

6.036 Spring 2018 : Week 3

February 25, 2018

- Least squares criterion for regression
- Stochastic gradient descent for fitting linear regression models
- Closed form linear regression solution
- Regularization and how it changes the solution, generalization

4 Linear regression

We have so far tried to predict a binary label $y \in \{-1, 1\}$ for each input example x. In many cases, the target we wish to predict is a real number instead. For example, we may be interested in predicting the value of a stock some time into the future (e.g., tomorrow) rather than just whether it will go up (+1) or down (-1). Similarly, we can predict the degree to which a user likes a certain product (e.g., 3.5 stars) rather than just whether they do or don't like it. Supervised learning problems where the target is a real number are called regression problems. More formally, our task is to find a function $f : \mathbb{R}^d \to \mathbb{R}$ such that $f(x) \approx y$ for all test examples x. As before, we have to select f based on a finite training set prior to seeing any test examples. In order to generalize well to the test examples (not to overfit), we need to limit the family of functions we consider, just as in classification. We will begin with simple linear functions, analogously to linear classification.

A linear regression function is simply a linear function of the feature vectors, i.e.,

$$f(x; \theta, \theta_0) = \theta \cdot x + \theta_0 = \sum_{i=1}^{d} \theta_i x_i + \theta_0 \tag{1}$$

Each setting of the parameters θ and θ_0 gives rise to a slightly different regression function. Collectively, different parameter choices $\theta \in \mathbb{R}^d$, $\theta_0 \in \mathbb{R}$, give rise to the set of functions $\mathcal{F}$ that we are entertaining. While this set of functions seems quite simple, just linear, the power of $\mathcal{F}$ is hidden in the feature vectors. Indeed, we can often construct different feature representations for objects. For example, there are many ways to map past values of financial assets into a feature vector x, and this mapping is typically completely under our control. This "freedom" gives us a lot of power, and we will discuss how to exploit it later on. To start with, we assume that a proper representation has been found, denoting feature vectors simply as x for convenience.

Our learning task is to choose one $f \in \mathcal{F}$, i.e., choose parameters $\hat{\theta}$ and $\hat{\theta}_0$, based on the training set $S_n = \{(x^{(t)}, y^{(t)}), t = 1, \ldots, n\}$, where $y^{(t)} \in \mathbb{R}$ (response). As before, our goal is to find $f(x; \hat{\theta}, \hat{\theta}_0)$ that would yield accurate predictions on yet unseen examples. There are several problems to address:

(1) How do we measure error? What is the criterion by which we choose $\hat{\theta}$ and $\hat{\theta}_0$ based on the training set?

(2) What algorithm can we use to optimize the training criterion? How does the algorithm scale with the dimension (feature vectors may be high dimensional) or the size of the training set (the dataset may be large)?

(3) When the size of the training set is not large enough in relation to the number of parameters (dimension) there may be degrees of freedom, i.e., directions in the parameter space, that remain unconstrained by the data. How do we set those degrees of freedom? This is a part of a broader problem known as *regularization*. The question is how to softly constrain the set of functions $\mathcal{F}$ to achieve better generalization.

4.1 Empirical risk and the least squares criterion

As in the classification setting, we will measure training error in terms of the average loss or *empirical risk*

$$R_n(\theta) = \frac{1}{n} \sum_{t=1}^{n} \text{Loss}(y^{(t)} - \theta \cdot x^{(t)}) \tag{2}$$

where, for simplicity, we have dropped the parameter θ_0. It will be easy to add it later on in the appropriate place. Note that, unlike in classification, the loss function now depends on the difference between the real valued target $y^{(t)}$ and the corresponding linear prediction $\theta \cdot x^{(t)}$. There are many possible ways of defining the loss function. We will use here a simple squared error: $\text{Loss}(z) = z^2/2$, where the division by 2 is entirely for convenience. The idea is to permit small discrepancies (we expect the responses to include noise) but heavily penalize large deviations that typically indicate poor parameter choices. As a result, we have

$$R_n(\theta) = \frac{1}{n} \sum_{t=1}^{n} \text{Loss}(y^{(t)} - \theta \cdot x^{(t)}) = \frac{1}{n} \sum_{t=1}^{n} (y^{(t)} - \theta \cdot x^{(t)})^2/2 \tag{3}$$

Recall that our learning goal in supervised learning is not to minimize $R_n(\theta)$; it is just the best we can do (for now). Minimizing $R_n(\theta)$ is a surrogate or proxi criterion since we don't have a direct access to the test or generalization error

$$R^{test}_{n'}(\theta) = \frac{1}{n'} \sum_{t=n+1}^{n+n'} (y^{(t)} - \theta \cdot x^{(t)})^2/2 \tag{4}$$

Let's briefly consider how $R_n(\theta)$ and $R^{test}_{n'}(\theta)$ are related. In the simplest setting, we select $\hat{\theta}$ by minimizing $R_n(\theta)$ but our performance will be eventually measured according to the

test error $R^{test}_{n'}(\hat{\theta})$. This test error can be large for two different reasons. First, we may have a large *estimation error*. This means that, even if the true relationship between x and y were linear, it would hard for us to estimate or recover it on the basis of just a small (and potentially noisy) training set S_n. Our estimated parameters $\hat{\theta}$ will not be entirely correct. The larger the training set is, the smaller the estimation error will be. The second type of error that also affects our test error is *structural error*. This means that we may be estimating a linear mapping from x to y when the true underlying relationship is highly non-linear. Clearly, we couldn't do very well in this case, regardless of how large the training set were. In order to reduce structural error, we would have to use a larger set of functions $\mathcal{F}$, e.g., by including polynomial features as additional coordinates. A larger set of functions ensures that there would at least be one $f \in \mathcal{F}$ that approximates the true underlying mapping well. But the problem is that, given only a noisy training set S_n, it will be harder to select this correct function from a large set of candidates $\mathcal{F}$. Our estimation error would necessarily increase. Finding the balance between the estimation and structural errors lies at the heart of effective learning.

When we formulate linear regression problem as a statistical problem, we imagine that the targets have been generated by some underlying function with noise added. In this case, we can talk about the structural error as "bias", i.e., how far off from the true function we are in expectation over different choices of training sets. The estimation error, on the other hand, corresponds to the "variance" of the function or the parameter *estimator* $\hat{\theta}(S_n)$. The parameters $\hat{\theta}$ are obtained with the help of training data S_n and thus can be viewed as functions of S_n. An estimator is a mapping from data to parameters. If we consider a large set of functions, our bias is low but the estimated parameters will vary quite a bit from one training set to another (high variance) since we can easily find a function that fits the noise as well.

4.2 Optimizing the least squares criterion

Perhaps the simplest way to optimize the least squares objective $R_n(\theta)$ is to use the stochastic gradient descent method discussed earlier in the classification context. Our case here is easier, in fact, since $R_n(\theta)$ is everywhere differentiable. At each step of the algorithm, we select one training example at random, and nudge parameters in the opposite direction of the gradient

$$\nabla_\theta (y^{(t)} - \theta \cdot x^{(t)})^2/2 = (y^{(t)} - \theta \cdot x^{(t)}) \nabla_\theta (y^{(t)} - \theta \cdot x^{(t)}) = -(y^{(t)} - \theta \cdot x^{(t)}) x^{(t)} \tag{5}$$

As a result, the algorithm can be written as

$$\begin{array}{l} \text{set } \theta^{(0)} = 0 \\ \quad \text{randomly select } t \in \{1, \ldots, n\} \\ \quad\quad \theta^{(k+1)} = \theta^{(k)} + \eta_k (y^{(t)} - \theta \cdot x^{(t)}) x^{(t)} \end{array} \tag{6}$$

where η_k is the learning rate (e.g., $\eta_k = 1/(k+1)$). Recall that, for example, in the perceptron algorithm, update was performed only if we made a mistake. Now the update is proportional to discrepancy $(y^{(t)} - \theta \cdot x^{(t)})$ so that any error, however small, counts. As in the classification context, the update is "self-correcting". For example, if our prediction

is lower than the target, i.e., $y^{(t)} > \theta \cdot x^{(t)}$, we would move the parameter vector in the positive direction of $x^{(t)}$ so as to increase the prediction next time $x^{(t)}$ is considered. This would happen in the absence of updates based on other examples.

4.2.1 Closed form solution

We can also try to minimize $R_n(\theta)$ directly by setting the gradient to zero. Indeed, since $R_n(\theta)$ is a convex function of the parameters, the minimum value is obtained at a point (or a set of points) where the gradient is zero. So, formally, we find $\hat{\theta}$ for which $\nabla R_n(\theta)_{\theta=\hat{\theta}} = 0$. More specifically,

$$\begin{aligned}
\nabla R_n(\theta)_{\theta=\hat{\theta}} &= \frac{1}{n}\sum_{t=1}^{n} \nabla_\theta \big\{ (y^{(t)} - \theta \cdot x^{(t)})^2/2 \big\}_{|\theta=\hat{\theta}} && (7)\\
&= \frac{1}{n}\sum_{t=1}^{n} \big\{ -(y^{(t)} - \hat{\theta} \cdot x^{(t)})x^{(t)} \big\} && (8)\\
&= -\frac{1}{n}\sum_{t=1}^{n} y^{(t)}x^{(t)} + \frac{1}{n}\sum_{t=1}^{n} (\hat{\theta} \cdot x^{(t)})x^{(t)} && (9)\\
&= \underbrace{-\frac{1}{n}\sum_{t=1}^{n} y^{(t)}x^{(t)}}_{=b} + \underbrace{\frac{1}{n}\sum_{t=1}^{n} x^{(t)}(x^{(t)})^T}_{=A}\ \hat{\theta} && (10)\\
&= -b + A\hat{\theta} = 0 && (11)
\end{aligned}$$

where we have used the fact that $\hat{\theta} \cdot x^{(t)}$ is a scalar and can be moved to the right of vector $x^{(t)}$. We have also subsequently rewritten the inner product as $\hat{\theta} \cdot x^{(t)} = (x^{(t)})^T\hat{\theta}$. As a result, the equation for the parameters can be expressed in terms of a $d \times 1$ column vector b and a $d \times d$ matrix A as $A\theta = b$. When the matrix A is invertible, we can solve for the parameters directly: $\hat{\theta} = A^{-1}b$. In order for A to be invertible, the training points $x^{(1)}, \ldots, x^{(n)}$ must *span* $\mathcal{R}^d$. Naturally, this can happen only if $n \geq d$, and is therefore more likely to be the case when the dimension d is small in relation to the size of the training set n. Another consideration is the cost of actually inverting A. Roughly speaking, you will need $\mathcal{O}(d^3)$ operations for this. If $d = 10,000$, this can take a while, making the stochastic gradient updates more attractive.

In solving linear regression problems, the matrix A and vector b are often written in a slightly different way. Specifically, define $X = [x^{(1)}, \ldots, x^{(n)}]^T$. In other words, X^T has each training feature vector as a column; X has them stacked as rows. If we also define $\vec{y} = [y^{(1)}, \ldots, y^{(n)}]^T$ (column vector), then you can easily verify that

$$b = \frac{1}{n}X^T\vec{y}, \quad A = \frac{1}{n}X^TX \tag{12}$$

4.3 Regularization

What happens when A is not invertible? In this case the training data provide no guidance about how to set some of the parameter directions. For example, if one of the coordinates

$x_k^{(i)} = 0$ for all $i = 1, \ldots, n$, we would not know how to set the corresponding parameter θ_k. We say in this case that the learning problem is ill-posed. The same issue inflicts the stochastic gradient method as well though the initialization $\theta^{(0)} = 0$ helps set the parameters to zero for directions outside the span of the training examples, thus correcting the simple problematic case (θ_k would be set to zero). The simple fix does not solve the broader problem, however, including when the coordinates of feature vectors are linearly dependent. How should we set the parameters when we have insufficient training data?

We will modify the estimation criterion, the mean squared error, by adding a *regularization term*. The purpose of this term is to bias the parameters towards a default answer such as zero. The regularization term will "resist" setting parameters away from zero, even when the training data may weakly tell us otherwise. This resistance is very helpful in order to ensure that our predictions generalize well. The intuition is that we opt for the "simplest answer" when the evidence is absent or weak.

There are many possible regularization terms that fit the above description. In order to keep the resulting optimization problem easily solvable, we will use $\|\theta\|^2/2$ as the penalty. Specifically, we will minimize

$$J_{n,\lambda}(\theta) = \frac{\lambda}{2}\|\theta\|^2 + R_n(\theta) = \frac{\lambda}{2}\|\theta\|^2 + \frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \theta \cdot x^{(t)})^2/2 \tag{13}$$

where the *regularization parameter* $\lambda \geq 0$ quantifies the trade-off between keeping the parameters small – minimizing the squared norm $\|\theta\|^2/2$ – and fitting to the training data – minimizing the empirical risk $R_n(\theta)$. The use of this modified objective is known as *Ridge regression*.

While important, the regularization term introduces only small changes to the two estimation algorithms. For example, in the stochastic gradient descent algorithm, in each step, we will now move in the reverse direction of the gradient

$$\nabla_\theta \Big\{ \frac{\lambda}{2}\|\theta\|^2 + (y^{(t)} - \theta \cdot x^{(t)})^2/2 \Big\}_{|\theta=\theta^{(k)}} = \lambda\theta^{(k)} - (y^{(t)} - \theta^{(k)} \cdot x^{(t)})x^{(t)} \tag{14}$$

As a result, the algorithm can be rewritten as

$$\begin{array}{l} \text{set } \theta^{(0)} = 0 \\ \quad \text{randomly select } t \in \{1, \ldots, n\} \\ \quad\quad \theta^{(k+1)} = (1 - \lambda\eta_k)\theta^{(k)} + \eta_k(y^{(t)} - \theta \cdot x^{(t)})x^{(t)} \end{array} \tag{15}$$

As you might expect, there's now a new factor $(1 - \lambda\eta_k)$ multiplying the current parameters $\theta^{(k)}$, shrinking them towards zero during each update.

When solving for the parameters directly, the regularization term only modifies the $d \times d$ matrix $A = \lambda I + (1/n)\, X^T X$, where I is the identity matrix. The resulting matrix is *always* invertible so long as $\lambda > 0$. The cost of inverting it remains the same, however.

The effect of regularization

The regularization term shifts emphasis away from the training data. As a result, we should expect that larger values of λ will have a negative impact on the training error. Specifically,

let $\hat{\theta} = \hat{\theta}(\lambda)$ denote the parameters that we would find by minimizing the regularized objective $J_{n,\lambda}(\theta)$. We view $\hat{\theta}(\lambda)$ here as a function of λ. We claim then that $R_n(\hat{\theta}(\lambda))$, i.e., mean squared training error, increases as λ increases. If the training error increases, where's the benefit? Larger values of λ actually often lead to lower generalization error as we are no longer easily swayed by noisy data. Put another way, it becomes harder to over-fit to the training data. This benefit accrues for a while as λ increases, then turns to hurt us. Biasing the parameters towards zero too strongly, even when the data tells us otherwise, will eventually hurt generalization performance. As a result, you will see a typical U-shaped curve in terms of how the generalization error depends on the regularization parameter λ.