# Ordinary Least Squares Linear Regression

Ryan P. Adams
COS 324 – Elements of Machine Learning
Princeton University

Linear regression is one of the simplest and most fundamental modeling ideas in statistics and many people would argue that it isn't even machine learning. However, linear regression is an excellent starting point for thinking about supervised learning and many of the more sophisticated learning techniques in this course will build upon it in one way or another. Fundamentally, linear regression seeks to answer the question: "*What linear combination of inputs best explains the output?*"

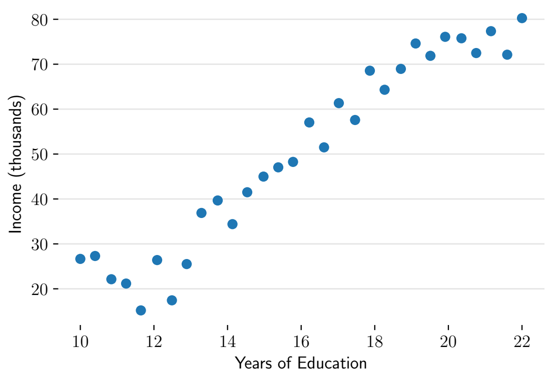
## Least Squares Regression with Scalar Inputs

For now, let's imagine the simplest possible situation, in which we have scalar real-valued features (inputs) and scalar real-valued labels (outputs). As we usually do in supervised learning, we take our training data to be $N$ tuples of these features and labels, denoted $\{x_n, y_n\}_{n=1}^N$, where in this case $x_n \in \mathbb{R}$ and $y_n \in \mathbb{R}$. Figure 1a shows such a data set, the `Income`[1] data, a synthetic data set with a hypothetical relationship between years of education and annual income.

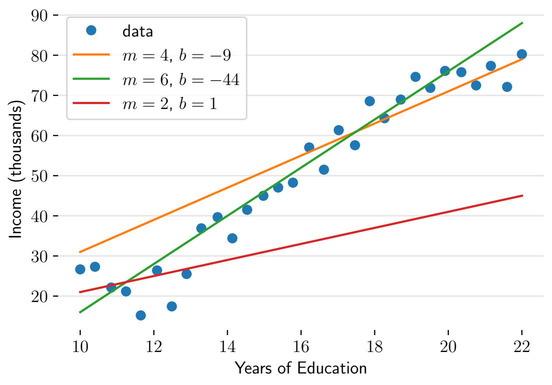
To do supervised learning, we first need a model. The model here is a family of straight-line functions that go from $\mathbb{R}$ to $\mathbb{R}$. Straight lines have a slope $m$ and an intercept $b$, with the equation for the line typically written as $y = mx + b$. When we say *family* in a case like this, we really mean *parametric family*; the $m$ and $b$ here are the parameters. Each possible value of $m \in \mathbb{R}$ and $b \in \mathbb{R}$ corresponds to a member of the family, also referred to as a *hypothesis*. One might then call the parametric family a *hypothesis class*. The learning question then is: given some data $\{x_n, y_n\}_{n=1}^N$ and a model, what are the best parameters $(m,b)$—member of the family—to fit the data? Figure 1b shows a couple of different values of $m$ and $b$ and you can think about which one might be best.

In order to reason about what it means to choose the "best" line from our family of lines, however, we will have to be clear about what it means to be good at explaining the data. There are many different ways to talk about what makes a good fit to data, but the most common framework in machine learning is to specify a *loss function*. Loss functions formalize how bad it is to produce output $\hat{y}$ when the truth was $y$. Larger numbers are worse; the loss is the cost of being wrong. In this course, I will write loss functions as $\ell(\hat{y}, y)$. In our basic linear regression setup here, $\ell : \mathbb{R}\times\mathbb{R} \to \mathbb{R}$, as it takes two real-valued arguments (prediction $\hat{y}$ and truth $y$) and produces a real-valued cost.

[1]The `Income` data set is from *An Introduction to Statistical Learning, with applications in R* (Springer, 2013) and is used with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.



(a) Raw `Income` data relating years of education to annual income.



(b) Some potential linear fits to the `Income` data with the parameterization $y = mx + b$.

Figure 1: Raw data and simple linear functions.

There are many different loss functions we could come up with to express different ideas about what it means to be bad at fitting our data, but by far the most popular one for linear regression is the *squared loss* or *quadratic loss*:

$$\ell(\hat{y}, y) = (\hat{y} - y)^2\,. \tag{1}$$

Figure 2a plots the squared loss function, but the intuition is simple: there is no cost if you get it exactly right, and the (non-negative) cost gets worse quadratically, so if you double $\hat{y} - y$, your cost goes up by a factor of four. Other desiderata and problem formulations lead to different loss functions. For example, in *robust regression* one wants to avoid being influenced by outlier data and might use an absolute value in Eq. 1 instead of the square. This still gives non-negative value that is zero if you're correct and bigger when you're wrong, but if you double $\hat{y} - y$ it only doubles your loss. We'll revisit loss functions multiple times during the course, but for now let's just stick with the vanilla squared loss.

For a single one of our data tuples $(x_n, y_n)$, the loss function lets us reason about how good a given set of parameters $m$ and $b$ are. We plug $x_n$ into the equation for the line and call that a prediction:

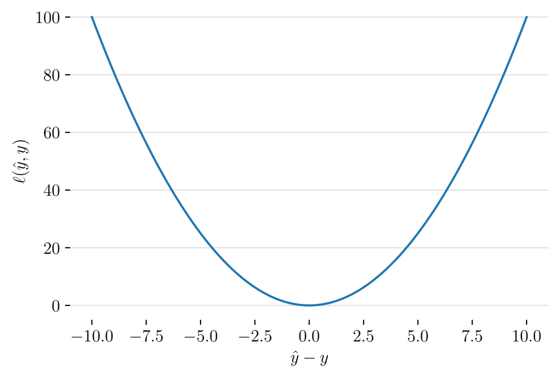
$$\hat{y}_n = mx_n + b\,. \tag{2}$$

Figure 2b shows the gap between the prediction and the true label for the `Income` data for an arbitrary set of parameters. For the $n$th datum, we get a squared loss

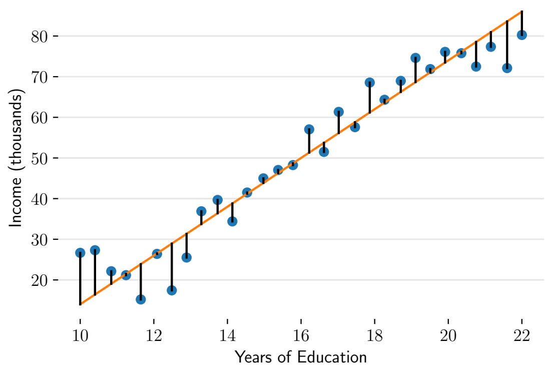
$$\ell(\hat{y}_n, y_n) = (mx_n + b - y_n)^2\,. \tag{3}$$

Of course, we have $N$ data, not just one, so we can look at the *average* loss across the data set, to give us an overall loss for all of the data together:

$$L(m, b) = \frac{1}{N}\sum_{n=1}^{N} \ell(\hat{y}_n, y_n) = \frac{1}{N}\sum_{n=1}^{N}(mx_n + b - y_n)^2\,. \tag{4}$$



(a) The squared loss function $\ell(\hat{y}, y) = (\hat{y} - y)^2$ is a simple quadratic function.



(b) A visualization of the "residuals" associated with the `Income` data for a linear function with $m = 6$ and $b = -46$.

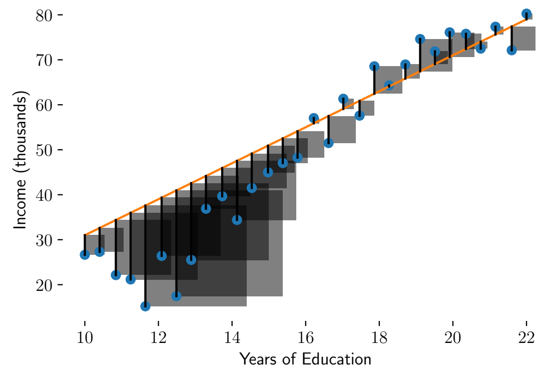
Figure 2: Squared loss and regression residuals

Figures 3a and 3b show a visualization of this quantity in you can get an intuition for how the squared error decomposes over the data set. This quantity is often called the *empirical risk*. The *risk* is the expected loss from using a particular hypothesis $(m, b)$ to make our predictions. We don't know what the true distribution over the data is, so we can't take an expectation under it; the *empirical* risk uses the empirical average under the observed data to estimate the risk associated with a particular hypothesis $(m, b)$. We assume the data are drawn independent and identically distributed (i.i.d.) from P(x,y), a joint probability distribution over X and Y. Our hope is that this average is pretty close to what we'd get if we somehow knew the true distribution, much in the same way that we poll small sets of people to estimate properties of the whole population. If that is true, when we might reasonably hope that for some unseen, or *out of sample*, data arrived for which we only had the input, then we could produce outputs that were probably good. Proving that this can happen—that machine learning models can generalize to unseen data—is its own subfield called *learning theory* that is out of the scope of this course. If you'd like to read more about learning theory, I suggest doing a search for the phrase "*PAC learning*", where PAC stands for "probably approximately correct".

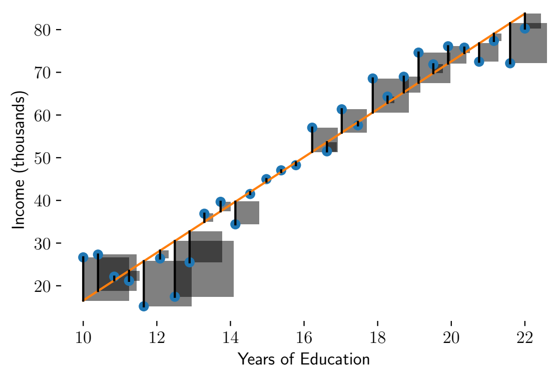
Having written down this empirical risk—the function that tells us how well $m$ and $b$ explain the data—we can try to minimize it with respect to these parameters. We can therefore write down the problem that represents the way we think about many kinds of supervised machine learning:

$$m^\star, b^\star = \underset{m,b}{\arg\min}\, L(m, b) = \underset{m,b}{\arg\min} \sum_{n=1}^{N} (mx_n + b - y_n)^2 \,. \tag{5}$$

The "arg min" here is a way to say that we're looking for the arguments that minimize $L(m, b)$. We call those the optimal parameters and denote them $m^\star$ and $b^\star$. We don't care about the $1/N$ factor because it doesn't change the minimum. This problem setup is sometimes called *empirical risk minimization*, as we are coming up with a loss function and trying to choose parameters that minimize the average loss on the training data.



(a) A visualization of the loss function that shows how the squared loss penalizes quadratically. The loss is the total area of the squares. Here $m = 6$ and $b = -46$.



(b) The same visualization as that shown on the left, but with the optimal parameters $m^\star = 5.6$ and $b^\star = -39.4$.

Figure 3: Visualizing aggregate squared loss

So how do we perform this minimization? Optimization is a big topic and we'll encounter many different ways to solve this kind of problem, but in this simple least squares setup we can derive the minimum in closed form. We first take the partial derivative of $L(m, b)$ with respect to both $m$ and $b$:

$$\frac{\partial}{\partial m}L(m,b) = \frac{2}{N}\sum_{n=1}^{N} x_n(mx_n + b - y_n) = 2m\left(\frac{1}{N}\sum_{n=1}^{N} x_n^2\right) + 2b\left(\frac{1}{N}\sum_{n=1}^{N} x_n\right) - 2\left(\frac{1}{N}\sum_{n=1}^{N} x_n y_n\right) \tag{6}$$

$$\frac{\partial}{\partial b}L(m,b) = \frac{2}{N}\sum_{n=1}^{N} mx_n + b - y_n = 2m\left(\frac{1}{N}\sum_{n=1}^{N} x_n\right) + 2b - 2\left(\frac{1}{N}\sum_{n=1}^{N} y_n\right). \tag{7}$$

Notice right away that these equations only involve the data via four averages, which I'll denote with their own symbols:

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x_n \qquad \bar{y} = \frac{1}{N}\sum_{n=1}^{N} y_n \tag{8}$$

$$s^2 = \frac{1}{N}\sum_{n=1}^{N} x_n^2 \qquad \rho = \frac{1}{N}\sum_{n=1}^{N} x_n y_n\,. \tag{9}$$

We can now set both partial derivatives to zero, eliminate the extra factor of two, and get a system of two equations with two unknowns:

$$\frac{1}{2}\frac{\partial}{\partial m}L(m,b) = ms^2 + b\bar{x} - \rho = 0 \qquad \frac{1}{2}\frac{\partial}{\partial b}L(m,b) = m\bar{x} + b - \bar{y} = 0 \tag{10}$$

We can straightforwardly solve this system via variable elimination to get:

$$m^\star = \frac{\rho - \bar{x}\bar{y}}{s^2 - \bar{x}^2} = \frac{\frac{1}{N}\sum_{n=1}^N x_n y_n - \left(\frac{1}{N}\sum_{n=1}^N x_n\right)\left(\frac{1}{N}\sum_{n=1}^N y_n\right)}{\frac{1}{N}\sum_{n=1}^N x_n^2 - \left(\frac{1}{N}\sum_{n=1}^N x_n\right)^2} \qquad b^\star = \bar{y} - \bar{x}m^\star \tag{11}$$

You might reasonably ask whether setting these partial derivatives to zero is enough to ensure that the critical point we find is a minimum. One of the reasons we like this simple linear least squares setup is because $L(m, b)$ is convex and so the solution to the system of equations above is the global minimum.

## Least Squares Regression with Multiple Inputs

Let's now consider the case where instead of a single feature $x_n \in \mathbb{R}$ we have $D$-dimensional features $\boldsymbol{x}_n \in \mathbb{R}^D$. We'll continue to use scalars for the outputs $y_n \in \mathbb{R}$. Now our regression functions are sums over the dimensions and in addition to the intercept $b$ (often called a *bias*) we also have vector valued regression weights, which we'll denote here as $\boldsymbol{w} \in \mathbb{R}^D$, so that $y_n = b + \sum_{d=1}^D w_d x_{n,d}$. The subscripts $x_{n,d}$ indicate the $d$th dimension of the $n$th data vector $\boldsymbol{x}_n$. We're using the machine learning convention of $\boldsymbol{w}$ for "weights", but note that statisticians almost universally use $\boldsymbol{\beta}$ for regression weights.

Let's introduce a couple of notational shortcuts. First, rather than writing out the sum over dimensions, let's use vector notation and make it an inner product, i.e., $\sum_{d=1}^D w_d x_{n,d} = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$. Second, rather than having a separate bias term $b$, we can always append a 1 to the end of our feature vectors $\boldsymbol{x}_n$. Then, $b$ just becomes one of the elements of $\boldsymbol{w}$ and can be treated like all the other regression weights. With these two shortcuts, we can now write the regression function a bit more concisely as $y_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$. In fact, we can take it even a step farther and introduce some notation that stacks things up over the $n$ indices as well:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^\mathsf{T} \\ \boldsymbol{x}_2^\mathsf{T} \\ \vdots \\ \boldsymbol{x}_N^\mathsf{T} \end{bmatrix} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,D} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,D} \end{bmatrix} \qquad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}. \tag{12}$$

The matrix $\boldsymbol{X}$ is often called the *design matrix*. These notational shortcuts allow us to write "in parallel" all of the $N$ input-output relations as a single matrix-vector product $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w}$.

The sum-of-squares loss function can also be written in vector notation as an inner product, giving us a new very concise loss function:

$$L(\boldsymbol{w}) = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\mathsf{T}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) = ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2\,. \tag{13}$$

The last part on the right above is denoting this quantity in a slightly different way, as a squared *vector norm*. Vector norms are just ways to talk about the lengths of vectors and in this case the norm is the Euclidean one that you're used to, also called the $\ell_2$ norm. With a notion of length, we also have a notion of distance and so here the norm is measuring how far our model's predictions are from the true labels. The compactness of this vector notation can be intimidating at first, but it is helpful for avoiding the bookkeeping of indices and for eventually building better geometric intuition from the linear algebra. Also, modern scientific computing—things like graphical processing units (GPUs)—really like to have their operations constructed from simple vectorized primitives; putting things into vector form makes writing fast code easier down the line.

We're now in a position to solve for $\boldsymbol{w}$ using essentially the same procedure we did before. We take the derivative of $L(\boldsymbol{w})$ with respect to the vector $\boldsymbol{w}$ and set that *gradient* to zero:

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = 2\boldsymbol{X}^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) = 0\,. \tag{14}$$

Then, we solve for $\boldsymbol{w}$. Moving things around we get:

$$\boldsymbol{X}^{\mathsf{T}}\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}^{\mathsf{T}}\boldsymbol{y} \tag{15}$$

If $\boldsymbol{X}^{\mathsf{T}}\boldsymbol{X}$ is full-rank, which usually happens when $N \geq D$ and there are no repeated input vectors—when the columns of $\boldsymbol{X}$ are linearly independent—then we can find $\boldsymbol{w}$ by solving the linear system:

$$\boldsymbol{w}^{\star} = \left(\boldsymbol{X}^{\mathsf{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathsf{T}}\boldsymbol{y}\,. \tag{16}$$

This system is sometimes called the *normal equations*. If you've taken some linear algebra, you might recognize this as the Moore-Penrose pseudoinverse of $\boldsymbol{X}$.

## Changelog

- 27 August 2018 – Initial version