Validation and Testing

October 12, 2022

After looking at some specific predictors (polynomials, nearest-neighbor predictors, linear predictors), we return to a more general discussion pertaining to the evaluation of the performance of machine learning algorithms. Given this broadening of scope, it may be useful to summarize first some of the concepts we have encountered so far.

1 Summary of Relevant Concepts

This Section collects in one place concepts and definitions concerning (i) training as empirical risk minimization, (ii) the difference between data fitting and machine learning, and (iii) the concept of validation for the estimation of hyper-parameters.

Training as Empirical Risk Minimization The predictors we have encountered so far are trained through data fitting: Define the loss that a predictor incurs on the training data, and find a predictor that minimizes the average loss (risk) on the training set. To summarize, when a predictor h predicts value $h(\mathbf{x})$ for a data point $\mathbf{x}$ and the true value associated with $\mathbf{x}$ is y, we experience a *loss* $\ell(y, h(\mathbf{x}))$. The loss is zero when $h(\mathbf{x}) = y$ and positive otherwise. The loss function has signature

$$\ell \ : \ Y \times Y \to \mathbb{R}_0^+$$

where $\mathbb{R}_0^+$ is the set of nonnegative real numbers. We have seen some possible definitions of loss for different problems, for either classification or regression. For instance, the *zero-one loss* is very commonly used for classification:

$$\ell(y, h(\mathbf{x})) = \begin{cases} 0 & \text{if } h(\mathbf{x}) = y \\ 1 & \text{otherwise} \end{cases}$$

and so is the cross-entropy loss for score-based classifiers

$$\ell(y, \mathbf{p}) = -\log p_y \ ,$$

where $\mathbf{p} = [p_1, \ldots, p_K]^T$ is a vector that scores each of the K classes with a real value $p_k \in [0, 1]$. For regression, the *square loss* is a popular choice:

$$\ell(y, h(\mathbf{x})) = (h(\mathbf{x}) - y)^2 \ .$$

The *empirical risk* over the training set T of data point/value pairs is the average loss over that set:[1]

$$L_T(h) \stackrel{\text{def}}{=} \frac{1}{|T|} \sum_{(\mathbf{x},y)\in T} \ell(y, h(\mathbf{x})) \ .$$

Given a training set T and a hypothesis space $\mathcal{H}$, *Empirical Risk Minimization* (ERM) is the problem of finding a predictor in $\mathcal{H}$ with the lowest empirical risk[2] on T:

$$\text{ERM}_T(\mathcal{H}) \in \arg\min_{h\in\mathcal{H}} L_T(h) \ .$$

> **Notation:** This notation should be familiar by now. To recall, the quantity $\min_{h\in\mathcal{H}} L_T(h)$ is a single (non-negative) number, the smallest achievable risk over all choices of h in $\mathcal{H}$. The notation $\arg\min_{h\in\mathcal{H}} L_T(h)$ represents the *set* of all functions in $\mathcal{H}$ that achieve that minimal risk (there could be more than one such function). Finally, $\text{ERM}_T(\mathcal{H})$ is one of these functions, it does not matter which.

ERM is a *fitting* problem, as it seeks a function h that best fits the data in T.

Generalization What is the point of fitting a predictor to a set of data if we only look at how well the predictor does on that data, as the risk L_T does? A predictor estimates a value $\hat{y}$ from a data point $\mathbf{x}$, but we already know the correct answer y: What is the point of estimating the answer again?

This question goes to the key difference between data fitting and machine learning: Data fitting is asked to do well on the training set, and is used when something about the predictor itself is of interest. Perhaps an economist postulates that price and demand are linearly correlated, and wants to determine the regression coefficients. The goal of such a study is not so much to predict new prices for new levels of demand, but rather to use the slope of the regression line to understand the effects of a demand fluctuation on changes of price.

In contrast, machine learning focuses on prediction, and is asked to do well *on previously unseen data*. The parameters of the predictor are not of interest *per se*, and the goal is instead to estimate values $\hat{y}$ corresponding to new data points $\mathbf{x}$. A predictor that does this well is said to *generalize well*. What can we say about new data points, that is, data we have not yet seen? Specifically, how can we formalize the notion of "doing well on previously unseen data?" Since prediction is the goal, rather than fitting, the residual empirical risk $L_T(\hat{h})$ is not a good measure of performance. How else can we measure prediction performance? Part of this note focuses on this question.

Optimizing the Hyper-Parameters A closely related issue is how to determine the hyper-parameter(s) of a predictor, if any. So far we have encountered two predictors whose hypothesis space $\mathcal{H}$ is a function of some hyper-parameter. Specifically, the maximum degree k of the fitting polynomial is the hyper-parameter of a polynomial regressor, and the hyper-parameter of the k-Nearest-Neighbor (k-NN) predictor is the number k of neighbors to consider. How can we compute a good value for k in either case?

A first idea is to incorporate k into the optimization performed for training the predictor, that is, optimize jointly over parameters and hyper-parameters. This approach would eliminate the

[1] We will look at empirical risks over sets other than the training set T. In view of this greater generality, we denote the size of set T as $|T|$ rather than N.

[2] We are assuming that the minimum in this expression exists. Discussion of this assumption is beyond the scope of these notes.

distinction between parameters and hyper-parameters. However, there are two difficulties in doing so. The first one is technical: For nearest-neighbor, there is no training. For polynomials, the coefficients are real-valued, while k is a positive integer. Because of this, we cannot simply use a descent method, as the gradient of the loss function with respect to k is undefined. For either scenario, one needs to rethink some of the techniques. However, this issue is minor: For both problems, we could just try values of $k = 1, 2, \ldots$ and look for one that yields good results.

The second difficulty, on the other hand, is fundamental and unsurmountable: *The answer from optimization would be trivial and useless.* Specifically, for k-NN, setting $k = 1$ gives zero risk on T, as every data point $\mathbf{x}_n$ is closest to itself. Thus, the value of k that minimizes $L_T(h)$ is 1, and this cannot possibly be the answer we are looking for. For polynomials, we know that we can interpolate $k + 1$ points exactly with a polynomial of degree k, so the bigger k, the better, up to a limit: Just set $k = N - 1$, where N is the size of the training set, and the resulting training error will be, again, exactly zero.

In other words, for both k-NN and polynomial fitting we can drive $L_T(h)$ to zero with a suitable choice of hyper-parameter, but this does not tell us anything about how well we will do *on data we have not yet seen.* Again, the ERM risk is not adequate for determining hyper-parameters.

Not all types of predictor face this quandary. For instance, a linear predictor has no hyper-parameters,[3] and we can achieve $L_T(h) = 0$ only when T happens to be linearly separable (for a linear classifier) or when the training samples $(\mathbf{x}_n, y_n)$ happen to be on a hyperplane (for a linear regressor). Whether the data complies with these assumptions or not, the smallest possible (and typically nonzero) $L_T(h)$ says little on how well the predictor does on previously unseen data.

Thus, we face at least three broad problems, which we discuss in the sections that follow:

1. How to think about "previously unseen data." The answer is probabilistic, in the form of what is called a *generative data model* (Section 2).

2. How to choose good hyper-parameters, if $\mathcal{H}$ has any. This problem is called *model selection*, and a common technique for it is called *validation* (Section 3).

3. How to evaluate the generalization performance of a predictor. This is called *testing* (Section 4).

2 A Generative Data Model

A predictor has no chance to do well on data that is different from the data it was trained on unless the training data bears some resemblance or connection to the new data. For instance, it would not be fruitful to train a predictor on human faces and evaluate its performance on handwritten digits, or even on cat muzzles. Performance is likely to be very bad in those cases. More subtly, if the human faces in the training set are all frontal snapshots and the resulting classifier is tested on profiles of human heads, we cannot hope for the classifier to do well. Generalization performance may be poor even if the test images are of the same type as the training images, but perhaps taken under systematically different lighting. We need some way to formalize the notion of "resemblance or connection" between two data sets.

[3]Unless regularization is used. To make training stable and improve generalization, a *regularization term* $\lambda\|\mathbf{v}\|^2$ is added to the training risk, where $\mathbf{v}$ is the vector of parameters. We have not discussed this aspect in class. The regularization coefficient λ is a hyper-parameter.

A probabilistic formulation of machine learning provides the conceptual link between "seen" and "unseen" data. Specifically, one assumes that all samples $(\mathbf{x}, y)$, both those in the training set and those seen after training, are drawn independently and at random from some joint probability distribution $p(\mathbf{x}, y)$ called the *generative model of the data*, or "model" for short. The model is the link between training and testing data. Of course, when we deploy a trained predictor we only see the data point $\mathbf{x}$, and it is the predictor's job to guess the corresponding target $h(\mathbf{x})$. However, we can think of the true target y as being "out there" even then: We just don't have access to it. Thus, it makes mathematical sense to ask what the loss $\ell(y, h(\mathbf{x}))$ is even during deployment, although we may not be able to compute it. In this spirit, the goal of machine learning is not to minimize the empirical risk, bur rather the *risk*:

$$L_p(h) = \mathbb{E}_p[\ell(y, h(\mathbf{x}))] \ ,$$

that is, the statistical expectation of the loss over the model p. This is a measure of how poorly h is expected to do not just over the training set, but rather over all possible data drawn from the model. Sometimes, the risk is called *statistical risk* for emphasis, when it is necessary to distinguish it from the empirical risk. The problem of machine learning is then (Statistical) Risk Minimization, that is, the problem of finding

$$\mathrm{RM}_p(\mathcal{H}) \in \arg\min_{h \in \mathcal{H}} L_p(h) \ ,$$

a much taller order than ERM. The predictor $\mathrm{RM}_p(\mathcal{H})$ is an *optimal predictor in* $\mathcal{H}$ and the risk it achieves is the *lowest risk on* $\mathcal{H}$, denoted by

$$L_p(\mathcal{H}) \stackrel{\text{def}}{=} \min_{h \in \mathcal{H}} L_p(h) \ .$$

> **Notation:** Here and elsewhere, the subscript for risks and predictors (that is for L, ERM, and RM) denotes what the risk is computed on, so the notational distinction between ERM and RM is redundant: A risk computed on a data set is always an empirical average, and a risk computed over a probability distribution is always a statistical mean. Nonetheless, this distinction is maintained for both emphasis and consistency with the literature. The term in parentheses denotes either a predictor (h) or a hypothesis space ($\mathcal{H}$). In the latter case, the risk is the smallest over all predictors in that hypothesis space. Thus, $L_p(\mathcal{H})$ is a minimum over a set, while $L_p(h)$ is not.

If the optimal predictor $\mathrm{RM}_p(\mathcal{H})$ in $\mathcal{H}$ does well (on average, on all possible data), then it does well also on the training set T only to the extent that T is a statistically representative sample of the data model p. If this is not the case, then there is no immediate relation between the performance of $\mathrm{RM}_p(\mathcal{H})$ and that of $\mathrm{ERM}_T(\mathcal{H})$: The training set T may just be a "fluke," that is, have low probability in p, so how well we do on T may not say much about how we do on other data drawn from p.

Unfortunately, T being a "fluke" is rather common: The data points $\mathbf{x}$ often represent complex objects (images or emails or web pages, for instance) and therefore live in a data space $X \subseteq \mathbb{R}^d$ with many dimensions ($d \gg 1$). The curse of dimensionality then implies that it is effectively impossible for the training set T to sample any significant part of X to any reasonable extent. Machine learning is quite often starved of data.

For later reference, we define a hypothesis space $\mathcal{H}$ to be *realizable* over the model p if zero statistical risk can be achieved in $\mathcal{H}$, that is, if $L_p(\mathcal{H}) = 0$. This terminology is a bit awkward. What is really realizable is the choice of a predictor from $\mathcal{H}$ that performs perfectly. However, it

would be more cumbersome to say "the choice of a perfect predictor from $\mathcal{H}$ is realizable" than to say "$\mathcal{H}$ is realizable."

The Model is Unknown If the empirical risk $L_T(h)$ on the training set T is not a good indicator of performance, why not just use as indicator the lowest statistical risk $L_p(\mathcal{H})$ on the hypothesis space $\mathcal{H}$ according to the model p? Given a data model p and a hypothesis space $\mathcal{H}$, we could just find an optimal predictor on $\mathcal{H}$ and its corresponding statistical risk, and we are done.

The fly in the ointment is that *we typically do not know the data model p*: What is, for instance, the probability distribution over the set of all possible images? Or over all possible English sentences? All we usually have is data, and it may well be that the training set T is all the data we have. Even if we have large amounts of data, the curse of dimensionality suggests, as noted earlier, that estimating probability distributions over spaces with many dimensions is hopeless. Thus, for all but the simplest cases, *the optimal predictor $RM_p(\mathcal{H})$ is beyond computing.*

A deeper question then arises: If we cannot know the data model p, what use is there in introducing it? There are several answers to this question.

- While $\mathrm{RM}_p(\mathcal{H})$ itself may be beyond reach, we can try and *estimate* it. There will be some uncertainty around the estimate, but that's better than nothing. For the estimate we need data, but we know how to use data to estimate statistical means—and risks are means—even if we don't know the underlying distribution.

- We may be able to *bound* some of these quantities. For instance, can we compute two numbers that bound the lowest risk $L_p(\mathcal{H})$ on a given hypothesis space $\mathcal{H}$ from above and from below, and *over all possible choices of model p*? To do so, we don't need to know a particular p, but just how to marginalize over all of them.

- Perhaps most importantly, it is hard to find a way to link training and testing data that is conceptually cleaner and simpler than introducing p. In some sense, assuming a probabilistic model is conceptually the right thing to do, and we can think of p as the holy grail we only know of indirectly and keep referring to.

A useful mental picture of the model p is that of an oracle that gives out samples from $X \times Y$ upon request. The oracle knows the distribution, but we do not. In addition, samples do not come for free, and a larger set of samples comes at a higher price than a small one. While the actual price is not part of the model, this cost consideration reflects the reality that collecting data points, especially with their targets, is laborious and potentially expensive.

> **Practical Aspects: Making a Training Set.** To get a sense of the difficulty of collecting training samples, it may be useful to examine what it takes to produce the training images $\mathbf{x}_n$ for the task of recognizing handwritten digits described in an earlier note.
>
> One could place a camera above one of the conveyer belts that move envelopes around in a USPS distribution center, and take a picture of each envelope. That picture, however, contains much more than a digit, and someone must therefore sit at a computer, display each envelope image on the screen in turn, drag a bounding box around each digit with a mouse, and save each cropped digit as a separate file. Automatic methods won't do: How can we find the digits on the envelope automatically if we haven't yet trained a system that knows how to recognize a digit?
>
> Another file will list the names (file names) of all the images and the corresponding labels (what digit each image represents). This collection of data (images with bounding boxes and labels) is the training set T, and associating a label to every training image is called *annotation* or *labeling*.

Different images may have different quality depending on factors such as lighting, the ink used to write the digit, the color of the envelope paper, the size of the digit, and so forth. In addition, the digit images may be cropped inconsistently if the operator gets tired, or multiple operators are employed to crop and label images. These inconsistencies often make learning harder, and training sets are sometimes *curated* because of this. In the USPS example, curation involves running manual procedures to make the images more uniform in terms of the factors mentioned above.[4] Curation helps machine learning research by making the problem easier for initial study. However, curation cannot be used to train real-life machine learning classifiers, because the whole point of computing h is to eliminate manual intervention during deployment (testing). In these cases, only automatic *preprocessing* of the images is a viable option to make the digits in them easier to recognize, because the preprocessing can then be applied during both training and testing. Preprocessing techniques are beyond the scope of these notes.

In the days of Google Images and web crawlers, *collecting* data for training is often inexpensive if no curation or preprocessing is needed or can be performed by automatic means. Annotation, on the other hand, involves humans and is laborious, time-consuming, and error-prone. A common method for annotating large amounts of data is to publish it to the Amazon Mechanical Turk, an online marketplace for the type of repetitive work involved in labeling. People around the world access the marketplace searching for easy jobs, and you pay them a small amount (often 1-3 cents) per label. Even so, labels are not always of good quality, and various schemes are devised to address this issue. For instance, multiple people could be employed to label each training sample, and a majority vote could then be taken to determine the most likely label. In any case, the cost of labeling is high, and many machine learning algorithms require large amounts of data. This high cost is arguably the main hurdle to a broader use of machine learning.

3 Model Selection

If the hypothesis space $\mathcal{H}$ depends on one or more hyper-parameters, a method is needed to select these. The problem of finding good hyper-parameters is called *model selection*, and we saw that optimizing hyper-parameters over the training set is not an option for accomplishing this task. A very popular technique for model selection is called *validation*, and uses a data set V of the same structure as T (datapoint/value pairs), but disjoint from T. The set V is called the *validation set*.

When the set of possible hyper-parameters is finite, validation can be understood as a nested optimization. Specifically, let Π be a set of possible hyper-parameter vectors. If $\boldsymbol{\pi}$ is a vector of hyper-parameters out of Π (a vector because there may be more than one hyper-parameter), let $\mathcal{H}_{\boldsymbol{\pi}}$ be the corresponding hypothesis space. Then, validation computes

$$\hat{\boldsymbol{\pi}} = \arg\min_{\boldsymbol{\pi}\in\Pi} L_V(\mathrm{ERM}_T(\mathcal{H}_{\boldsymbol{\pi}})) \ .$$

Notation: This formula is quite a mouthful, but the idea is simple, and can be understood by unpacking this expression from the inside-out: Pick each vector $\boldsymbol{\pi}$ of hyper-parameters out of Π in turn. The choice of $\boldsymbol{\pi}$ identifies a particular hypothesis space $\mathcal{H}_{\boldsymbol{\pi}}$. Search this hypothesis space (by a descent method or whatever other technique is appropriate for the given type of predictor) for an optimal estimator

$$\hat{h}_{\boldsymbol{\pi}} \in \mathrm{ERM}_T(\mathcal{H}_{\boldsymbol{\pi}})$$

on the *training* set T. Once $\hat{h}_{\boldsymbol{\pi}}$ is found, compute its empirical risk $L_V(\hat{h}_{\boldsymbol{\pi}})$ on the *validation* set V. When done performing this computation over all choices $\boldsymbol{\pi}$ out of Π, return the hyper-parameter vector $\hat{\boldsymbol{\pi}}$ that

[4] Think calibrating, touching up, and re-cropping or resizing images in Photoshop.

has the smallest empirical risk on V. Another way of viewing this validation procedure is through a loop, as shown in Algorithm 1.

Algorithm 1 Model Selection by Validation

procedure VALIDATION($\mathcal{H}, \Pi, T, V, \ell$)
  $\hat{L} = \infty$ ▷ Stores the best risk so far on V
  for $\boldsymbol{\pi} \in \Pi$ **do**
    $h \in \arg\min_{h' \in \mathcal{H}_{\boldsymbol{\pi}}} L_T(h')$ ▷ Use loss ℓ to compute best predictor $\mathrm{ERM}_T(\mathcal{H}_{\boldsymbol{\pi}})$ on T
    $L = L_V(h)$ ▷ Use loss ℓ to evaluate the predictor's risk on V
    if $L < \hat{L}$ **then**
      $(\hat{\boldsymbol{\pi}}, \hat{h}, \hat{L}) = (\boldsymbol{\pi}, h, L)$ ▷ Keep track of the best hyper-parameters, predictor, and risk
    end if
  end for
  return $(\hat{\boldsymbol{\pi}}, \hat{h}, \hat{L})$ ▷ Return best hyper-parameters, predictor, and risk estimate
end procedure

Logically, since the goal of validation is to select a good set of hyper-parameters, the validation procedure only needs to return $\hat{\boldsymbol{\pi}}$, the best set it found. Practically, however, it would be wasteful to then train a predictor on $\mathcal{H}_{\hat{\boldsymbol{\pi}}}$ again, since that predictor was trained during the procedure. Because of this, Algorithm 1 also returns the optimal predictor on the optimal hypothesis space. The validation risk is also returned for informational purposes.

A complication arises when the set Π of possible hyper-parameters is not finite. Even so, this set is often discrete. The validation procedure then scans Π in some order and finds the first *local* optimum, that is, a vector $\hat{\boldsymbol{\pi}}$ whose validation risk $L_V(\hat{h}_{\boldsymbol{\pi}})$ is smaller than previously encountered risk values, and such that subsequent risks are higher. This idea assumes that the dependence of $\mathcal{H}_{\boldsymbol{\pi}}$ on $\boldsymbol{\pi}$ is somehow simple and regular. When Π is not even countable, a grid of samples in it is often scanned in search of a good (but not necessarily optimal) set of hyper-parameters.

An example of validation with a discrete but infinite set Π is the choice of polynomial degree in polynomial regression or of number of neighbors in k-NN prediction. In these cases, Π is scanned in the order $k = 1, 2, \ldots$, and the value $\hat{k}$ corresponding to the first (significant) local minimum in the validation risk sequence $L_V(\hat{h}_k)$ is returned. Figure 1 (a) shows the training and validation risk for the polynomial data fitting problem we saw in an earlier note. The blue dots in panel (b) of the Figure are the training data, and the orange dots are the validation data. The Least Squares loss is used to compute risks.

From panel (a) we see that the training risk decreases monotonically. This stands to reason, because a polynomial of higher degree can fit a given set of points better (or at least no worse). The validation risk, on the other hand, is high for $k = 1, 2$, where the training risk is high as well. The degrees of these two polynomials (shown in yellow and purple in panel (b)) are too low to fit the data well. A high training risk is a symptom of underfitting.

The validation risk is rather low for all degrees between $k = 3$ and $k = 8$ inclusive. This means that polynomials of these degrees generalize quite well to the given validation data. The validation risk is minimal when $k = 3$, and the procedure therefore returns $\hat{k} = 3$ as the solution. The corresponding polynomial is shown as a thick green line in panel (b).

The difference between the validation risks for $k = 3$ and $k = 4$ is small. With so few data points for validation, the choice between these two values is rather uncertain.

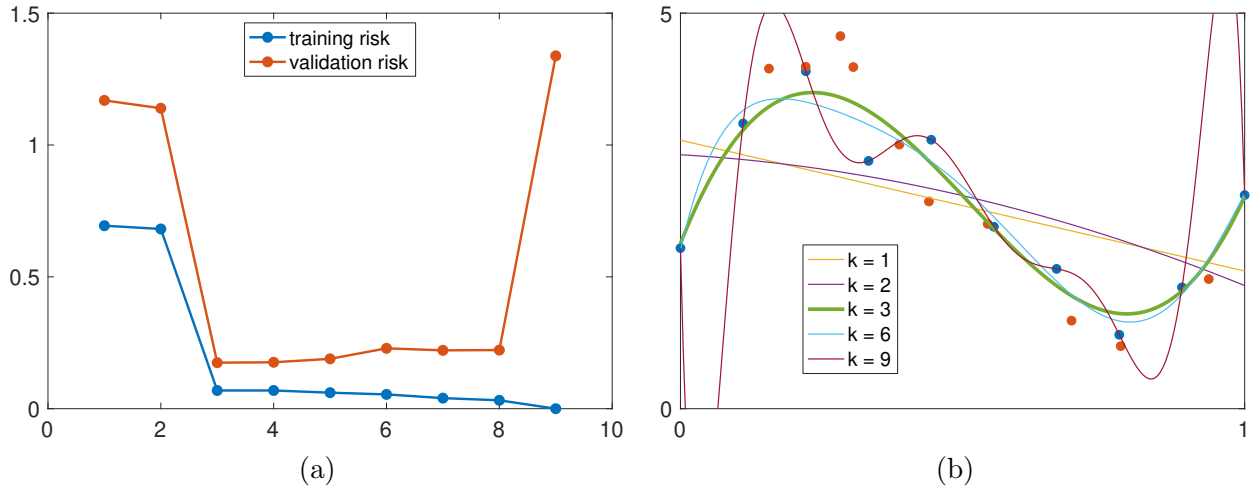

Figure 1: (a) Training and validation risk as a function of the degree k of a polynomial fit to the blue dots in (b). The lowest validation risk is achieved for $k = 3$, while the training risk keeps decreasing even beyond that. (b) Polynomials of degrees between 1 and 9 fit to the training data. Blue points are the training set and orange points are the validation set. Only some of the polynomials are shown, to reduce figure clutter. The polynomial that generalizes best to the validation data has degree $k = 3$ and is shown as a thicker, green line.

The gap between training and validation risk in Figure 1 (a) is always positive. This makes sense because the polynomial is fitted to the training data, and the fitting procedure never sees the validation data, on which it is therefore likely to perform less well. In principle, however, it is possible for the validation risk to be lower than the training risk.

Two dramatic events occur for $k = 9$: The training risk goes to zero, and the validation risk jumps suddenly to a large value. The first event occurs because $k = N - 1$: If the degree of a polynomial is at least the number of data points minus one, a perfect fit can be achieved. Thus, a training risk of zero has to do with polynomials and the size of T. The large upward jump in validation risk, on the other hand, when compared to a much smaller training loss (zero or otherwise), implies overfitting: The magenta polynomial ($k = 9$) in panel (b) follows the vagaries of the specific data sample too closely, and does poorly on previously unseen data as a consequence.

4 Measuring Generalization Performance

Suppose that you train a predictor on a data set T. This procedure may or may not involve validation, depending on whether the hypothesis space $\mathcal{H}$ depends on hyper-parameters or not. Either way, once the optimal predictor $\hat{h}$ has been found, the following question arises: How well can you expect your predictor to do on new data? That is, how well does $\hat{h}$ generalize? This question is what makes machine learning different from data fitting.

If no validation is used, the answer should be rather obvious by now, in light of our previous discussion: We cannot use T to measure generalization, because a low empirical risk is not necessarily an indication of good performance. Instead, we collect a new data set S, called the *test*

set, and compute the predictor's empirical risk $L_S(\hat{h})$ on S as an estimate of its statistical risk. None of S is used for training, and all of it used exclusively for *testing*, that is, for measuring the generalization performance of the predictor.

If validation is used, on the other hand, one may be tempted to use the validation risk $L_V(\hat{h})$ as a measure of generalization performance, since this risk is computed on a data set V that has not been used for training $\hat{h}$.

However, just as training "taints" the training data, so that validation must be performed on a data set separate from the training set, validation "taints" the validation data. In other words, model selection, just like training, is an estimation procedure that finds an optimal hyper-parameter set $\hat{\boldsymbol{\pi}}$ by evaluating the performance of various choices of $\boldsymbol{\pi} \in \Pi$ on the validation set V. If V is too small, overfitting may occur during validation as well, and $\hat{\boldsymbol{\pi}}$ may adapt too much to the idiosyncrasies of V. Perhaps if we were to use a different V we would obtain a different $\hat{\boldsymbol{\pi}}$. In summary, the issues in validation are analogous to those encountered in training as far as overfitting is concerned.

Thus, we generally need the following three sets:

- A *training set* T to train the predictor given a specific set of hyper-parameters (if any).
- A *validation set* V to choose good hyper-parameters, if the hypothesis space $\mathcal{H}$ depends on any.
- A *test* set S to evaluate the generalization performance of the predictor $\hat{h}$ learned by training on T (and, optionally, by hyper-parameter validation on V).

All these sets contain data point/value pairs $(\mathbf{x}, y)$, that is, they are all subsets of $X \times Y$.

If collecting a training set is difficult (and we saw that it typically is), collecting *three* data sets is even harder, and may even be unrealistic in applications where each sample is very costly. As an example, think of a regression system that predicts the effectiveness of a certain medical treatment for patients that exhibit certain types of symptoms. In that case, $\mathbf{x}$ may be some encoding of the symptoms, and y may be the number of years the patient survives. Clearly, following up a patient until he or she passes away is a very costly procedure, and yields just a single data sample.

Even when expense is not an overriding issue, using a separate training set T, validation set V, and testing set S means that we forgo using all the available data $T \cup V \cup S$ for training, and we therefore make the predictor intentionally worse than it could be, in order to hold out data for validation and testing.

Because of these considerations, some *resampling* techniques have been developed that allow using a single data set for both training and model selection. The most popular of these techniques is called cross-validation. Another technique, called boosting, is used less frequently in this context, but shows up in an important category of predictors called decision forests, which we will study later in this course. The next Section examines these resampling techniques.

While the need for a separate validation set may often be dispensed with through resampling, *the availability of a test set S, entirely disjoint from all other data sets used during training, is indispensable.* No research article in machine learning is ever accepted for publication if there is even the suspicion that any part of S has been used during training, either directly or indirectly.

Most new contributions in the field are tested on standard benchmark data sets, and the test data set S is closely guarded. For instance, the organizers of image recognition competitions typically make training and validation data sets available to all, but withhold test data. Performance

evaluation then occurs in one of two ways. In one protocol, the data points $\mathbf{x}$ from the test set S are published, but not the corresponding labels y. Researchers then run their predictors on these data points and submit the predictions $\hat{y}$ to the organizers. These estimate the generalization risk using $\hat{y}$ and the true values y they keep secret. An alternative protocol is to keep all of S secret. Researchers submit their code to the organizers, and these run the code for performance evaluation on S. The sanctity of S is indeed a big deal in practice!

5 Resampling Methods for Validation

Since using data sets separate from training data for validation or testing is an expensive proposition, and is sometimes unaffordable, we consider two different *resampling* techniques called *cross-validation* and the *bootstrap* that make a single data set serve double duty, for both training and validation.

These methods repeatedly *split* the training set T into two complementary sets[5] T_k and V_k, for $k = 1, \ldots, K$. For each split, the predictor is trained on T_k and tested on V_k, and the average performance of these predictors is then used as an estimate of the statistical risk of yet another predictor trained on all of T. The difference between cross-validation and the bootstrap is in how the splits are made.

5.1 Cross-Validation

In *K-fold cross-validation*, the training data are split once and for all into K (approximately) equal-sized sets V_k chosen at random. For every $k = 1, \ldots, K$, one trains the predictor on all training data points *except* those in V_k. The empirical risk of the predictor on V_k is then recorded. Once K empirical risks have been computed in this way, their mean is computed as an estimate of the generalization risk, and their variance, if desired, gives an idea of the uncertainty about this estimate. This variance is not available when standard validation (no cross-validation) is used, so this is another advantage of cross-validation. Algorithm 2 details model selection by cross-validation procedurally. Of course, V is no longer a parameter to this procedure, in contrast with validation.

Leave-one-out cross-validation (LOOCV) is K-fold cross-validation with $K = N$: In each split, each sample is held out in turn, and the predictor is trained on the remaining $N - 1$ samples and evaluated on the lone held-out sample.

The cross-validation risk is generally a biased estimate of the statistical risk, because the data sets used for training and validation are drawn from the same set T. However, since training and validation set are distinct within each split, the cross-validation risk is a better estimate of the generalization risk than the training risk.

> **Choosing a Good Value for K:** Since the training set in each split is $(K-1)/K$ times the size of T, each predictor is somewhat worse than it would be if all of T were used, and the cross-validation risk estimate is therefore somewhat pessimistic (that is, biased upward). This bias decreases monotonically as the number K of folds increases. It is smallest for LOOCV, since then $(K-1)/K = (N-1)/N \approx 1$. Nadeau and Bengio [1] show theoretically and empirically that the variance of a K-fold cross validation decreases with K as well. Thus, LOOCV ($K = N$) would seem to be the method of choice, as it leads to the smallest bias and

[5] The letter k is commonly used in the literature both in "k-nearest-neighbors" and to index rounds (or *folds*) in cross-validation, and we adhere to this use. Of course, the two "k" are semantically unrelated.

Algorithm 2 Model Selection by K-Fold Cross-Validation

procedure CROSSVALIDATION($\mathcal{H}, \Pi, T, K, \ell$)
  $\{V_1, \ldots, V_K\} = \text{SPLIT}(T, K)$ ▷ Split T in K approximately equal-sized sets at random
  $\hat{L} = \infty$ ▷ Will hold the lowest risk over Π
  for $\boldsymbol{\pi} \in \Pi$ **do**
    $s, s_2 = 0, 0$ ▷ Will hold sum of risks and their squares to compute risk mean and variance
    for $k = 1, \ldots, K$ **do**
      $T_k = T \setminus V_k$ ▷ Use all of T except V_k as training set
      $h \in \arg\min_{h' \in \mathcal{H}_{\boldsymbol{\pi}}} L_{T_k}(h')$ ▷ Use the loss ℓ to compute $h = \text{ERM}_{T_k}(\mathcal{H}_{\boldsymbol{\pi}})$
      $L = L_{V_k}(h)$ ▷ Use the loss ℓ to compute the risk of h on V_k
      $(s, s_2) = (s + L, s_2 + L^2)$ ▷ Keep track of quantities to compute risk mean and variance
    end for
    $L = s/K$ ▷ Sample mean of the risk over the K folds
    if $L < \hat{L}$ **then**
      $\sigma^2 = (s_2 - s^2/K)/(K-1)$ ▷ Sample variance of the risk over the K folds
      $(\hat{\boldsymbol{\pi}}, \hat{L}, \hat{\sigma}^2) = (\boldsymbol{\pi}, L, \sigma^2)$ ▷ Keep track of the best hyper-parameters and their risk statistics
    end if
  end for
  $\hat{h} = \arg\min_{h \in \mathcal{H}_{\hat{\boldsymbol{\pi}}}} L_T(h)$ ▷ Train predictor afresh on all of T with the best hyper-parameters
  return $(\hat{\boldsymbol{\pi}}, \hat{h}, \hat{L}, \hat{\sigma}^2)$ ▷ Return best hyper-parameters, predictor, and risk statistics
end procedure

variance. However, LOOCV is very expensive, because training is repeated N times. In addition, Nadeau and Bengio show that the improvements deriving from greater and greater values of K tend to saturate once K is around 10. As a result of their studies, they recommend $K = 15$ as a good compromise between accuracy and precision, on one hand, and computational cost on the other.

5.2 The Bootstrap

Notation: A *bag* or *multiset* is a set that allows for multiple instances for each of its elements. The number of repetitions of an element in a bag is called the *multiplicity* of that element in that bag, and the *cardinality* of the bag is the sum of its multiplicities. A set is a bag with the added constraint that every element has multiplicity 1. For instance, $\{a, a, b, b, b, c\}$ is a bag of cardinality 6 with multiplicities 2 for a, 3 for b, and 1 for c. The set $\{a, b, c\}$ is also a bag.

In the *bootstrap*, one draws K training bags T_k of N samples uniformly at random and with replacement out of the original training set T, which also has N samples. The result is that each training bag T_k is missing some of the samples in T, and contains multiple copies of some other samples.

The predictor is then trained on each bag T_k in turn, and the risk over the set V_k of samples that are not in T_k is recorded. The average of these risks over all the validation sets is the bootstrap estimate of the predictor's statistical risk, and their empirical variance is an estimate of the statistical variance of the risk.

The algorithm is the same as Algorithm 2 except for the following two changes:

- Replace the line

$\{V_1, \ldots, V_K\} = \textsc{Split}(T, K)$

with the following code:

for $k = 1, \ldots, K$ **do**
  $T_k = \textsc{Draw}(T)$
end for

where the function DRAW draws $|T|$ elements out of T at random and with replacement.

- Replace the line

  $T_k = T \setminus V_k$

  with the line

  $V_k = T \setminus T_k$

Of course, the difference

$$V_k = T \setminus T_k$$

between a set and a bag produces a set: An element in the set T is included in the set V_k once if it does not show up in the bag T_k, that is, if its multiplicity in the bag T_k is zero.

Using repeated training samples effectively changes the risk from an average loss to a weighted average of losses. Suppose that each of the *distinct* samples $(\mathbf{x}_j, y_j)$ in bag T_k has multiplicity m_j for $j = 1, \ldots, J$. Then, $J \leq N$ and

$$m_1 + \ldots + m_J = N$$

because

$$|T_k| = |T| = N \; .$$

Then, the risk on T_k, which the training algorithm minimizes, becomes

$$L_{T_k}(h) = \frac{1}{N} \sum_{j=1}^{J} m_j \; \ell(y_j, h(\mathbf{x}_j))$$

because sample $(\mathbf{x}_j, y_j)$ occurs m_j times in T_k.

The risk estimates from the bootstrap are often quite good. They are sometimes more biased than those from cross-validation, which is therefore the method of choice for selecting a hypothesis space. However, we will see later on that the bootstrap is a key ingredient of random forests, a very interesting machine learning algorithm.

Quantitative Aspects of the Bootstrap: On average, about 37% of the samples in the set T are left out of each bag T_k (and an equal fraction of samples are therefore repetitions), so the size of the set V_k is about 37% that of T on average. To see this, fix k and consider the experiment of drawing a single element out of T. The probability that any one element is drawn is $1/N$, so the probability that it is not drawn is $1 - 1/N$, and the probability that that same element is not drawn in any of the N draws is

$$\left(1 - \frac{1}{N}\right)^N$$

because the elements are drawn independently of each other. Since all elements in T are treated the same, this expression is also the expected fraction of elements that do not end up in T_k, and are therefore placed in V_k. Finally, since

$$\lim_{N \to \infty} \left(1 - \frac{1}{N}\right)^N = \frac{1}{e} \approx 0.37 \; ,$$

for large enough N about 37% of the elements of T end up in V_k and the remaining 63% end up in T_k, with repetitions. This approximation is good rather soon. For instance, when $N = 24$ we have $(1-1/24)^{24} \approx 0.36$.

References

[1] C. Nadeau and Y. Bengio. Inference for the generalization error. In *Advances in Neural Information Processing Systems (NIPS)*, pages 307–313, 2000.