Generalization

We've talked a lot about algorithms for finding ML models: linear regression, perceptron, gradient descent, minimum description length. Most of these algorithms work by trying to find a model that performs well on a set of training data (maybe in combination with other criteria).

But, we don't care how well our model performs on the training data. Instead, we care how well it performs when we deploy it. We hope that a model that performs well on training data will also do well in deployment — this property is called *generalization*.

Unfortunately, there are a *lot* of things that can go wrong and hurt generalization. Some examples:

- Concept shift: the thing we told the model to learn turns out to be not quite the right concept. E.g., the person who needs the ML model didn't communicate adequately with the person who labeled the training data.
- Distribution shift: even if the concept stays the same, the training data is not representative of the examples we see during deployment.
- Adversaries: someone wants to fool our learned model, and can influence either the training data or what the model can see during deployment.
- Missing information: we lose access to some information during deployment that we depended on during training. For example, we could measure some features of an example accurately during training, and noisily or sporadically during deployment.
- Goodhart's Law: "When a measure becomes a target, it ceases to be a good measure." If our learned model is used for anything important, people will try to present their examples in the best possible light. This is like giving a used car a new coat of paint: it's still rusty underneath, but the model doesn't see it. (This law is related to adversaries and to distribution shift, but it is enough different to be worth mentioning on its own.)
- Overfitting: even if none of the above problems happen, models rarely perform as well during deployment as they do during training.

The first five problems stem from how we use ML in the context of a larger *socio-technical system*: a combination of software, hardware, and humans that work together to try to accomplish some goal. As such, they are extremely important, but beyond the scope of what we can cover here.

> *It's worth pointing out that system-level problems can do a whole lot more than just hurt generalization; for example, they can be at the root of unfairness, bias, and lack of transparency, as well as a host of other unintended consequences.*

By contrast, the last problem (overfitting) is a purely ML problem. In these notes, we'll look why overfitting happens, as well as how to measure it and how to compensate for it.

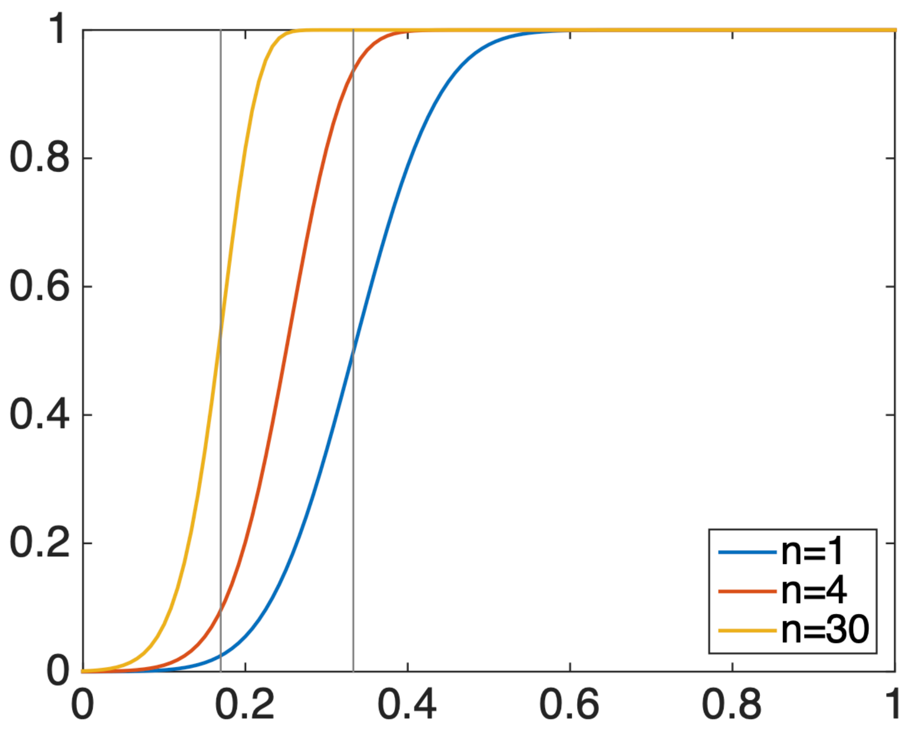
Lucky or good?

If a sports team wins a tournament, are they lucky or good? The answer is likely a little of both: they probably couldn't win if they were the worst team, but on any given day the second-best team might have a decent chance of beating the best team. Since there are often a fair number of good teams, and only one best team, it can even happen that we're fairly sure the best team will *not* win.

If a model performs best on our training data, is it lucky or good? Again, the answer is likely a little of both: since we selected our training data randomly, a model's measured performance on the training data can be higher or lower than its true performance. So, if we pick the model with the highest measured performance, we're likely to get one that was at least a bit lucky.

This phenomenon is called *selection bias*: if we use some criterion to select a model, then even if the criterion was an unbiased measurement to start off with, it will be biased for the model we pick. The amount of selection bias — the difference between the selected model's true performance and its measured performance — is called the *generalization gap*.

Here is selection bias in action:



In this plot, we look at the cumulative density function of the best of n samples from a fixed distribution. We can think of each sample as representing a performance measurement on our training data. If we measure $n = 1$ model, we'll get an unbiased but noisy estimate of its performance. If we measure more than one model (here we show $n = 4$ and $n = 30$), the distribution of the best model's performance will shift to the right — even though we set up this plot so that all models actually do equally well on our data.

> *In this plot, the performance is the model's error rate on a training set of 32 examples. All models have true error rate of $\frac{1}{3}$, but since our training set is not that big, the standard deviation of the measured error is about 0.0833.*

The amount of selection bias that we see depends on two things: how much can luck influence our performance measurement, and how many models are we picking from? In our example, with 4 models, we have almost a 95% chance of thinking our error rate is lower than it is. With 30 models, the median measured performance is around the 2.5% quantile of the unbiased distribution.

> *More precisely, instead of the total number of models, the amount of selection bias depends on how many halfway-reasonable models there are. We can get a lot of selection bias from a moderate number of pretty-good models, or a much larger number of OK models, or a really huge number of mediocre models.*

Overfitting

Overfitting is what happens when we get selection bias while trying to pick the best model. Overfitting actually covers two related problems:

- First, we might pick a bad model that just happens to get lucky. This tends to happen when there are a lot of models to choose from, and when performance measurements are fairly noisy.
- Second, we might pick a pretty good model, but have a too-rosy view of its performance.

We don't really want either of these problems, but the first problem is usually worse for us if it happens. Fortunately, overfitting has to be fairly bad for the first problem to kick in. So, typically, successful ML systems are able to mostly avoid the first problem, but still have to deal at least somewhat with the second problem.

For that reason, it's interesting to try to get an idea how strongly the second problem affects us — that is, how much we're overestimating our performance. There are two main strategies we can use for this purpose: *generalization bounds* and *holdout data*. We'll also mention one more, less common for this purpose but still interesting: *differential privacy*.

Generalization bounds

In some situations, we can analytically calculate a bound on our likely generalization gap. The array of available methods for this task could fill several courses on their own, but one of the simplest and most useful is the bound for choosing among finitely many classifiers. This bound is representative of the sort of results we can prove in other situations as well, so we will describe it in a bit of detail.

Suppose we have a supervised classification problem: we want to learn from data $(x_1, y_1), \ldots, (x_n, y_n)$ to predict $y_i \in \{-1, 1\}$. And, suppose we have a finite set $\{f_1, f_2, \ldots, f_m\}$ of classifiers.

We use our training data to measure the error rate of each classifier: define ϵ_{ij} to be 0 if $y_i = f_j(x_i)$, that is, if classifier j gets example i right. And define ϵ_{ij} to be 2 if $y_i \neq f_j(x_i)$, that is, if classifier j gets example i wrong. The observed error rate of classifier j is the average of ϵ_{ij} for $i = 1 \ldots n$, $\epsilon_j = \frac{1}{n} \sum_i \epsilon_{ij}$. Write $\bar{\epsilon}_j$ for the *true* error rate of f_j, that is, the expected value of ϵ_j. We'd like to pick the classifier f_j that minimizes $\bar{\epsilon}_j$, but the best we can do is to pick the one that minimizes ϵ_j.

Because of this difference, we expect selection bias. The more training data we have (the larger n is), the more accurately we can measure the error of each classifier f_j; this mitigates selection bias. But the more classifiers we pick from (the larger m is), the worse our selection bias will get.

It turns out that we can prove the following bound. For all j and for any $\delta \in (0,1)$, the following holds with probability at least δ:

$$\bar{\epsilon}_j \leq \epsilon_j + \sqrt{\frac{\ln m + \ln \frac{1}{\delta}}{2n}}$$

That is, the true error could be larger than the observed error. But with high probability the excess error is $O(\sqrt{\frac{1}{n}\ln m})$: it grows only logarithmically with the number of models we consider, and decreases as the inverse square root of the number of training examples. Importantly, this bound holds for *all* of our classifiers, including the one we picked by minimizing training error. So, if we have a lot of training examples and not hugely many models, our selection bias won't be too bad.

To prove the above bound, we can use the Chernoff inequality to bound the difference between each ϵ_j and its mean. Then we can use a union bound to get a result that holds for all j simultaneously.

*One of the most useful extensions of the above result is for the case of an infinite space of classifiers indexed by a vector of parameters or weights. We can typically **cover** such a space with a finite set, in the sense that each of our infinitely many original classifiers is close to at least one of the finitely many chosen classifiers. In this case, we can almost directly apply the above result, and the generalization gap will be bounded by the log of the size of our cover. The latter number can be related to the **dimension** of our set of classifiers, which often approximately coincides with the dimension of our parameter vector.*

Holdout data

The other way to estimate the generalization gap is with holdout or validation data. We'll show some slides about this class of approaches in lecture.

Within this class of approaches, there are at least three important sub-classes:

- validation
- cross-validation
- bootstrap

Differential privacy

There's one more approach that people use to help control generalization: *differential privacy*. As the name suggests, this approach was actually designed for a different purpose, namely to help ensure privacy — itself a very important goal. Fortunately, the two goals are not in conflict: we often want both generalization and privacy.

Differential privacy is still somewhat rare in practice: while it is deployed in real systems, it is also subject to some difficult tradeoffs that we will discuss below. So, it's likely not the first approach we should try for promoting generalization, but it is definitely one to keep in mind.

The rough idea behind differential privacy is to try to ensure that anyone who has access to the *output* of a learning algorithm can't figure out very much about its *input*. For example, suppose we want a classifier that recommends whether to approve loan applications. In order to learn this classifier, we might expect to need training data that contains sensitive information, like applicants' salaries, debt levels, and credit histories. It would be potentially embarrassing if someone could look at the learned classifier and recover the fact that James Q. Doe of 123 Evergreen Terrace has just been fired from his job.

> *This scenario is more plausible than one might naively hope. Researchers have demonstrated the ability to recover surprising details from the outputs of trained neural networks.*

Our approach to differential privacy will be to make sure that the output of the learner can't tell us whether a particular example is even present in its training set. This ensures privacy, since it's hard to reveal anything about an example if we can't even be sure we have seen it.

Interestingly, the same property also ensures generalization. If our performance on test examples were significantly different from our performance on training examples, that would let us tell whether we have seen a particular example — which we aren't supposed to be able to do.

Algorithm as function

We can think of a learning algorithm $\mathcal{A}$ as a function: it takes in a dataset D and produces a hypothesis (or model) h. If each training example comes from some input space X, then $D \in X^n$. We'll also allow $\mathcal{A}$ to take some random bits $r \in \{0,1\}^m$ as input: e.g., it could use these to initialize the weights for gradient descent, or decide on the order to process examples. If we write H for the hypothesis class (the set of possible hypotheses), then

$$\mathcal{A} \in (X^n \times \{0,1\}^m) \to H$$

For example, here's a small dataset (a subset of the Fisher iris data):

petal length	petal width	sepal length	sepal width	species
6.8	3.0	5.5	2.1	virginica
5.7	2.5	5.0	2.0	virginica
5.8	2.8	5.1	2.4	virginica
6.5	3.0	5.5	1.8	virginica
6.3	2.3	4.4	1.3	versicolor
5.6	3.0	4.1	1.3	versicolor
5.5	2.5	4.0	1.3	versicolor
5.5	2.6	4.4	1.2	versicolor

We can train a Gaussian naive Bayes classifier on this data. Recall that the GNB linear discriminant is

$$w \cdot x + b \geq 0$$

where the weight vector w is the difference between class means,

$$w = \mu_{vc} - \mu_{vi}$$

and the intercept b is chosen so that the class boundary bisects the line between class means,

$$w \cdot \frac{\mu_{vc} + \mu_{vi}}{2} + b = 0$$

From our dataset, we get

$$w = (-0.475, -0.225, -1.05, -0.8)^T \qquad b = 4.89$$

This function happens to be deterministic: the standard GNB classifier doesn't use any randomness.

Privacy

We will say that $\mathcal{A}$ is ϵ-differentially private if its distribution of output hypotheses is insensitive to changing a single example in the training set: for all datasets $D \in X^n$, all datasets D' we can get by substituting a single example in D, and all $S \subseteq H$,

$$\exp(-\epsilon) \le \frac{P(\mathcal{A}(D, r) \in S)}{P(\mathcal{A}(D', r) \in S)} \le \exp(\epsilon)$$

The quantity in the middle is the ratio of the probability of $h \in S$ for D and D'. The inequalities say that this ratio is close to 1. (Note that, by symmetry, we only need to assert one side of the bound, and the other side would follow; but we state it this way for clarity.)

There's a more general version called (ϵ, δ)-differential privacy that allows for additive as well as multiplicative terms in the probability bound.

For example, we could ask how our Gaussian naive Bayes classifier above changes when we substitute an example. Let's say we change the first row above by subtracting 0.5 from all of the features:

petal length	petal width	sepal length	sepal width	species
6.3	2.5	5.0	1.6	virginica

This changes our classifier to

$$w = (-0.35, -0.1, -0.925, -0.675)^T \qquad b = 3.88$$

That is, it subtracts 0.125 from each component of w, since it subtracts $\frac{0.5}{4}$ from each component of μ_{vi}. And it changes b to compensate: we can calculate

$$db = -\mu_{vi} \cdot d\mu_{vi}$$

which lets us estimate a change in b of about -1.

This new classifier is different from the one we learned before. Since the GNB learner is deterministic, that means we can tell with certainty which training set we used. So, the standard GNB classifier is *not* differentially private at any ϵ.

In fact, we can see something from the above calculation: there are no deterministic algorithms that are differentially private. Instead, we have to depend on randomness in the algorithm itself if we want privacy. This makes sense: a deterministic algorithm must choose between ignoring a datapoint completely or changing its hypothesis in response to it. In the first case the algorithm is private — but algorithms that ignore their data aren't that useful. In the second case the algorithm is completely non-private, since an observer can figure out with perfect confidence whether the datapoint is present by checking the algorithm's output.

Randomness

There's a simple wrapper we can put around any algorithm to improve its privacy: we add a little bit of random noise to the output. For GNB, the entire output is determined by the two class mean vectors μ_{vi} and μ_{vc}. So, instead of reporting the exact class means, we could report the noisy versions

$$\mu_{vi} + \eta_{vi} \qquad \mu_{vc} + \eta_{vc}$$

where $\eta_{vi}, \eta_{vc} \in \mathbb{R}^4$ are random noise vectors. If we randomize like this, it can help hide our exact input data: we don't know if a large mean is due to a large datapoint or just to the random noise.

How much randomness is enough? That depends on how sensitive our algorithm is to each individual datapoint. If the output can change a lot from changing an individual datapoint, then we need a lot of noise to mask this change.

GNB is actually pretty stable: its output is computed as the mean of its inputs, so changes in the inputs don't wind up changing the outputs a lot. For example, if we assume that each input feature varies within a range of 2cm, then the biggest possible change that we can make by replacing one out of the four points in one class is to change the corresponding class mean vector by $\frac{2}{4}$ in every component. So, if the distribution of η is such that the probability of $\eta = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})^T$ is at least $\exp(-\epsilon)$ times the probability of $\eta = (0, 0, 0, 0)^T$, we will have ϵ-differential privacy.

The most common way to add noise is to let each noise coordinate η_i follow an independent Laplace distribution with scale parameter c:

$$P(\eta_i) \propto \exp(-|\eta_i|/c)$$

Since there are four coordinates in each class mean vector, and we want to let each coordinate change by $\frac{1}{2}$, we need $\exp(-|\frac{1}{2}|/c) \geq \exp(-\frac{\epsilon}{4})$. Equivalently, we need $c \geq \frac{2}{\epsilon}$. For example, if we pick $\epsilon = 0.1$, meaning that the evidence for or against a particular point being included (represented as a probability ratio) is no stronger than $\exp(-0.1) \approx 0.9$, then we need $c \geq 20$.

Notice that this is a *lot* of noise: the standard deviation of a Laplace distribution is $\sqrt{2}c$, or about 28 with this setting of c. So the noise in μ_{vi} and μ_{vc} is substantially larger than μ_{vi} and μ_{vc} themselves. We should expect this outcome: we have a small training set, so there's a substantial effect from swapping out even one datapoint, and we need a lot of noise to hide this effect. But even with larger training sets and modest privacy guarantees, the amount of noise needed can be significant.

The above discussion shows that differential privacy is possible. But it also illustrates the biggest tradeoff that we have to manage: if we add too little noise, our privacy guarantees are weak, while if we add too much, we can ruin our learner's performance. Unfortunately, in many situations, we need a lot of noise for useful privacy guarantees; this limits the applicability of differential privacy. For this reason, most current applications of differential privacy are in situations where there's a really large dataset that we can't access unless we provide privacy guarantees. For example, big tech companies may employ differential privacy to learn from data on users' phones: this data can be strongly private, but there's a huge amount of it. In this situation it's worth taking the performance hit of adding noise in order to be able to access a large and valuable dataset.

Post-processing

The differential privacy guarantee still holds if we post-process the output of an algorithm in any way that doesn't use the training data. For example, above we proved differential privacy for the algorithm that returns μ_{vc} and μ_{vi}; but we actually care about the algorithm that returns the classifier parameters w and b. Since w and b are functions of μ_{vc} and μ_{vi}, differential privacy still holds for the algorithm that returns w and b. (It makes sense that post-processing like this is OK: if there were some kind of post-processing we could do to reveal whether an example was part of our training set, then the original algorithm couldn't have been differentially private to start with.)

Adaptivity

If we want to run more than one algorithm on the *same* dataset, with each new algorithm depending on the output of the previous one, this is called *adaptivity*. We can maintain differential privacy even in the face of adaptivity if we are careful: if we run k adaptive steps, each of which is $\frac{\epsilon}{k}$-differentially private, the whole sequence is ϵ -differentially private. It's easy to see that this works by multiplying the bounds on probability ratios:

$$\exp(-\frac{\epsilon}{k})\exp(-\frac{\epsilon}{k})\ldots\exp(-\frac{\epsilon}{k}) = \exp(-\epsilon)$$

Generalization

Now we can come back to the question of generalization. Suppose that we run a differentially private learning algorithm, and part of the algorithm's output is an estimate of the performance of the learned model. It turns out that this performance estimate has to generalize: roughly, if the performance on test data were significantly different from the reported value, we'd be able to tell the difference between training data and test data. Unfortunately, stating the precise generalization bound requires more notation than we have developed here; but an example of the kind of bound we can prove is in this paper.