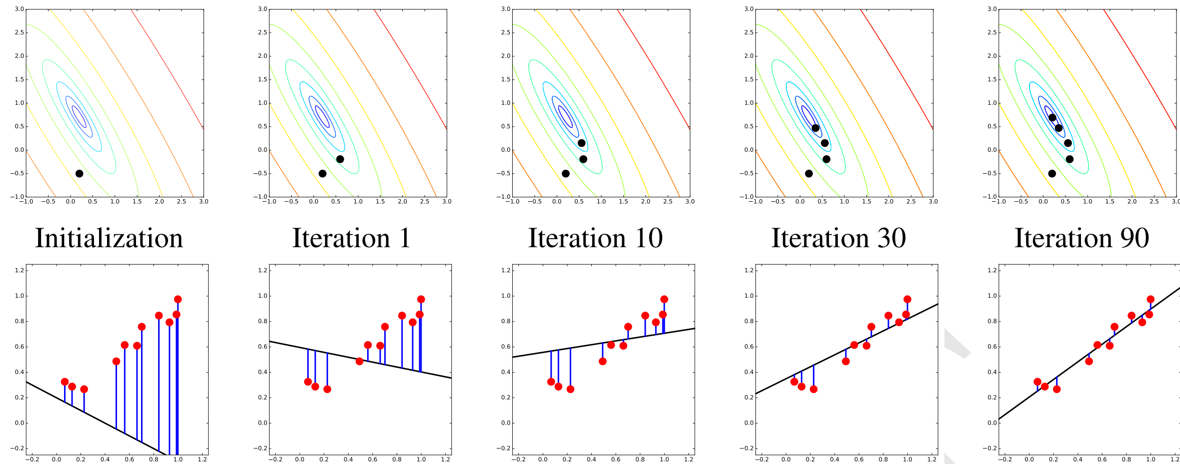
A convenient property of the linear regression function and mean squared error loss is that its optimum can actually be computed in closed form. A simple example helps illustrate the idea.

Suppose we have a simple regression problem with a single, scalar feature x. Given two training points, $D = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)})\}$, with $x^{(1)} \neq x^{(2)}$, there is a unique line that passes through them,

[2]Parameter such as the step size α that influence the amount by which the model is updated at each step are sometimes called a *learning rate*.

DRAFT

"Parameter space": loss $J(\theta)$ and values of θ during optimization



"Feature space": Data $x^{(i)}, y^{(i)}$ and current predictor $f(x;\theta)$

Figure 4.3: Optimization of the parameters θ via gradient descent. The top row shows the space of possible parameter settings (values of θ), displaying the sequence of models obtained during optimization of the loss function $J(\theta)$, along with the contours of J. The bottom row shows the current iteration's model (value of θ) in terms of its prediction $f(x\,;\,\theta)$ and current fit to the training data.

and it is trivial to solve for the parameters of this line, by solving the equations:

$$y^{(1)} = \theta_0 + \theta_1\, x^{(1)}$$
$$y^{(2)} = \theta_0 + \theta_1\, x^{(2)}$$

or, equivalently, $\mathbf{y} = \mathbf{X} \cdot \theta^T$. The condition $x^{(1)} \neq x^{(2)}$ ensures that $\mathbf{X}$ is full rank, and thus invertible, so that we can solve $\theta = \mathbf{y}^T \cdot (\mathbf{X}^T)^{-1}$.

Given more than two data points, of course, there may be no line which passes through the examples; such a system is called *over-determined*. The preceding idea generalizes by solving for the optimum of the squared error loss, i.e, the point at which the gradient is zero. Plugging in our formula for the gradient, we have:

$$\nabla J(\theta) = (\mathbf{y}^T - \theta \cdot \mathbf{X}^T) \cdot \mathbf{X} = 0$$

Distributing $\mathbf{X}$ inside and rearranging gives a quadratic equation, the solution of which is the minimum squared error (MSE) estimator

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

The term $(\mathbf{X} \cdot (\mathbf{X}^T \cdot \mathbf{X})^{-1})$ is called the (Moore-Penrose) pseudo-inverse. For non-square $\mathbf{X}$, for example $m > n$ (an overconstrained set of equations), the pseudo-inverse can be thought of as a generalization of the standard matrix inverse $(\mathbf{X}^T)^{-1}$; if $m = n$ and $\mathbf{X}$ is full rank, the two will be equivalent.

4.2 Increasing the number of features

So far we have considered linear functions of several observed features (x_1), (x_2). Suppose that we have only one feature, $(x = x_1)$, but would like our predictor to be a "nonlinear" function of x, for example $f(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$. We can simply define "new" features $x_2 = x^2$, $x_3 = x^3$, and so on (just as we defined $x_0 = 1 = x^0$), and this predictor becomes simply $f(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$. In other

DRAFT

words, it still fits the linear regression model, but in a new ”feature space” with additional features that are deterministic functions of our observations. Applying our least-squares estimator gives polynomial fits,

$$f(x) = \sum_{i=1}^{p} \theta_i x^i.$$

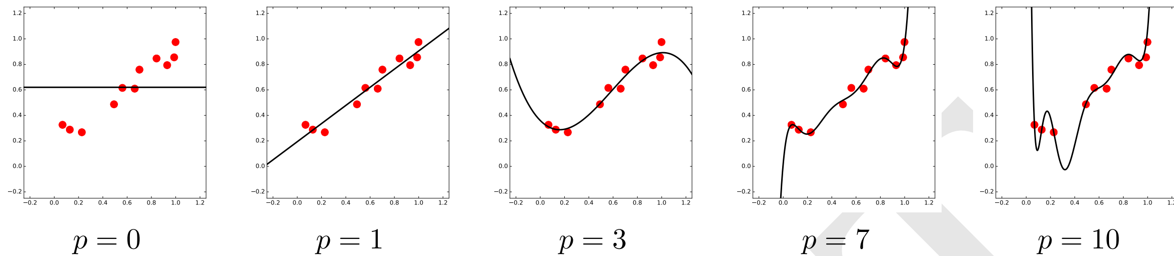


Figure 4.4: Fit to the training data as a function of the number of polynomial features p, from $p = 0$ (a constant predictor, $f(x;\theta) = \theta_0$) to $p = 10$ (i.e., $f(x;\theta) = \theta_0 + \theta_1 x + \ldots + \theta_1 0x^{10}$).

If we think that our target variable y is likely to be linearly related to some complex function of several observed variables, that combination can also be added as a new, observed feature.

There are two ways to view this process. First, we are creating a “more complex” functional predictor ($\hat{y} = f(x)$); the set of functions we can represent is now larger (e.g., increasing from the set of all lines to the set of all cubic polynomials).

An alternative is to forget that the features are now deterministically related to one another, and to imagine that we are adding extra measurements (features) with which to predict y. In this view, we are learning a linear predictor from data, but those data lie in a higher dimensional space. (We will return to these views later, in classification.)

Feature transform, $\Phi(x)$

4.3 Overfitting and Model Selection

As we saw in our polynomial fits, for 11 data points and a degree-10 polynomial (with 11 coefficients), we can predict all of our training data exactly! And yet, something about that predictor is not satisfying – it does not ”look” like a good predictor. In fact, we have ”overfit” to the data – we have chosen such a complex predictor that, although it is able to reconstruct the training data, we have no faith that it will accurately predict new data.

We can see the ”generalization” or test error of a predictor simply by gathering more data, and evaluating our cost function (e.g. mean squared error) on those new points (green). What we find is that, for very simple predictors (P=0,1), the performance on new test data is much like the performance on the training data. However, as the function gets more complex, the training error continues to decrease – but the test error does not. By P=10, training error is zero, but test error is extremely high.

TBD

We can see this directly by plotting training and test error as a function of polynomial degree P. For very simple predictors, we are unable to capture the complexity of the dependence between x and y; but for overly complex predictors, we memorize the values of y at the expense of generalization.

TBD

DRAFT

Notionally, the ”P” axis can be thought of as ”complexity”, and we find a similar curve whenever the complexity of our learner increases. We will see that much of the practical aspects of machine learning come down to choosing and controlling our position on this curve, increasing the complexity when underfitting and decreasing it when overfitting.

Bias and Variance

A useful way to view the notion of overfitting is through the (frequenist) statistical concepts of bias and variance. Suppose that we train our model on a training data set D. Our training process will fit the model, finding a value for the parameters θ; since this depends on the data we used for training, we can explicitly write the trained parameters as $\theta(D)$.

Now, let us view our data set D as a random variable: a random subsample of examples we could have collected, each drawn from some true underlying distribution $p(x, y)$. Because D is random, $\theta(D)$ and thus the prediction $f(x; \theta(D))$ are as well. How much variability do we expect from different data sets? If f is a very simple function (such as a constant predictor), we would expect that it will not be too different, but may also not match the optimal predictor (for squared error, $f^*(x) = \mathbb{E}[y|x]$) very closely. As we allow f to be more flexible, it becomes more likely to be able to represent $f^*(x)$, but may also exhibit more randomness due to D.

We can quantify these two concepts in terms of *bias* and *variance*. The bias measures the accuracy of $f(x; \theta(D)$ *on average* over draws of D, while the variance is the variance of $f(x)$ due to D:

$$\begin{aligned} \text{Bias}_f(x) &= \bar{f}(x) - f^*(x) = \mathbb{E}_D\big[\, f(x; \theta(D))\big] - \mathbb{E}[y|x] \\ \text{Var}_f(x) &= \mathbb{E}_D\Big[\big(f(x) - \bar{f}(x)\big)^2\Big] \end{aligned}$$

Notice that both bias and variance are a function of the location x to be predicted.

Let us see an explicit example. Suppose that, as a thought experiment, we imagine making three clones of ourselves to go out and collect training data for our model. Each of our clones collects a different training data set D, each drawn from $p(x, y)$, but differing in exactly what examples were collected. We denote each clone’s data by a color (red, blue, or green) and plot the three data sets in Figure **??** (top row). Then, each clone estimates the optimal parameters $\theta(D)$ using their training set, for several different models (polynomial fits with degree $p = 0$, 1, and 3). On the bottom row of Figure **??**, we show the three models estimated by our clones, grouped by degree p.

We can see that the models are quite similar for $p = 0$; since even a few data are good enough to estimate θ_0 well. However, their linear fits ($p = 1$) are less similar to one another, and cubic fits ($p = 3$) are less similar still; the variance of f is increasing. (Note also that, not surprisingly, the models have less variation in the center of the data than at the edges.) Conversely, the simpler models are likely to be further from the true f^*, and thus have higher bias.

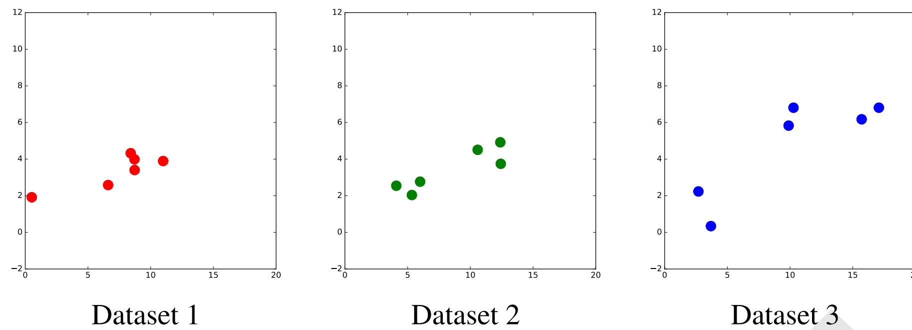
Bias and variance are different types of errors, but both contribute to the potential for poor performance. Consider the expected squared error of a regression model predicting a test point (x, y), in expectation over both the target point (x, y) and the training set D used to build the model. We can show that this decomposes into three parts:

$$\begin{aligned} &\mathbb{E}_{D,x,y}\Big[\big(y - f(x; \theta(D))\big)^2\Big] \\ &\qquad = \mathbb{E}_{x,y}\Big[(y - f^*(x))^2\Big] + \mathbb{E}_x\Big[(f^*(x) - \bar{f}(x))^2\Big] + \mathbb{E}_{D,x}\Big[(\bar{f}(x) - f(x; \theta(D)))^2\Big] \end{aligned}$$

The first of these terms is the expected squared error of the optimal predictor, $f^*(x) = \mathbb{E}[y|x]$. The second and third correspond to the squared bias, $\text{Bias}_f(x)^2$, and the variance, $\text{Var}_f(x)$, respectively (in expectation over x).

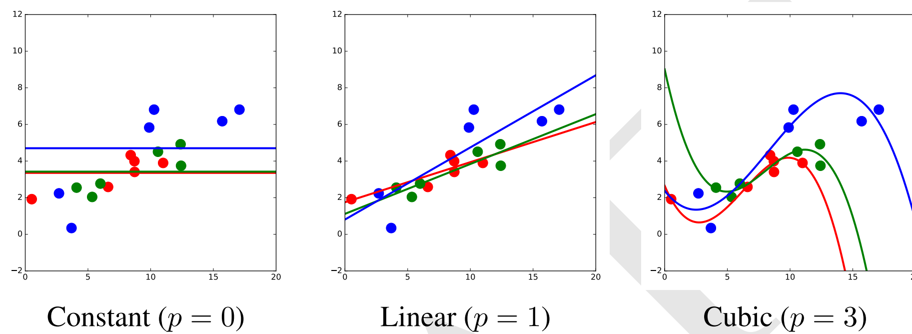
DRAFT

Data Sets:



Dataset 1 Dataset 2 Dataset 3

Models:



Constant ($p = 0$) Linear ($p = 1$) Cubic ($p = 3$)

Figure 4.5: Variability of our predictors $f(x; \theta(D))$, using the minimum MSE parameters $\theta(D)$ estimated using the data set D, for three different draws of D and different polynomial degrees p.

When the bias is the dominant factor in the error, this indicates that our model is underfitting – our model is insufficiently flexible to approximate f^*, even in expectation over the data set D. When variance is the dominant factor, the model is overfitting – our model is so flexible that different data sets D would produce very different prediction functions. Returning to our example, we can see that the flexible, degree $p = 3$ model fits its training data well, but does not fit the other data sets (as illustrated by the fact that each data set would choose a very different estimate of f), the hallmark of overfitting.

DRAFT