Overfitting and Regularization

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

Machine learning doesn't always work. Actually, in a fundamental way we should be surprised that machine learning is even possible! Consider the problem of learning from examples an unknown Boolean function $f : \{0,1\}^D \to \{0,1\}$, i.e., a function from a binary vector to a single binary value. Our goal is to learn a deterministic Boolean formula for this unknown function, using N examples $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$ where $\boldsymbol{x}_n \in \{0,1\}^D$ and $y_n \in \{0,1\}$. For notational convenience, imagine that M of the y_n are 1 and $N - M$ are 0, and that the indices of the pairs are such that $y_1, \ldots, y_M$ are the 1's and $y_{M+1}, \ldots, y_N$ are the 0's.

Suppose a new instance $\boldsymbol{x}'$ arrives and we wish to predict whether it yields a 0 or 1 according to the Boolean formula we are trying to estimate. If one of our training data matches $\boldsymbol{x}'$ exactly, then this is easy: we just return the y_n for the match. But what if it doesn't match one of our training data? Why do we think our training data provides information about this unseen example? The function we're trying to learn could just be a binary vector with 2^D entries in it and no structure, with $\boldsymbol{x}$ just indexing into one of the dimensions. Knowing $N < 2^D$ of the entries in the vector does not in general tell us anything at all about the other $2^D - N$.

To see this more clearly, we can come up with two functions that are both consistent with the data, try to make minimal assumptions, but that make exactly opposite predictions for all out-of-sample $\boldsymbol{x}'$. The first such function tries to minimize its assumptions by being *specific*, that is, by avoiding predicting a 1 unless it is certain about the label, based on the training set. It classifies everything else as zero:

$$f_{\text{specific}}(\boldsymbol{x}') = \bigvee_{n=1}^{M} \bigwedge_{d=1}^{D} (x_{n,d} \wedge x'_d) \vee (\neg x_{n,d} \wedge \neg x'_d) \,. \tag{1}$$

Another coherent function is to say that we want to be maximally *general* and that we should classify things as 1 unless we know them to be zero. As such, it takes a liberal view and only produces a zero if a matching zero example is in the data set. Otherwise, it produces a 1:

$$f_{\text{general}}(\boldsymbol{x}') = \neg \bigvee_{n=M+1}^{N} \bigwedge_{d=1}^{D} (x_{n,d} \wedge x'_d) \vee (\neg x_{n,d} \wedge \neg x'_d) \,. \tag{2}$$

*These notes are heavily influenced by Harvard CS181 course notes by Avi Pfeffer and David Parkes.

These are both consistent with the training data and yet provide completely different answers for every datum not in the training data. This should make it clear that we really don't get anything intrinsic from the training data itself about other unseen examples. We are going to have to make additional assumptions about what kinds of functions we can consider, if we are going to be able to generalize to out-of-sample data. Fitting the training data well is not enough.

Learning is induction. Induction is only possible if we are willing to posit structure that exists in both data we have seen and data we have not yet seen.

> *That even after the observation of the frequent or constant conjunction of objects, we have no reason to draw any inference concerning any object beyond those of which we have had experience.*
> —David Hume, *A Treatise of Human Nature*, Book I, Part 3, Section 12

Inductive Bias

For learning to be possible, we must posit structure that enables generalization and we must have a way to prefer some hypotheses about the world over others. The assumptions we make about this structure—about the functions we try to capture with data—we call our *inductive bias*.

Broadly speaking, there are two kinds of inductive bias: restriction bias and preference bias. A restriction bias says that only some kinds of hypotheses are possible. For example, when trying to fit a Boolean function we might limit our functions to only consider *conjunctive concepts*. A conjunctive concept is a Boolean function that can only be a logical AND over literals, i.e., over single features or their negations.

A preference bias, on the other hand, is a kind of ranking over potential hypotheses. When two hypotheses are both consistent with the data, then we choose the one with the higher ranking. The most common approach to the construction of such a ranking is to specify a penalty function. For example, we might penalize Boolean functions that have larger circuits, or require more clauses in conjunctive normal form. Among hypotheses that explain the training data equally well, we choose the one with the lowest penalty. Another approach to constructing a preference bias is to have a search order through hypotheses, and stop the search when a hypothesis is encountered that explains the training data sufficiently well. Thus the search process implicitly produces a preference ordering on hypotheses.

The selection of an inductive bias is an extremely important aspect of machine learning. The inductive bias is literally what sorts of things can be learned and generalized from. The *No Free Lunch Theorem*[1] shows that no inductive bias can in general be "the right one" across all universes; for any set of target functions on which an inductive bias works well, it will work badly on the complement of that set. It is tempting to try to do machine learning "without making assumptions" but *there is no such thing* because generalization is not possible without assumptions. The important thing is to be clear about what those assumptions are, and construct them carefully for the problem you wish to solve.

[1] Wolpert, David H. "The lack of a priori distinctions between learning algorithms." *Neural Computation* 8.7 (1996): 1341-1390.

Overfitting and Underfitting

There are lots of ways that a machine learning algorithm can fail, but two of the big failure modes arise from poor choice of inductive bias: *underfitting* and *overfitting*. Underfitting happens when the hypothesis space is too restricted to capture important structure about the function being modeled, even though that structure is present in the training data. Overfitting happens when the hypothesis space is large enough that it is possible to fit noise in the training data that is not generalizable structure. In both cases, test (out-of-sample) performance suffers, but when overfitting it appears that the training is succeeding via low training error. As such, overfitting is one of the biggest issues in machine learning, across almost all models and algorithms. The difficulty is a fundamental tension between the objective of fitting the data well, and generalizing to new examples. Note in particular, that "fitting the noise" does not necessarily only arise when the data are noisy; rather it is possible for a learning algorithm to identify spurious patterns in noise-free but finite data sets.

Figure 1 gives an example of how overfitting can occur as the model becomes more flexible. In this case, I generated 30 training data and 30 test data and fit regression weights to the training data with least squares using polynomials of increasing degree. The x_n in both training and test were drawn uniformly from the interval $[-5, 5]$ and the true function is $y = 2\cos(x) - 2\tanh(x)$, with noise variance $\sigma^2 = 1$. Each figure shows the training and test losses in the bottom left, and then they are all plotted together in Figure 2. In Figure 2 the overfitting phenomenon is clear: increasing the flexibility of the model (increasing degree, shown on the horizontal axis) makes it possible to reduce the training loss, while at some point that flexibility hurts generalization performance as seen by the increase in test loss.

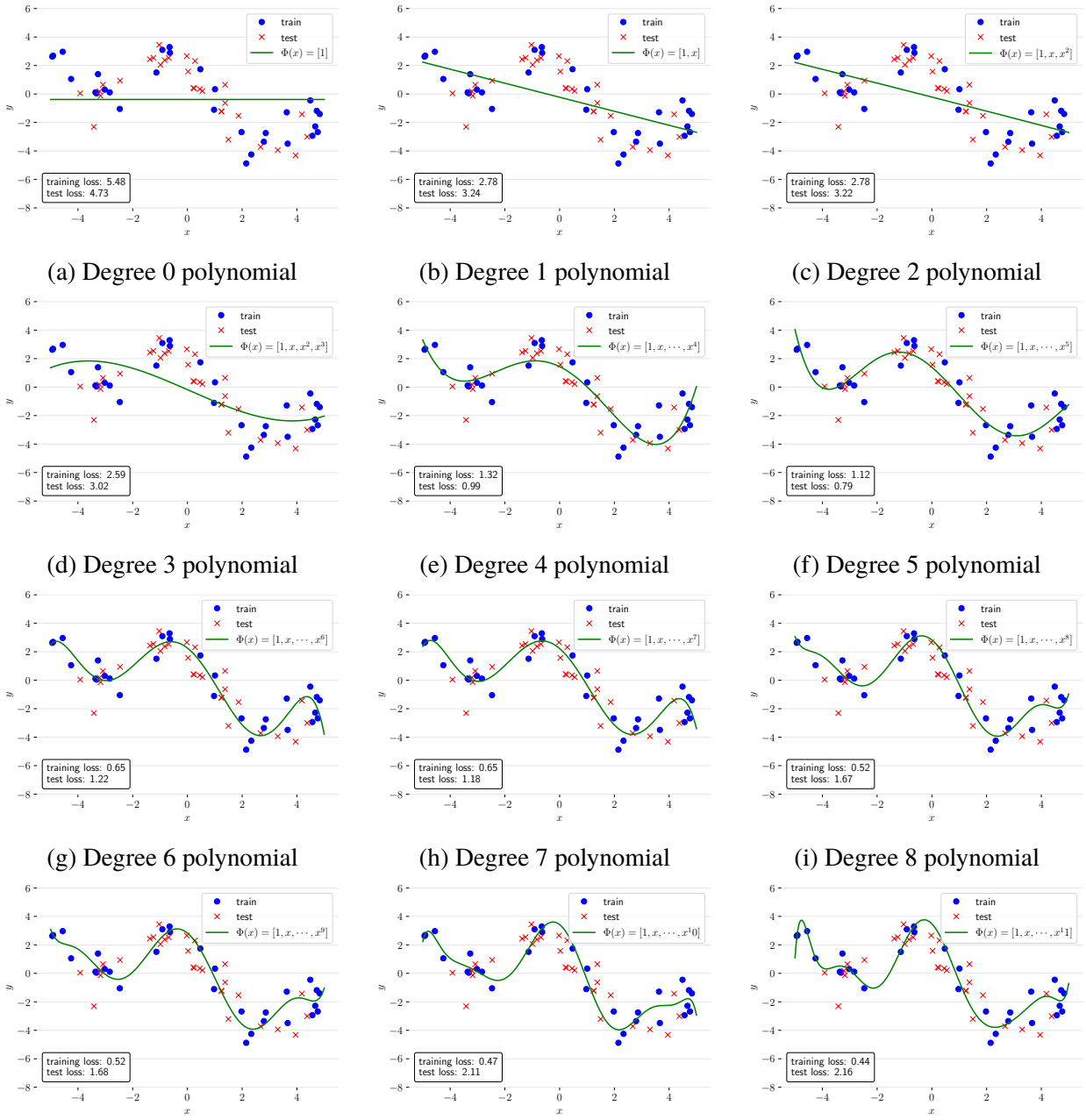
Overfitting can occur for a variety of reasons, such as:

Too few training data Small data sets are more likely to have spurious patterns that would not arise with a larger training set. For example, if you only see the buildings on Princeton campus, you might draw very strange conclusions about what all buildings must look like.

Noise in the data If the signal-to-noise ratio is bad in the data, it may be very hard to identify structure that can be learned. Algorithms may instead try to predict from the noise.

The hypothesis space is too large When the hypothesis space is small, i.e., the inductive bias is strong, the model is less likely to have the capacity to fit noise. Noise in the data are unlikely to fit exactly onto a simple hypothesis. However, the richer the hypothesis space is, the greater the possibility is that spurious signals can be latched onto in the training data, or that irrelevant features can appear to impact the label.

The input space is high-dimensional Every time you add a feature (or a basis function!) there is some chance that this feature will have random noise that happens to align with the label. This is particularly problematic in domains where there are huge numbers of features that may not matter, as in statistical genetics. Trying to limit the number of features is the motivation behind methods that perform *feature selection* in machine learning and statistics.



(a) Degree 0 polynomial (b) Degree 1 polynomial (c) Degree 2 polynomial

(d) Degree 3 polynomial (e) Degree 4 polynomial (f) Degree 5 polynomial

(g) Degree 6 polynomial (h) Degree 7 polynomial (i) Degree 8 polynomial

(j) Degree 9 polynomial (k) Degree 10 polynomial (l) Degree 11 polynomial

Figure 1: Fitting polynomials with least squares out to degree 11. There are 30 training data and 30 test data, both from the same distribution. The training and test losses are shown in the bottom left of each figure.

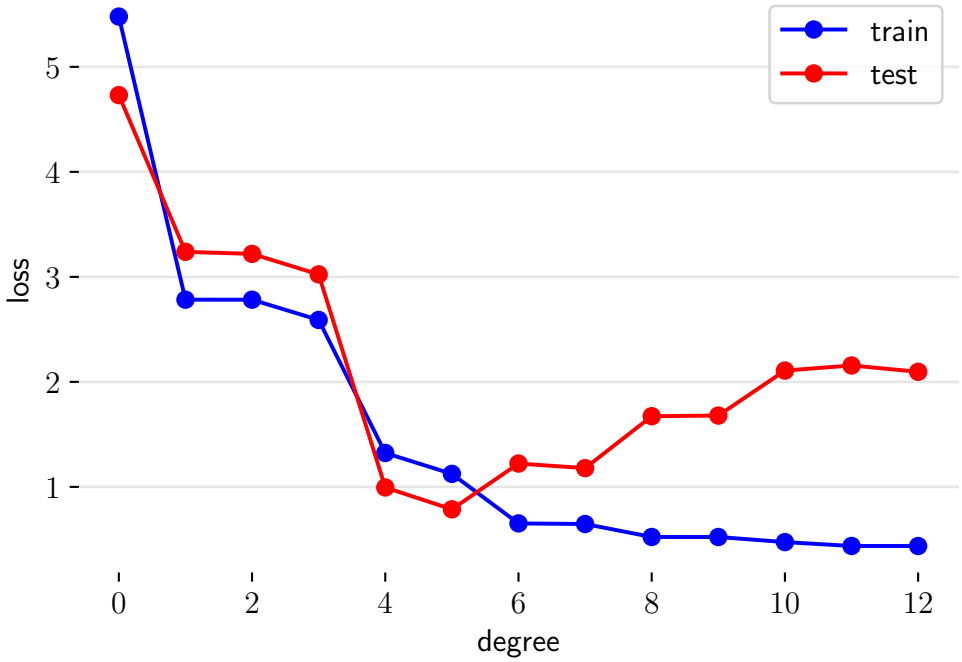


Figure 2: Test versus training losses, as a function of polynomial degree for the data and fits shown in Figure 1. The training loss improves with increasing flexibility of the polynomial, and would go to zero when the degree was high enough to go through all of the training data. Test loss decreases initially, but then starts to increase again. This is the overfitting phenomenon.

Bias Versus Variance

One of the ways we understand underfitting versus overfitting is via the *bias-variance tradeoff*. Imagine that there is a distribution over the inputs $\Pr(x)$ and some true function $f(x)$. The observations y that we get are the result of adding noise ϵ to $f(x)$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. When we get a set of N data, we can think of these data as being a random variable which I will denote as $\mathcal{D}$. We fit a model to this randomly chosen data set and get an approximate function $\hat{f}(x)$. The thing we want $\hat{f}(x)$ to be good at is minimizing the expected squared error under the joint distribution over x and y for some new datum.

$$\text{Goal is to minimize:} \quad \mathbb{E}_{\mathcal{D},x,y}\left[(y - \hat{f}(x))^2\right] . \tag{3}$$

The expectation here is taken over three things: the dataset $\mathcal{D}$ that we used to train $\hat{f}$, the test input x, and the test label y. There are two kinds of things we might worry about when thinking about building a model and fitting $\hat{f}$ to $\mathcal{D}$: the *bias* of $\hat{f}$ and the *variance* of $\hat{f}$. The variance of $\hat{f}$ reflects the amount of fluctuation we get in our estimates when we get different data sets:

$$\text{Var}[\hat{f}] = \mathbb{E}_{\mathcal{D},x}\left[(\hat{f}(x) - \mathbb{E}_{\mathcal{D}}[\hat{f}(x)])^2\right] . \tag{4}$$

Figure 3 demonstrates this in the same setup as was used to produce Figure 1. Twenty data sets were generated from the same distribution and fit with polynomials of increasing degree, plotting each of the functions. The variability of these functions around the mean is evident, and gets larger with increasing degree. The squared bias of $\hat{f}(x)$, on the other hand, is the expected squared deviation of our estimate from the true underlying function:

$$\text{Bias}^2[\hat{f}] = \mathbb{E}_x \left[(\mathbb{E}_{\mathcal{D}}[\hat{f}(x)] - f(x))^2 \right] . \tag{5}$$

Figure 4 illustrates the concept of bias. The mean $\hat{f}(x)$ is shown for each degree of polynomial (as computed from the samples in Figure 3), compared to the true function.

It turns out that we can rewrite Eqn. 3 as a sum of the variance of $\hat{f}$, the squared bias of $\hat{f}$, and the observation noise variance σ^2. First, recall that the variance of a random variable Z can be written as the difference between its expected square and the square of its expectation:

$$\text{Var}[Z] = \mathbb{E}[Z^2] - \mathbb{E}[Z]^2 . \tag{6}$$

We can use this identity to replace quantities like $\mathbb{E}[Z^2]$ with $\text{V}[Z] + \mathbb{E}[Z]^2$ in the following:

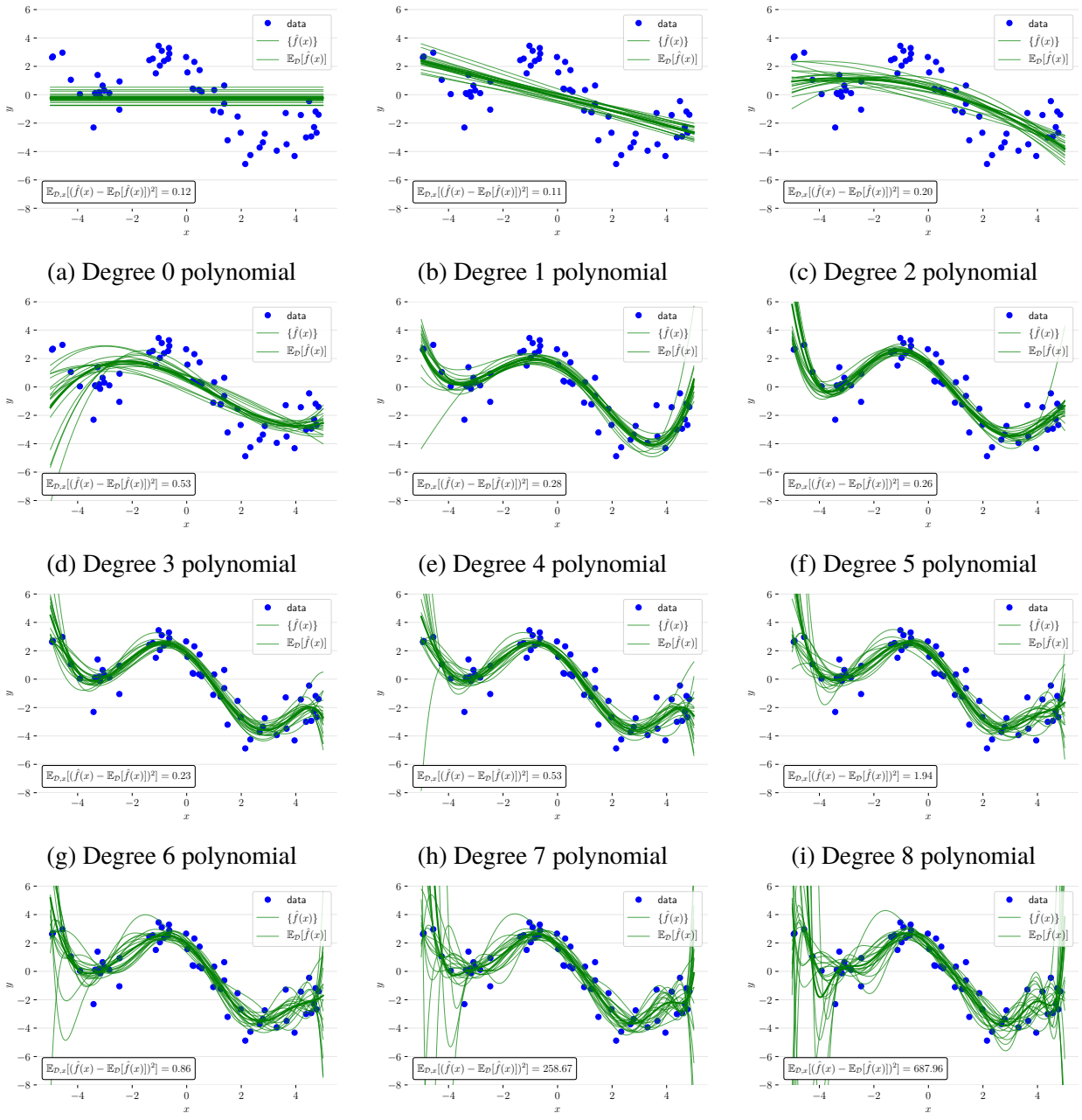
$$\begin{aligned}
\mathbb{E}_{\mathcal{D},x,y} \left[(y - \hat{f}(x))^2 \right] &= \mathbb{E}_{\mathcal{D},x,y} \left[y^2 - 2y\hat{f}(x) + \hat{f}(x)^2 \right] && (7)\\
&= \mathbb{E}_x \left[\mathbb{E}_y[y^2 \mid x] - 2\mathbb{E}_y[y \mid x]\, \mathbb{E}_{\mathcal{D}}[\hat{f}(x)] + \mathbb{E}_{\mathcal{D}}[\hat{f}(x)^2] \right] && (8)\\
&= \mathbb{E}_x \left[\text{Var}[y \mid x] + f(x)^2 - 2f(x)\, \mathbb{E}_{\mathcal{D}}[\hat{f}(x)] + \text{Var}[\hat{f}(x)] + \mathbb{E}_{\mathcal{D}}[\hat{f}(x)]^2 \right] && (9)\\
&= \sigma^2 + \mathbb{E}_x \left[\text{Var}[\hat{f}(x)] + f(x)^2 - 2f(x)\, \mathbb{E}_{\mathcal{D}}[\hat{f}(x)] + \mathbb{E}_{\mathcal{D}}[\hat{f}(x)]^2 \right] && (10)\\
&= \sigma^2 + \mathbb{E}_x \left[\text{Var}[\hat{f}(x)] + (f(x) - \mathbb{E}_{\mathcal{D}}[\hat{f}(x)])^2 \right] && (11)\\
&= \sigma^2 + \text{Var}[\hat{f}] + \text{Bias}^2[\hat{f}] . && (12)
\end{aligned}$$

This decomposition makes it clear that there really is no free lunch. If we make our models flexible, we can reduce bias but at the cost of increasing the variance. If make our models simpler, we might reduce variance but increase the expected gap between our fit and the truth. Figure 5 illustrates the effect empirically using the data and samples from Figures 3 and 4. The expected squared error is the sum of the squared bias, the variance, and the noise variance σ^2.

Regularization

The discussion of overfitting and bias-variance tradeoff so far has been focused on model complexity as expressed by increasing numbers of basis functions (and therefore increasing numbers of parameters). In practice, complexity management is often done by directly penalizing the parameters of the model, as referred to above as a preference bias. There are a wide variety of ways to do this, that vary from simple counts to highly structured probabilistic models based on knowledge of a physical system.

The most widely used regularization penalty, however, is to use an L^p norm on the parameters. A *norm* in a vector space like $\mathbb{R}^D$ is a generalized notion of the length of the vectors. We write the



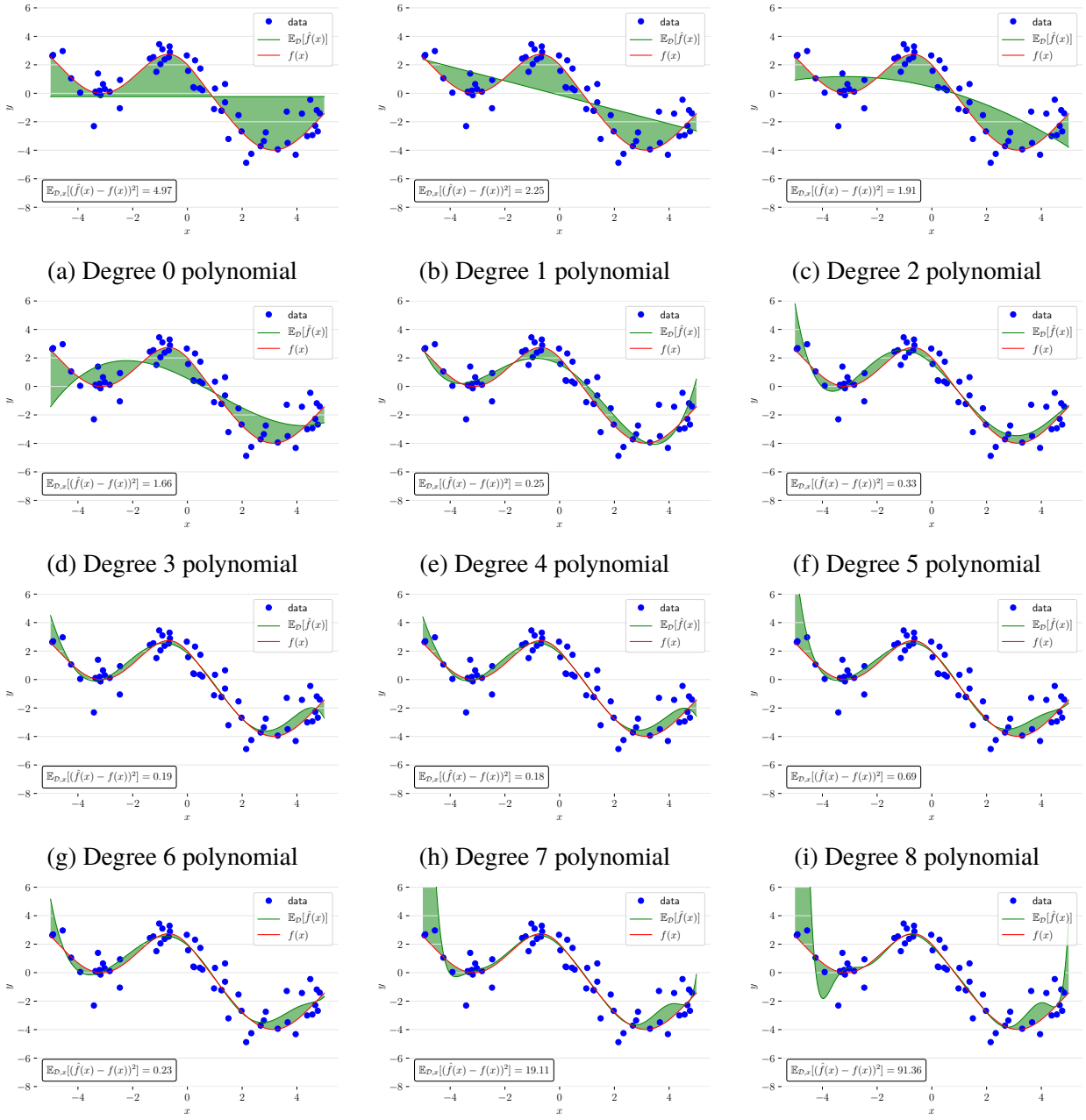
(a) Degree 0 polynomial (b) Degree 1 polynomial (c) Degree 2 polynomial

(d) Degree 3 polynomial (e) Degree 4 polynomial (f) Degree 5 polynomial

(g) Degree 6 polynomial (h) Degree 7 polynomial (i) Degree 8 polynomial

(j) Degree 9 polynomial (k) Degree 10 polynomial (l) Degree 11 polynomial

Figure 3: This collection of figures illustrates the concept of variance as it arises in the bias-variance tradeoff. These figures were produced by taking 60 data generated as in Figure 1, and then taking 20 random subsets of 30 data. For each of those 20 subsets, a line was fit using least squares, with the basis specified. Each of those lines is shown along with the line that is the pointwise sample mean — a Monte Carlo approximation to $\mathbb{E}_{\mathcal{D}}[\hat{f}(x)]$. In the bottom left of each figure is the estimated variance, which can be seen to explode for high degrees due to large fluctuations at the boundaries.



(a) Degree 0 polynomial (b) Degree 1 polynomial (c) Degree 2 polynomial

(d) Degree 3 polynomial (e) Degree 4 polynomial (f) Degree 5 polynomial

(g) Degree 6 polynomial (h) Degree 7 polynomial (i) Degree 8 polynomial

(j) Degree 9 polynomial (k) Degree 10 polynomial (l) Degree 11 polynomial

Figure 4: This sequence of figures illustrates the concept of bias as it applies to the bias-variance tradeoff. The mean $\hat{f}(x)$ from Figure 3 is plotted, along with the true $f(x)$ and the gap between them is shaded to show the magnitude of the difference. Note that as the model gets more flexible due to higher degree polynomials, the gap shrinks. In the bottom left of each figure is shown the Monte Carlo estimate of the squared bias. The reason the bias tends to go up again for larger degrees is because the variance causes large Monte Carlo error in the estimate of $\mathbb{E}_{\mathcal{D}}[\hat{f}(x)]$.

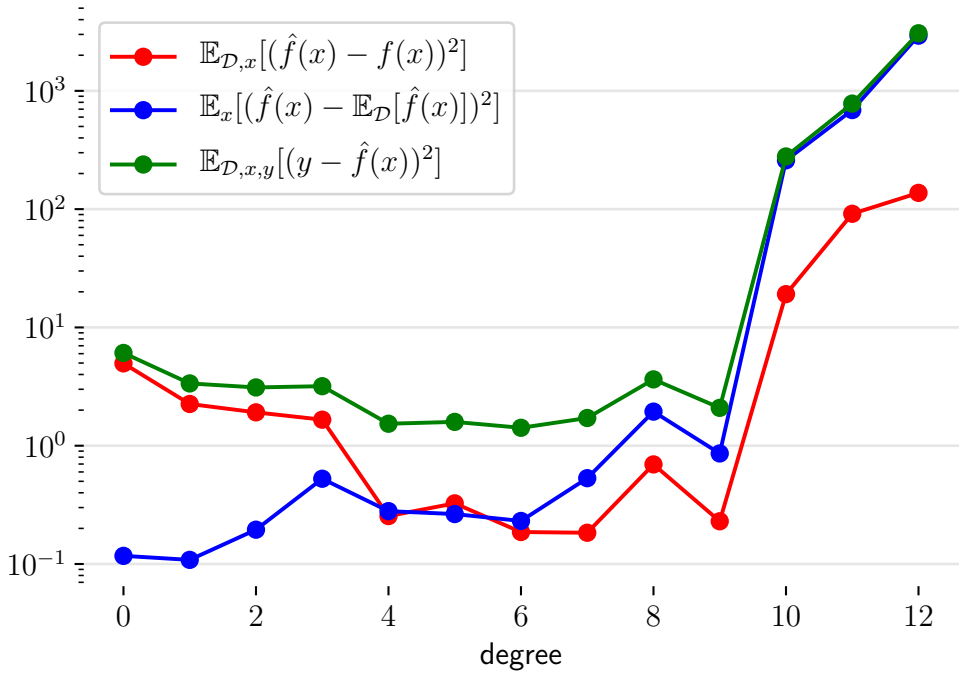


Figure 5: Combining the estimates from each subfigure in Figure 3 and Figure 5, we can plot the squared bias and variance as a function of degree, along with the expected squared error. The red and blue lines (plus $\sigma^2 = 1$) equal the green line. Generally we see an increase in variance and for low degrees a decrease in bias with increasing degree. The reason the bias tends to go up again for larger degrees is because the variance causes large Monte Carlo error in the estimate of $\mathbb{E}_{\mathcal{D}}[\hat{f}(x)]$.

p-norm of a vector $\boldsymbol{z}$ as

$$||\boldsymbol{z}||_p = \left(\sum_{d=1}^{D} |z_d|^p \right)^{1/p} . \tag{13}$$

For $p = 2$ we get the usual Euclidean norm and that is what we will study here. However, don't be surprised to encounter L^0, L^1, and L^∞ norms in different machine learning contexts out in the real world.

In the case of linear regression, having a *regularization penalty* such as an L^2 norm means literally penalizing the length of the vector $\boldsymbol{w}$ as part of the objective function. That is, we're saying that larger weights represent more extreme—and therefore complex—models and that we're willing to absorb some amount of additional training error in order to have smaller parameters. We of course have to decide just what that balance should be and so we introduce a new *hyperparameter* $\lambda > 0$ that governs it. For computational convenience, we often use a squared norm as the penalty:

$$L(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} (\boldsymbol{x}_n^\mathsf{T} \boldsymbol{w} - y_n)^2 + \frac{\lambda}{2} ||\boldsymbol{w}||_2^2 = \frac{1}{2N} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\mathsf{T} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{\lambda}{2} \boldsymbol{w}^\mathsf{T} \boldsymbol{w} . \tag{14}$$

This regularization penalty has many names. It is sometimes called *Tikhonov* regularization and sometimes *weight decay* for reasons that will be clearer when we discuss objectives that do not have closed-form solutions. When this squared L^2 norm is used in least-squares linear regression, the overall setup is often called *ridge regression*.

Finding the optimal weights with ridge regression is not any harder than the usual ordinary least squares setup, in fact it typically makes it easier because the additional term makes the linear system better conditioned. We proceed as before and take the gradient with respect to $\boldsymbol{w}$, set it to zero and solve for $\boldsymbol{w}$:

$$\nabla_{\boldsymbol{w}} \left\{ \frac{1}{N}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{\lambda}{2}\boldsymbol{w}^\top\boldsymbol{w} \right\} = \frac{1}{N}\boldsymbol{X}^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda\boldsymbol{w} = 0 \tag{15}$$

$$\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^\top\boldsymbol{y} + \lambda\boldsymbol{w} = 0 \tag{16}$$

$$(\boldsymbol{X}^\top\boldsymbol{X} + \lambda\mathbf{I})\boldsymbol{w} = \boldsymbol{X}^\top\boldsymbol{y} \tag{17}$$

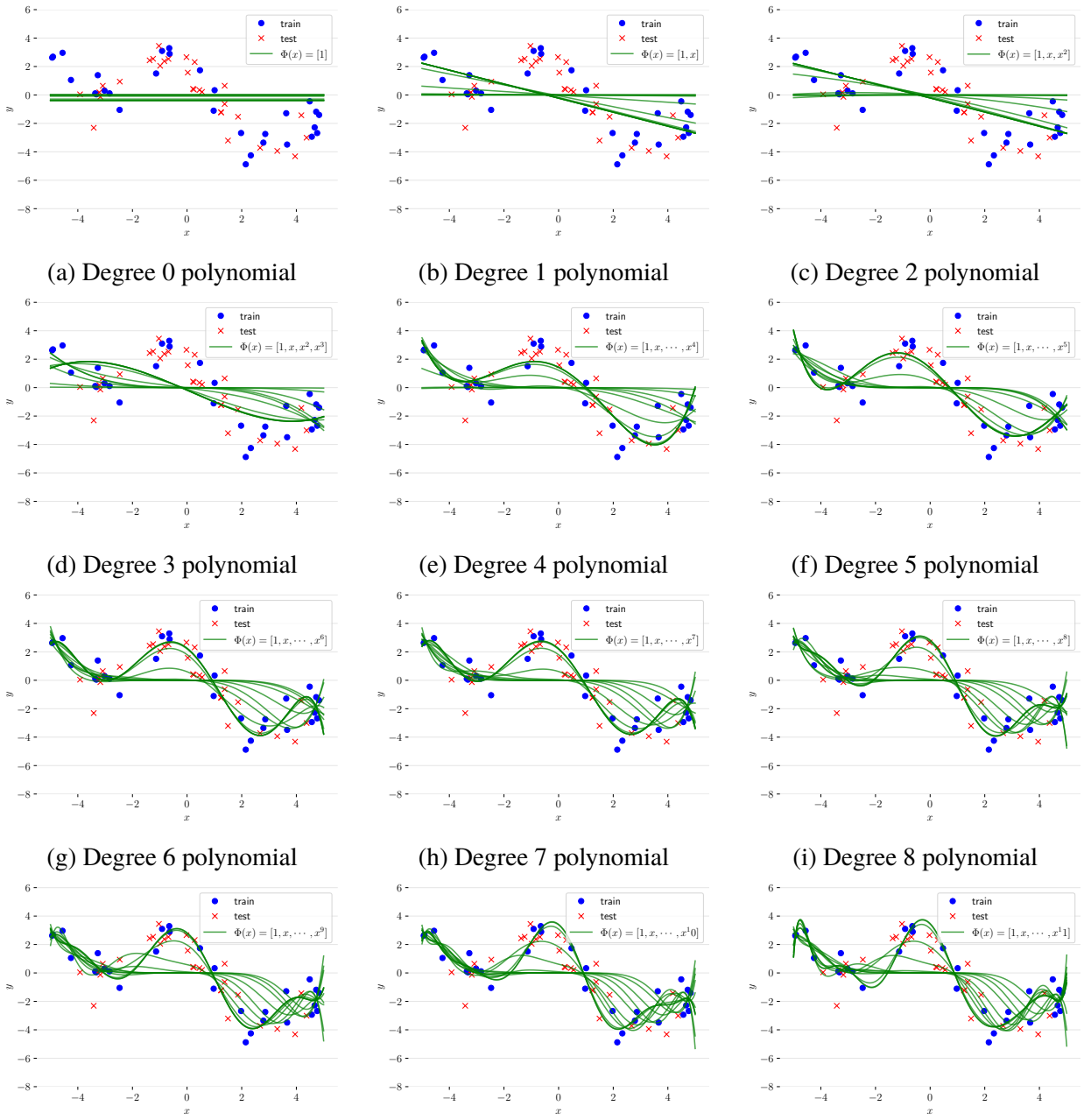
$$\boldsymbol{w}^\star = (\boldsymbol{X}^\top\boldsymbol{X} + \lambda\mathbf{I})^{-1}\boldsymbol{X}^\top\boldsymbol{y} \tag{18}$$

Figure 6 finds the optimal weights using this formula for different values of λ across polynomials of different degrees, all with the same data and setup as before. The value of λ is varied over a wide range, from 10^{-3} to 10^7 and different fits are shown. Figure 7 shows the training and test losses across these bases as a function of the penalty. As expected, the training loss is always worse because the penalty is moving the optimum toward zero. However, the test loss often improves up to some "sweet spot" where the regularization is helping prevent a model from overfitting. This effect is most dramatic with the higher-degree models that are most likely to overfit. For example, the 12-degree polynomial cuts its test loss almost in half with $\lambda = 1$.

Penalized Maximum Likelihood

Just as we can penalized our loss-based objective functions, we can also perform penalized maximum likelihood and recover essentially the same solution with Gaussian observations. There is an interesting interpretation of penalized maximum likelihood, however, that bears a brief discussion. If you have studied statistics or probability previously, you may have noticed that I write the likelihood function for regression as $\Pr(y \mid \boldsymbol{x}, \boldsymbol{w}, v)$ where by conditioning upon $\boldsymbol{w}$ and v I am elevating them to the status of random variables rather than simple parameters governing the likelihood. The reason for this is because in the penalized maximum likelihood case it provides a natural Bayesian interpretation. Instead of finding single estimates of the regression weights $\boldsymbol{w}^\star$, what if we examined the distribution induced over them using Bayes' theorem? This would require a prior distribution over the parameters $\boldsymbol{w}$, which we will I will denote here as $\Pr(\boldsymbol{w})$. Bayes' theorem is essentially an identity of the calculus of probability that tells us how to "invert" a generative model that went from parameters to data; it lets us go from data to parameters instead:

$$\Pr(\boldsymbol{w} \mid \{\boldsymbol{x}_n, y_n\}_{n=1}^N, v) = \frac{\Pr(\{y_n\}_{n=1}^N \mid \{\boldsymbol{x}_n\}_{n=1}^N, \boldsymbol{w}, v)\ \Pr(\boldsymbol{w})}{\Pr(\{y_n\}_{n=1}^N \mid \{\boldsymbol{x}_n\}_{n=1}^N, v)}, \tag{19}$$



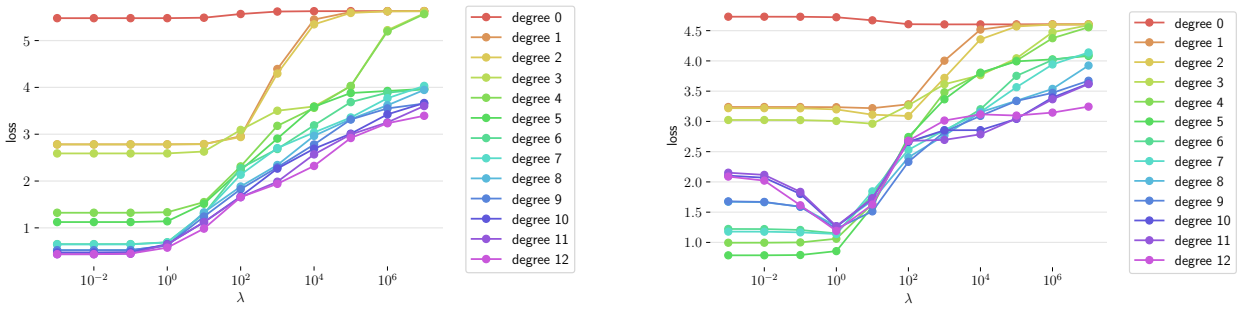
(a) Degree 0 polynomial (b) Degree 1 polynomial (c) Degree 2 polynomial

(d) Degree 3 polynomial (e) Degree 4 polynomial (f) Degree 5 polynomial

(g) Degree 6 polynomial (h) Degree 7 polynomial (i) Degree 8 polynomial

(j) Degree 9 polynomial (k) Degree 10 polynomial (l) Degree 11 polynomial

Figure 6: This sequence of figures illustrates the effect of a squared L^2 penalty on the different polynomial bases we have previously looked at in this note, with the same data and the same linear regression setup. In each of these figures there is a set of green lines corresponding to logarithmically increasing values of λ, i.e., $\lambda = 10^{-3}, 10^{-2}, \ldots, 10^{7}$. As the penalty gets larger, the weights shrink toward zero and the line becomes just $f(x) = 0$.



(a) Effect of L^2 penalty on training loss

(b) Effect of L^2 penalty on test loss

Figure 7: These figures show how the loss varies for the different polynomial regression setups, as a function of squared L^2 penalty on the weights. These are summaries of the fits shown in Figure 6. The penalty is varied over orders of magnitude. Note that it always makes training loss worse because it strictly reduces the ability of the parameters to fit the data. However, the test losses often improve for some regions of λ.

where the denominator $\Pr(\{y_n\}_{n=1}^N \mid \{\boldsymbol{x}_n\}_{n=1}^N, v)$ is a constant that does not depend on $\boldsymbol{w}$ and that we can ignore for now. As in the MLE linear regression case, we will use a Gaussian likelihood:

$$\Pr(\{y_n\}_{n=1}^N \mid \{\boldsymbol{x}_n\}_{n=1}^N, \boldsymbol{w}, v) = \Pr(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, v) = |2v\pi|^{-N/2} \exp\left\{-\frac{1}{2v}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\mathsf{T}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})\right\}. \quad (20)$$

With the prior we have flexibility much like we would when specifying a penalty. We have been thinking a lot about Gaussian distributions, so let's use a zero-mean Gaussian distribution for our prior, with covariance matrix $\frac{v}{\lambda}\mathbf{I}$. So:

$$\boldsymbol{w} \mid \lambda, v \sim \mathcal{N}(\boldsymbol{w} \mid 0, \frac{v}{\lambda}\mathbf{I}) \quad (21)$$

$$\Pr(\boldsymbol{w}) = \left(\frac{2\pi v}{\lambda}\right)^{-N/2} \exp\left\{-\frac{\lambda}{2v}\boldsymbol{w}^\mathsf{T}\boldsymbol{w}\right\}. \quad (22)$$

Now we have the tools we need to compute the posterior distribution over $\boldsymbol{w}$ given the observed data. There are various constants that don't depend on $\boldsymbol{w}$ (including the denominator of Bayes' theorem) and so we can ignore them by writing the bit that determines the proportionality:

$$\Pr(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}, v, \lambda) \propto \exp\left\{-\frac{1}{2v}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\mathsf{T}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) - \frac{\lambda}{2v}\boldsymbol{w}^\mathsf{T}\boldsymbol{w}\right\}. \quad (23)$$

In case it's unclear why I feel comfortable doing this, the reason is because I always know that probability distributions have to integrate to one. All I have to do is keep the factors that depend on $\boldsymbol{w}$ and I can always normalize it to get the constants back. This is a useful thing to remember so you don't have to keep track of terms like $(2\pi)^{-N/2}$.

One question I might want to ask about this posterior distribution over $\boldsymbol{w}$ is "what is its mode?" That is, under my posterior distribution, what is the most probable $\boldsymbol{w}$? When we use the mode

of a posterior as an estimate, we call it the *maximum a posteriori* or MAP estimate. If you spend much time on machine learning, you will hear about the MAP estimate many times even outside of a formally Bayesian context. Our method for finding the MAP estimate in this case looks exactly like it did when we maximized the likelihood: we take the log, compute the gradient, set it to zero, and then solve for $\boldsymbol{w}$.

$$\nabla_{\boldsymbol{w}} \log \Pr(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}, v, \lambda) = -\frac{1}{v}\boldsymbol{X}^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) - \frac{\lambda}{v}\boldsymbol{w} = 0 \tag{24}$$

$$\boldsymbol{X}^{\mathsf{T}}\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^{\mathsf{T}}\boldsymbol{y} + \lambda\boldsymbol{w} = 0 \tag{25}$$

$$(\boldsymbol{X}^{\mathsf{T}}\boldsymbol{X} + \lambda\mathbf{I})\boldsymbol{w} = \boldsymbol{X}^{\mathsf{T}}\boldsymbol{y} \tag{26}$$

$$\boldsymbol{w}^{\mathsf{MAP}} = (\boldsymbol{X}^{\mathsf{T}}\boldsymbol{X} + \lambda\mathbf{I})^{-1}\boldsymbol{X}^{\mathsf{T}}\boldsymbol{y} \tag{27}$$

This is exactly the same solution we got in Eqn. 18. Just like we realized when talking about MLE with Gaussian observations as being a probabilistic interpretation of least squares, we can see that MAP estimation with a zero-mean Gaussian prior is the same as introducing a squared L^2 penalty on the parameters.

Changelog

- 26 September 2018 – Initial version.
- 27 September 2018 – Fixed some typos in figures.
- 2 October 2018 – Should be precise and say “covariance matrix” rather than “covariance” when referring to the Gaussian distribution.