EECS 189 Introduction to Machine Learning
Fall 2020

Note 3

1 Feature Engineering

We've seen that the least-squares optimization problem

$$\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$$

represents the "best-fit" *linear* model, by projecting $\mathbf{y}$ onto the subspace spanned by the columns of $\mathbf{X}$. However, the true input-output relationship $y = f(\mathbf{x})$ may be nonlinear, so it is useful to consider nonlinear models as well. It turns out that we can still do this under the framework of linear least-squares, by augmenting the data with new **features**. In particular, we devise some function $\boldsymbol{\phi} : \mathbb{R}^\ell \to \mathbb{R}^d$, called a **feature map**, that maps each raw data point $\mathbf{x} \in \mathbb{R}^\ell$ into a vector of features $\boldsymbol{\phi}(\mathbf{x})$. The hypothesis function then writes

$$h_{\mathbf{w}}(\mathbf{x}) = \sum_{j=1}^{d} w_j \phi_j(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})$$

Note that the resulting model is still linear with respect to the features, but it is nonlinear with respect to the original data if $\boldsymbol{\phi}$ is nonlinear. The component functions ϕ_j are sometimes called **basis functions** because our hypothesis is a linear combination of them. In the simplest case, we could just use the components of $\mathbf{x}$ as features (i.e. $\phi_j(\mathbf{x}) = x_j$), but in general it is helpful to disambiguate the features of an example from the example's entries.

We can then use least-squares to estimate the weights $\mathbf{w}$, just as before. To do this, we replace the original data matrix $\mathbf{X} \in \mathbb{R}^{n\times\ell}$ by $\mathbf{\Phi} \in \mathbb{R}^{n\times d}$, which has $\boldsymbol{\phi}(\mathbf{x}_i)^\top$ as its ith row:

$$\min_{\mathbf{w}} \|\mathbf{\Phi}\mathbf{w} - \mathbf{y}\|_2^2$$

1.1 Example: Fitting Ellipses

Let's use least-squares to estimate the parameters of an ellipse from data.

Assume that we have n data points $\mathcal{D} = \{(x_{1,i}, x_{2,i})\}_{i=1}^n$, which may be noisy (i.e. could be off the actual orbit). Our goal is to determine the relationship between x_1 and x_2.

We assume that the ellipse from which the points were generated has the form

$$w_1 x_1^2 + w_2 x_2^2 + w_3 x_1 x_2 + w_4 x_1 + w_5 x_2 = 1$$

where the coefficients $w_1, \dots, w_5$ are the parameters we wish to estimate.

We formulate the problem with least-squares:

$$\min_{\mathbf{w}} \|\mathbf{\Phi}\mathbf{w} - \mathbf{1}\|_2^2$$

where

$$\boldsymbol{\Phi} = \begin{bmatrix} x_{1,1}^2 & x_{2,1}^2 & x_{1,1}x_{2,1} & x_{1,1} & x_{2,1} & \\ x_{1,2}^2 & x_{2,2}^2 & x_{1,2}x_{2,2} & x_{1,2} & x_{2,2} & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{1,n}^2 & x_{2,n}^2 & x_{1,n}x_{2,n} & x_{1,n} & x_{2,n} & \end{bmatrix}$$

In this case, the feature map $\boldsymbol{\phi}$ is given by

$$\boldsymbol{\phi}(\mathbf{x}) = (x_1^2, x_2^2, x_1x_2, x_1, x_2)$$

Note that there is no "target" vector $\mathbf{y}$ here, so this is not a traditional regression problem, but it still fits into the framework of least-squares.

1.2 Polynomial Features

The example above demonstrates an important class of features known as **polynomial features**. Remember that a polynomial is linear combination of monomial basis terms. Monomials can be classified in two ways, by their degree and dimension:

Degree / Dimension	0	1	2	3	$\cdots$
1 (univariate)	1	x	x^2	x^3	$\cdots$
2 (bivariate)	1	x_1, x_2	x_1^2, x_2^2, x_1x_2	$x_1^3, x_2^3, x_1^2x_2, x_1x_2^2$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

A big reason we care polynomial features is that any smooth function can be approximated arbitrarily closely by some polynomial.[1] For this reason, polynomials are said to be **universal approximators**.

One downside of polynomials is that as their degree increases, their number of terms increases rapidly. Specifically, one can use a "stars and bars" style combinatorial argument[2] to show that a polynomial of degree d in ℓ variables has

$$\binom{\ell + d}{\ell} = \frac{(\ell + d)!}{\ell! d!}$$

terms. To get an idea for how quickly this quantity grows, consider a few examples:

[1] **Taylor's theorem** gives more precise statements about the approximation error.

[2] We count the number of distinct monomials of degree at most d in ℓ variables $x_1, \ldots, x_\ell$, or equivalently, the number of distinct monomials of degree exactly d in $\ell + 1$ variables $x_0 = 1, x_1 \ldots, x_\ell$. Every monomial has the form $x_0^{k_0} \ldots x_\ell^{k_\ell}$ where $k_0 + \cdots + k_\ell = d$. This corresponds to an arrangement of d stars and ℓ bars, where the number of stars between consecutive bars (or the ends of the expression) gives the degree of that ordered variable. For example,

$$*|***|** \quad \leftrightarrow \quad x_0^1 x_1^3 x_2^2$$

The number of unique ways to arrange these stars and bars is the number of ways to choose the positions of the ℓ bars out of the total $\ell + d$ slots, i.e. $\ell + d$ choose ℓ. (You could also pick the positions of the d stars out of the total $\ell + d$ slots; the expression is symmetric in ℓ and d.)

ℓ \ d	1	3	5	10	25
1	2	4	6	11	26
3	4	20	56	286	3276
5	6	56	252	3003	142506
10	11	286	3003	184756	183579396
25	26	3276	142506	183579396	126410606437752

Later we will learn about the **kernel trick**, a clever mathematical method that allows us to circumvent this rapidly growing cost in certain cases.

2 Hyperparameters and Validation

As above, consider a hypothesis of the form

$$h_{\mathbf{w}}(\mathbf{x}) = \sum_{j=1}^{d} w_j \phi_j(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})$$

Observe that the model order d is not one of the decision variables being optimized when we fit to the data. For this reason d is called a **hyperparameter**. We might say more specifically that it is a **model hyperparameter**, since it determines the structure of the model.

For another example, recall **ridge regression**, in which we add an ℓ^2 penalty on the parameters $\mathbf{w}$:

$$\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_2^2$$

The regularization weight λ is also a hyperparameter, as it is fixed during the minimization above. However λ, unlike the previously discussed hyperparameter d, is not a part of the model. Rather, it is an aspect of the optimization procedure used to fit the model, so we say it is an **optimization hyperparameter**. Hyperparameters tend to fall into one of these two categories.

Since hyperparameters are not determined by the data-fitting optimization procedure, how should we choose their values? A suitable answer to this question requires some discussion of the different types of error at play.

2.1 Types of Error

We have seen that it is common to minimize some measure of how poorly our hypothesis fits the data we have, but what we actually care about is how well the hypothesis predicts *future* data. Let us try to formally distinguish the various types of error. Assume that the data are distributed according to some (unknown) distribution $\mathcal{D}$, and that we have a **loss function** $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, which is to measure the error between the true output y and our estimate $\hat{y} = h(\mathbf{x})$. The **risk** (or **true error**) of a particular hypothesis $h \in \mathcal{H}$ is the expected loss over the whole data distribution:

$$R(h) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\ell(h(\mathbf{x}), y)]$$

Ideally, we would find the hypothesis that minimizes the risk, i.e.

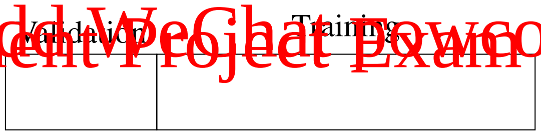
$$h^* = \arg\min_{h \in \mathcal{H}} R(h)$$

However, computing this expectation is impossible because we do not have access to the true data distribution. Rather, we have access to samples $(\mathbf{x}_i, y_i) \overset{\text{iid}}{\sim} \mathcal{D}$. These enable us to approximate the real problem we care about by minimizing the **empirical risk** (or **training error**)

$$\hat{R}_{\text{TRAIN}}(h) = \frac{1}{n} \sum_{i=1}^{n} \ell(h(\mathbf{x}_i), y_i)$$

But since we have a finite number of samples, the hypothesis that performs the best on the training data is not necessarily the best on the whole data distribution. In particular, if we both train and evaluate the hypothesis using the same data points, the training error will be a very biased estimate of the true error, since the hypothesis has been chosen specifically to perform well on those points. This phenomenon is sometimes referred to as "data incest".

A common solution is to set aside some portion (say 30%) of the data, to be called the **validation set**, which is disjoint from the training set and *not* allowed to be used when fitting the model:



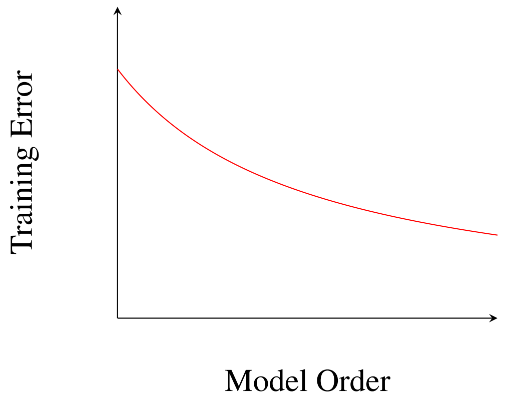
We can use this validation set to estimate the true error by the **validation error**

$$\hat{R}_{\text{VAL}}(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(h(\mathbf{x}_i^{\text{val}}), y_i^{\text{val}})$$

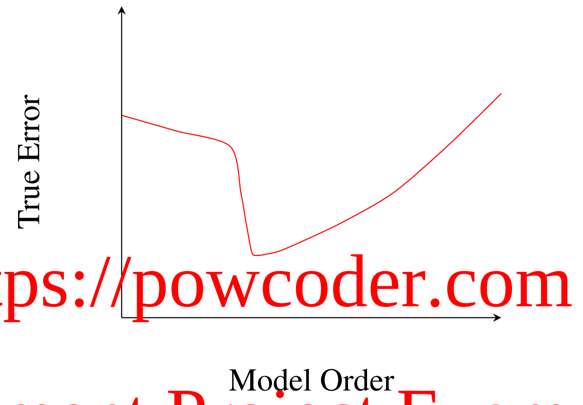
With this estimate, we have a simple method for choosing hyperparameter values: try a bunch of configurations of the hyperparameters and choose the one that yields the lowest validation error.

2.2 The effect of hyperparameters on error

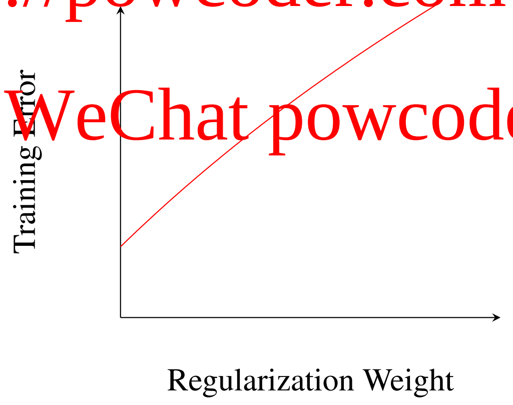
Note that as we add more features to a linear model, training error can only decrease. This is because the optimizer can set $w_i = 0$ if feature i cannot be used to reduce training error.



Adding more features tends to reduce true error as long as the additional features are useful predictors of the output. However, if we keep adding features, these begin to fit noise in the training data instead of the true signal, causing true error to actually increase. This phenomenon is known as **overfitting**.



The validation error tracks the true error reasonably well as long as the validation set is sufficiently large. The regularization hyperparameter λ has a somewhat different effect on training error. Observe that if $\lambda = 0$, we recover the exact OLS problem, which is directly minimizing the training error. As λ increases, the optimizer places less emphasis on the training error and more emphasis on reducing the magnitude of the parameters. This leads to a degradation in training error as λ grows:



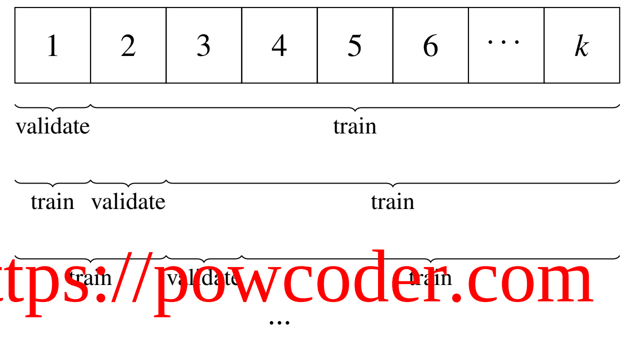
2.3 Cross-validation

Setting aside a validation set works well, but comes at a cost, since we cannot use the validation data for training. Since having more data generally improves the quality of the trained model, we may prefer not to let that data go to waste, especially if we have little data to begin with and/or collecting more data is expensive. Cross-validation is an alternative to having a dedicated validation set.

k-fold cross-validation works as follows:

1. Shuffle the data and partition it into k equally-sized (or as equal as possible) blocks.

2. For $i = 1, \ldots, k$,
   - Train the model on all the data except block i.
   - Evaluate the model (i.e. compute the validation error) using block i.



3. Average the k validation errors; this is our final estimate of the true error.

Observe that, although every datapoint is used for evaluation at some time or another, the model is always evaluated on a different set of points than it was trained on, thereby cleverly avoiding the "data incest" problem mentioned earlier.

Note also that this process (except for the shuffling and partitioning) must be repeated for every hyperparameter configuration we wish to test. This is the principle drawback of k-fold cross-validation as compared to using a held-out validation set – there is roughly k times as much computation required. This is not a big deal for the relatively small linear models that we've seen so far, but it can be prohibitively expensive when the model takes a long time to train, as is the case in the Big Data regime or when using neural networks.