
NE
E
SE
S
SW
W
NW
N

Understanding Machine Learning

Understanding Machine Learning (I)

Understanding Machine Learning is another book on Miri's research guide. I've previously written about Topology and Linear Algebra Done Right. With this one, my write-up ended up looking a lot more like a self-contained introduction into the field and a lot less like a review. I've decided to go with that, and make this a sort of shortened version of the book.

Throughout this series, I will be using ***bold and italics*** to mean "this is the first official mention of a new Machine Learning term", whereas normal italics means regular emphasis.

The Learning Model

The book begins with a sort of mission statement. Given a ***domain set*** X and a ***target set*** Y, we wish to *find a function* $h : X \to Y$, which we interchangeably call a ***hypothesis*** or a ***predictor*** or a ***classifier***. In the example used in the first chapter, X represents the set of all papayas, and Y represents a two-element set where one element is the label "tasty" and the other the label "not-tasty". To compress this into something manageable, we represent papayas by a set of ***features***, in this case *color* and *softness*, which we both map onto the interval $[0, 1]$, and we represent the two labels by 1 and 0. I.e. $X = [0, 1]^2$ and $Y = \{0, 1\}$. Now the idea is that, if we learn such a predictor, we can apply it to any new papaya we encounter, say on the market, and it will tell us whether or not it is tasty before we've spent any money on it.

This model seems quite general. It covers chess, for example: one could let X be the set of all possible states of the chessboard and Y be the set of all moves; then we want h to map each $x \in X$ onto the best possible move. In fact, it is hard to see which solvable problem could not be modeled this way. Any time we think we can in principle solve some problem, it would consist of us looking at some stuff and then presenting the solution in some way. Now we just let the set of all possible instances of such stuff be X and the set of all possible solutions be Y, and we have an instance of this model.

One would have to put in a lot more work to fully formalize this problem – and this is exactly what the book does. Recall that X is called the *domain set*. If Y is finite as it is for the papayas example, we also call it the ***label set***.

We model the ***environment***, which creates our labeled domain points (x, y) by a probability distribution D over $X \times Y$ as well as the *i.i.d. assumption* (not bolded since

it's not ML-specific), which states that all elements are *independently* and *identically* distributed according to D. In other words, whenever a papaya is plucked in the real world, the environment throws some coins and chooses its color and softness, as well as tastiness, according to D. Of course, we generally assume that the color and softness have a major impact on the tastiness of the papaya, but it is not reasonable to assume that they *fully* determine its tastiness – surely it is possible that two papayas that are equally colored and equally soft taste differently. Thus, given some particular x, we might reasonably hope that either $(x, 0)$ or $(x, 1)$ is going to be a lot more probable than the other, but both are still going to be possible. On the other hand, in the chess example, one could argue that the domain set fully determines the value of the target set, i.e. the best possible move (although then the idea of having a probability distribution over X becomes problematic). In such a case, one can alternatively model the environment as having a "true" function $f : X \to Y$. This approach is strictly less general than the former (it corresponds to a probability distribution where for every x, either $(x, 0)$ or $(x, 1)$ has probability 0), but in both cases, the joint distribution / true function is not known, and it is exactly what we are trying to figure out.

We say that the algorithm A which outputs a predictor $h : X \to Y$ is the ***learner***. As its input (assuming we are doing what is called ***supervised learning*** here), the learner has access to a ***training sequence*** S. This is a sequence of labeled domain points, i.e. $S = ((x_1, y_1), \ldots, (x_n, y_n))$ which *have been generated by the joint distribution* D. For this reason, one can consider the training sequence to be a glimpse into how the environment works, and the learner's job to be to *generalize* from that sequence to all of D, i.e. to output a predictor h which does a good job on not just the training data but also on new elements that are generated by D.

We also define a ***loss function*** ℓ to formalize what we mean by "good job". The loss function takes a hypothesis and outputs a number in $[0, 1]$ that is meant to be a rating of how good the hypothesis is, where small values are good and large values are bad. For the case of ***binary classification*** (where $|Y| = 2$), we will set $\ell(h) := D\{(x, y) \in X \times Y \mid h(x) \neq y\}$, i.e. the probability that our predictor gets the label on a freshly sampled point wrong (although in practice, one might also be interested in loss functions that penalize false positives more strongly than false negatives, or vice versa). A perfect predictor will have loss 0, but in the setting of a joint distribution D, this will generally be impossible. How good the best possible predictor will perform depends on how well the feature vector describes the domain points. With a poor choice of features, the learner is doomed to fail from the beginning (like if you want it to recognize spam emails, but you only show it aspects of the

emails such that there is at best a statistically weak correlation between them and whether the email is spam or not-spam). Of course, we cannot evaluate our loss function since we don't know D, but it is nonetheless possible to make statements about the expected loss of the predictor which a learner outputs.

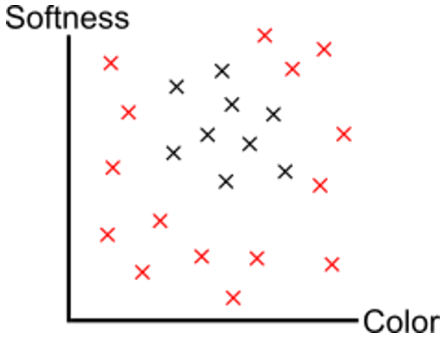
Finally, we define an ***empirical loss function*** ℓ_S which describes the error of a predictor *on the training sequence*, i.e. $\ell_S(h) := \frac{|\{(x,y)\in S : h(x)\neq y\}|}{|S|}$ Unlike the regular loss function, the empirical loss function ℓ_S is something we can evaluate. We use the term ***true error*** to refer to the output of ℓ, and ***empirical error*** to refer to the output of or ℓ_S.

This describes (almost) all objects of the learning model. Now the question is how to implement the learner A. Formally, A is described as simply an algorithm, so we have complete freedom to do whatever we want.

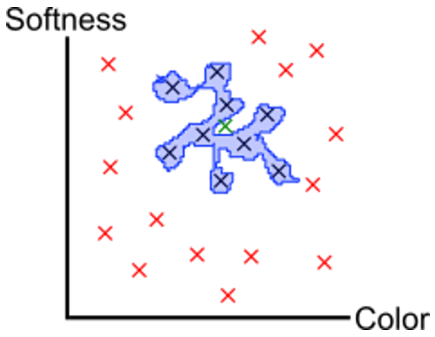
Generalization / Prior Knowledge / Overfitting

The learner A takes a training sequence, does whatever it wants with it, and outputs a predictor. How would we go about defining A? A reasonable-sounding starting point is to choose a predictor which performs well on the training sequence S. Given that S is representative of the environment (recall that it was generated by D), the hope is that such a predictor would also perform well on the environment.

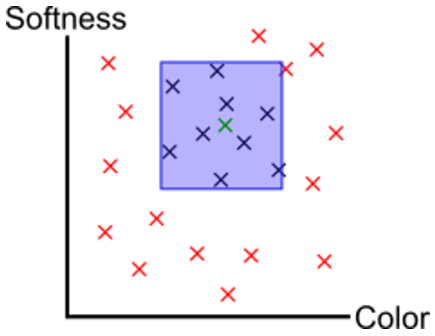
This alone does not answer the question, though. Suppose we are given the following training sequence (or -set as we will ignore order for now):



(Black points ↔ tasty instances, red points ↔ not-tasty instances.) There are many ways to construct a predictor with zero error on this training set. Here is one (namely the predictor that labels points as tasty iff they are in the blue area):



This does not seem like a good predictor. For example, the green point I added (which is meant to be a new instance not part of the training sequence) would almost certainly correspond to a tasty papaya, yet our hypothesis predicts that it is not tasty. Nonetheless, this predictor has zero error on the training sequence (by the definition of ℓ_S). An even more extreme example of this is A_{memorize}, which simply remembers all domain points and outputs a hypothesis that only labels those instances as positive that have occurred in the training sequence with a positive label. The process of outputting such a hypothesis is often called ***overfitting***. It's said that the hypothesis fits the data "too well," as the book puts it. But I think this is not quite the right way to look at it. Consider this alternative hypothesis:

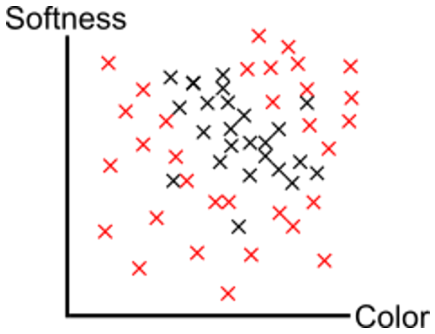


This hypothesis seems better, but not because it is a worse fit for the data – both hypotheses have zero empirical error. There is nothing wrong with fitting the data, there is only something wrong with fitting the data *at the cost of outputting an "unreasonable looking" predictor.* I think the way to arrive at a more accurate description is by looking at it from a Bayesian perspective. We have a prior on how a proper predictor will look like, and this prior says that the first predictor is worse (as illustrated by how it classifies the green point). However, suppose we had a new example right next to the green point, and it was not-tasty. We would probably still dislike that predictor – the one not-tasty papaya could be an outlier. On the other hand, if we had a million sample points right next to the green one, all of which not-tasty, we would eventually admit that something super weird is really going on here and a very particular softness-color combination really does lead to papayas being not-tasty.

I think the correct framing of the overfit/underfit dilemma is that it is about *how far to update away from priors based on evidence,* or alternatively, *how to weigh priors vs evidence*. It's also worth noting that if we had a training sequence that included *all* possible instances (assuming there is a true labeling function for the sake of this example), we could disregard priors altogether – in that case, A_{memorize} would work

work perfectly. Analogously, there is always some amount of evidence that should overturn any prior. Of course, in our formal model, there may be infinitely many possible feature vectors, but in the real world, every instance description is bounded so the space of all instances must be finite.

One crude way of deciding how much to weigh priors vs evidence is by committing to a ***hypothesis class*** ahead of time – i.e. to a subset $H \subset \{f : X \to Y\}$ – and then finding a predictor that performs maximally well on the training sequence, only taking into considerations predictors in the class we committed to (i.e., a predictor in the set $\text{argmin}_{h \in H}[\ell_S(h)]$). It is crude because it only allows the prior to give a binary "yes" or "no" to a hypothesis, rather than a "probably not, but maybe with enough evidence". Suppose (for the papaya example), we pre-commit to the class of *axis-aligned rectangles*, as the book suggests (i.e. rectangles with two horizontal and two vertical sides). Consider the following training sequence:



This does not seem like that unreasonable of a glimpse into reality – perhaps it is really the case that if green papayas are too soft, they become gross, but if they're more orange or whatever, then it's fine if they're soft. You might not expect that to be the case, but you'd probably assign it some reasonable probability. This is not allowed by the class of axis-aligned rectangles: cording to this class, the only possible predictor is one where both a certain softness area and a certain color area have to be hit, and one factor does not influence the other. Therefore, if we applied a learning algorithm to this training sequence and restricted it to that class, it would have to fit an axis-aligned rectangle in there as best as possible – which would perform much better than chance, but still seems like a suboptimal choice.

A more sophisticated way of answering the "how strongly to weigh evidence vs priors" question is to choose a *family of several hypothesis classes*, as well as a *weighting function* that privileges how much we like each class. Then, perhaps the class of axis-aligned rectangles would have the highest initial weighting, but the class of all rectangles would still be rated decently, so that if it performs substantially better – as is the case here (although it would still have nonzero empirical error) – the learning algorithm would end up preferring a non-axis-aligned rectangle. We can also look at this as having uncertainty about the right model in addition to uncertainty about which predictor within the model to choose. More about this later.

PAC learning

Even though I've criticized the pre-commitment approach, it is a good place to begin a more formal analysis. In particular, we can now talk about a hypothesis class being "learnable". Here, being learnable means that, with sufficiently many training instances, one can prove that, a predictor that performs optimally on the training sequence will *probably* perform *almost* as well on the real world *as the hypothesis which has the lowest real error in the class.* Crucially, this holds for every joint probability distribution D.

So this definition is not about how well the hypothesis we end up with performs on the real world in *absolute* terms – that's not something which is possible to bound, because the hypothesis class may just not contain any good predictor at all (as is the case in the example above if we use the class of axis-aligned rectangles, and of course other classes could be much worse). Rather, it is about how good of a choice the lowest-empirical-error hypothesis is *given that we are stuck with this class* – i.e. how large is the difference between the real error of [the hypothesis with minimal empirical error] and the real error of the best hypothesis, i.e. [the hypothesis with minimal real error]. Such a bound is important because, since we can't measure the real error, we cannot output the best hypothesis directly.

There are also two caveats built into the definition. One is the "almost" part; one can never guarantee that the hypothesis with minimal empirical error is exactly as good as the hypothesis with minimal real error because there is randomness in the creation of the training sequence. The second is the "probably" part, one can never upper bound the difference with full certainty because there is always some chance that the training sequence just happens to be unrepresentative. For example, there is always some chance that the training sequence doesn't contain any positively labeled points, in which case A has no way of reliably outputting anything reasonable. However, as the training sequence gets larger, this becomes increasingly unlikely.

The complete formal definition can be stated thus (where we write A(S) to denote the predictor which A outputs upon receiving the training sequence S):

> A hypothesis class H is called ***PAC learnable*** iff there exists a learner A and a function $m^* : (0,1)^2 \to \mathbb{N}$ such that, for any probability distribution D over $X \times Y$, any "gap" $\epsilon \in (0,1)$, and any failure probability $\delta \in (0,1)$, if the training sequence S consists of at least $m^*(\epsilon, \delta)$ i.i.d. examples sampled via D, then with probability at least $1 - \delta$ over the choice of S, it holds that $\ell(A(S)) \leq \ell(h^*) + \epsilon$ for all $h^* \in \operatorname{argmin}_{h \in H}[\ell(h)]$.

While this defintion is complicated, it is not unnecessarily complicated – the complexity is just an inherent feature of the thing we are trying to define. This kind of

learnability is called PAC learnability because the definition states that A(S) is *probably* (hence the δ) *approximately* (hence the ϵ) *correct* (hence the "≤").

One can prove that, whenever a class is PAC-learnable at all, then the algorithm A_{ERM} which chooses its predictor such that $A_{ERM}(S) \in \text{argmin}_{h \in H}[\ell_S(h)]$ is a successful learner. In other words, we can always just choose the predictor that performs best *on the training sequence* (or one of them if several are tied). The abbreviation ERM stands for ***empirical risk minimization.***

One can also prove that *every finite hypothesis class is PAC learnable.* In particular, the ***sample complexity*** (this is a name for the output of the function m* which upper-bounds the number of training points needed to be probability approximately correct) is upper bounded by the term $\frac{\ln(|H|/\delta)}{\epsilon}$.

Let's reflect on this result a bit. If we use A_{ERM} as our learner, then learnability implies that the difference between the true error of $A_{ERM}(S)$ and the true error of the best hypothesis in H is probably very small. In particular, if H consists of 10000 hypotheses, and we want a 99% assurance that the difference between $A_{ERM}(S)$ and the best predictor in H is at most 0.1, then $\ln(10000/0.01)/0.1 \approx 138.15$ so we if we have a training sequence with 139 labeled points, we're good. The only remaining error is then the error of the best hypothesis in H. This is also called the ***approximation error***, while the error we just bounded is called the ***estimation error***.

Now consider declaring the class H to be the set of all predictors such that there exists a C program, whose compiled code is at most 10000 bits long, that implements the predictor. It's a safe bet that this class has a very low approximation error for almost all practically relevant cases (or if not then we can allow longer programs). Given that the dependence on the size of H – in this case, 2^{10000} – is only logarithmic, it increases the sample complexity by a factor smaller than 10000. If we then demand an estimation error of at most 0.01 and some very high confidence (note that the dependence on the confidence is also logarithmic, so even demanding a 0.9999 certainty doesn't change much), we would still end up with a sample complexity of only around a million. Gathering a million examples is probably doable in many cases (although as George points out, assuming that the i.i.d. assumption holds may not be realistic in practice). Thus, we have just derived a general algorithm which, provided there is a way to obtain sufficiently many i.i.d. data points, solves every binary classification task, namely

1. Perform an exhaustive search over all C programs of compiled length in bits at most 10000

2. Throw out all programs that don't implement a predictor

3. Compute the empirical risk for each one that does

4. Output the one with the lowest empircal risk.

For the output of this algorithm, both the *approximation error* (i.e. the error of the best C program) and the *estimation error* (i.e. the difference between the error of the hypothesis A_{ERM} outputs and that of the best C program) are going to be very low!

Back in the real world, step 1 does of course mean that this algorithm is utterly useless in practice, but *only* due to its runtime. With unbounded computing power, this approach would work wonderfully. In principle, everything (well, at least every binary classification task) is learnable from a relatively small set of examples, certainly fewer examples than what is often available in practice. It doesn't seem completely unreasonable to consider this some (small) amount of evidence on the importance of talent vs data, and perhaps as decent evidence on the inference power of a really smart AI?

The next post will discuss some theoretical limits of learning and prove the No-Free-Lunch theorem.

Understanding Machine Learning (II)

This is part 2 of a three part series on the fundamentals of Machine Learning as presented by this book. It builds heavily onto part I.

Uniform Convergence

Uniform convergence is a property which a hypothesis class may satisfy that will end up being equivalent to PAC learning, and the ability to use either definition whenever needed will be quite useful. "Uniform convergence" means *convergence* with respect to the difference between the empirical error and the real error, which is *uniform* with respect to all hypotheses in the class – i.e. we can bound the difference between real and empirical error across all hypotheses simultaneously. We have the following formal definition:

> A hypothesis class H has the uniform convergence property iff there is a function $u^* : (0,1)^2 \to \mathbb{N}$ such that, for every probability distribution D over $X \times Y$, every gap $\epsilon \in (0,1)$ and every confidence level $\delta \in (0,1)$, if $|S| > u^*(\epsilon, \delta)$, then with probability at least $1 - \delta$ over the choice of S, the inequality $|\ell_S(h) - \ell(h)| \leq \epsilon$ holds for all $h \in H$.

Recall that PAC learnability demands that the true error of the hypothesis output by A_{ERM} is low. On the other hand, uniform convergence demands that the gap between the empirical and the true error is low. Unsurprisingly, both properties are equivalent. We will prove one direction now, namely that uniform convergence implies PAC learnability.

Suppose the uniform convergence property holds for values ϵ, δ. We apply A_{ERM} to our task, feed it our training sequence S which fulfills the property that $|S| > u^*(\epsilon, \delta)$, and now want to argue that the true error of $A_{ERM}(S)$ is not much higher than the true error of some $h^* \in \operatorname{argmin}_{h \in H}[\ell(h)]$. The uniform convergence property does not allow us to compare both errors directly, rather, we can argue that $\ell(A_{ERM}(S)) - \ell_S(A_{ERM}(S)) \leq \epsilon$ because of uniform convergence, that $\ell_S(A_{ERM}(S)) - \ell_S(h^*) \leq 0$ because of what A_{ERM} does, and finally that $\ell_S(h^*) - \ell(h^*) \leq \epsilon$ because of uniform convergence. Putting these three inequalities together, we do indeed obtain a bound on $\ell(A_{ERM}(S)) - \ell(h^*)$, but it is 2ϵ not ϵ. The confidence parameter δ stays the same. Of course, the doubling of the gap does not matter for the definition, since we can choose both ϵ and δ to be arbitrarily small anyway (to reach the gap ϵ for PAC learning, we simply choose $\frac{\epsilon}{2}$ for u^*), so this simple argument indeed proves that every hypothesis class which has the uniform convergence property is PAC learnable.

Before we get to more exciting things, consider the following problem in isolation. Let X be some finite set and let H be the set of all functions from X to $\{0,1\}$. Think of some "true" function $f^* : X \to \{0,1\}$ and of H as hypothesis class containing every possible candidate. To begin, the composition of H reflects that we're wholly ignorant as to the nature of f^* (every possible function is in H). Now, suppose we are given some partial information about f^* in the form of some subset $U \subseteq X$ and a function $g : U \to \{0,1\}$ such that $g = f^*_{|U}$. (This notation means, "g restricted to the set U".) We now only care about those functions in H that agree with g on the set U.

How much does this help? For every element $x \in U$, we now know the correct label, namely g(x). But consider an element $y \in X - U$. Does our partial knowledge of f^* help us to figure out the label of $f^*(y)$? The answer to this is a resounding no. Even if U contains every element in X except y, there are still 2 possible candidates in H,

namely the function f_0 which agrees with g on U and sets y to 0, and f_1 which agrees with g on U and sets y to 1.

This means we're still at 50/50 odds with regard to the label of y, the same as we were from the beginning. Thus, *if every labeling combination is possible* (and equally likely), *then learning the label of some points is zero information about the label of other points.* This principle will be key to the next two sections.

The No-Free-Lunch Theorem

The overfitting question is about how to weigh evidence vs priors, but is there a need for priors at all? What happens if one only relies on evidence? The papayas example shows that A_{ERM} will not reliably do the job, but perhaps there is some other, more clever learner that will always succeed given enough samples?

It will come to no surprise that the answer is negative (for any reasonable definition of "enough samples"), and this is formally proven in the ***No-Free-Lunch Theorem***. Given any learning algorithm A, we shall construct a domain set X, a probability distribution D, and a hypothesis class H such that A(S) will have a high expected error. ("Expected" as in, we take the set of all possible training sequences as our probability space and consider A(S) to be a random variable so that we can compute its expected value.) The label set will still be just $\{0, 1\}$, so the No-Free-Lunch theorem holds for binary classification. One noticeable feature of this proof is that we will require the instance domain to be much larger than the training sequence.

Let m be the length of the training sequence S, and let $C \subseteq X$ be a set of size 2m. We will only consider probability distributions which put all of their probability mass into elements of C, so the instances outside of C will not matter. Let H' be the set of all functions from C to $Y = \{0, 1\}$, and let H be the set obtained by taking every function from H' and extending it to a function from all of X to Y by letting it label every element outside of C by 0. (Again, these elements do not matter.) In symbols,

$$H = \{ h : X \to Y \mid \exists h' \in H' \text{ such that } [\forall x \in C : h(x) = h'(x)] \wedge [\forall x \in X - C : h(x) = 0] \}$$

Note that we have $|H| = |H'|$.

Now, we construct a set D of many different probability distributions. Each $D \in D$ will sample each domain point $x \in C$ with equal probability, but they will label the element differently. (We shall not formally switch to the setting of a true function in the environment, but the probability distributions which we consider are all consistent with such a function.) To be more precise, we construct one probability distribution for each function $f \in H'$ that labels each element according to f. In symbols, for each $f \in H'$ we set

$$D_f((x, y)) := \begin{cases} 1/|C| & ; \quad x \in C \wedge y = f(x) \\ 0 & ; \quad \text{otherwise} \end{cases}$$

and we let $D = \{D_f \mid f \in H'\}$. Notice that the learner is now in a situation analogous to what we've discussed earlier: it can learn about at most half of the points in the environment (because $|S| = m = \frac{1}{2}|C|$), and since every in-principle-possible labeling function is equally viable, that means it doesn't know anything about the labels it doesn't see, and those are at least half. Thus, it will have a sizeable total error – in fact, at least $\frac{1}{4}$ in expectation, since A has to blindly guess about half of all elements.

The proof consists of formalizing the above argument. This gets more technical, but I'll provide the details for the part that I think is at least an interesting technical problem. For each probability distribution D_f, one can

enumerate all possible training sequences $S_1^{D_f}, \ldots, S_k^{D_f}$ that are obtained by querying D_f repeatedly, m times in total. (We have $k = (2m)^m$ but this is not important.) The expected loss for this probability distribution is now $\frac{1}{k}\sum_{i=1}^{k} \ell(A(S_i^{D_f}))$. The problem here is that A may just guess all unseen labels correctly – it may even ignore S entirely and guess the labels of *all* elements correctly. Of course, such a learner would then perform exceptionally poorly on other probability distributions; the point is that it cannot do better than guess. Thus, averaging over all training sequences does not do the trick; we actually want to average over all *probability distributions*. To achieve this, recall that we need to show that there exists s*ome* probability distribution with high error, or in other words, the *maximum* across all probability distributions (of the above sum) will have high error. We can then lower bound the maximum by replacing it with the average, hence summing over probability distributions after all. To be precise, let $H' = \{f_1, \ldots, f_T\}$ (we have $T = 2^{2m}$ but this is not important) so that $D = \{D_{f_1}, \ldots, D_{f_T}\}$, then

$$\max_{D \in D} \left(\frac{1}{k}\sum_{i=1}^{k} \ell(A(S_i^{D}))\right) \geq \frac{1}{T}\sum_{j=1}^{T} \frac{1}{k}\sum_{i=1}^{k} \left(\ell(A(S_i^{D_{f_j}}))\right)$$

We can now exchange the order of these to sums so that we finally get to sum over all possible probability distributions. Then, we replace the right sum (across all training sequences) with the *minimum* across all training sequences, because in fact the training sequence does not matter. For each training sequence, we have that for each of the m points outside the sequence, exactly half of all probability distributions will label that point 0 and the other half 1. Now the learner *cannot* differentiate between them, no matter how it is defined, because they have lead to the identical training sequence, and thus it gets it right in only half of all cases. Alas, it only gets half of all elements outside the training sequence right in expectation, and thus at most three quarters of all elements total. (For the remaining details, see pages 62/63.)

Now, this leads to an expected error of at least $\frac{1}{4}$. However, instead of choosing a set C of size 2m, we could more generally choose a set C of size $b \cdot m$. With an analogous construction, we can show that the learner might still get the m elements in the training sequence right, but will merely guess on the other $(b-1) \cdot m$, and its expected error ends up as $\frac{1}{2}\frac{b-1}{b}$. Thus, by increasing b we can push the error arbitrarily close toward $\frac{1}{2}$. In the setting of binary classification, this is the error achieved by blindly guessing, and so it is fair to say that no meaningful lower bound of quality can be achieved without making assumptions about the nature of the learning task. In particular, such assumptions are the basis of generalization: the idea that a papaya which is surrounded by similar papayas, all of which tasty, is probably tasty itself – that idea is an assumption about how we can generalize; it would not be true if we took all labeling functions as equally likely candidates for the true function. In fact, having such a dense hypothesis class makes generalization impossible, as we've just demonstrated. This fact is also key for the upcoming chapter.

The VC-Dimension

The ***VC-dimension*** is a concept that falls right out of the above line of thinking. You might recall that every finite hypothesis class is learnable – but many infinite hypothesis classes are also learnable. For example, the class of all intervals (or more precisely, the set of all predictors h such that there exists an interval I with the property that $h(x) = 1 \iff x \in I$) over the domain $X = \mathbb{R}$ is learnable. (I won't give a proof because we're about to get a much more general result, but it's probably not surprising – if you want, stop reading and think about it.) This suggests that the decisive property for learnability is not the size of the hypothesis class, but rather *some kind of measure for how easy it would be to construct a no-free-lunch style example*.

The key piece in the preceding proof is the same idea I've emphasized before: if all possible functions on a finite set are possible, then learning about the value of the true function on some points is zero information about its value on the other points. How does that work in the case of intervals? Can we construct a set of points such that all labeling functions are possible? Note that the important question is not whether the environment distribution D can label them in every possible combination – the answer to this is likely affirmative since the D need not be deterministic. Rather, the question is whether there are predictors *in our class* – i.e., interval predictors – which allow every combination of labels on any particular set. This is so because PAC learnability only implies that we can get close to the best classifier *in our hypothesis class*, so in order for a no free lunch type argument to go through, it has to be impossible to obtain information about this predictor (which is in our class).

So is this possible with intervals? Let's start with a 1-point set:

Success! We have found interval predictors that give every possible labeling combination (i.e. (1) and (0)) on our one domain point $x \in X$. (To be clear, the black and red x in the image are meant to denote the same element; we drew two possible scenarios of where the interval might be with regard to that element). We thus say that our one-point set is ***shattered*** by the class of interval predictors – a bad thing as it means it's harder to learn.

What about a two-point set?

Indeed, we see that our hypothesis class is dense enough to shatter 2-point sets, since all four sequences of labels (namely (11), (10), (01), (00)) are possible. Now, the training sequence is of course a *sequence* and could therefore contain the same point twice. In that case, this two-point sequence could not be shattered. However, we actually care about the existential quantifier rather than the universal quantifier here. The existence of a single sequence of size two that is shattered will be important for our upcoming definition. (Also, the cases where the same point is included more than once are uninteresting anyway because those sequences can trivially never be shattered.) This is why we go about defining this new concept in terms of sets rather than sequences.

What about three-point sets? Take a moment to think about it: there are eight possible labeling combinations. Are there eight different interval classifiers that will produce all eight different labels?

The answer is no – if we label the three points x and y and z such that $x < y < z$, any interval containing both x and z must also contain y, which makes the labeling $(1, 0, 1)$ impossible (although all 7 other combinations are possible). No set of size three can be shattered by our class.

This proves that *the VC-dimension of our class is 2*, because the VC-dimension is defined simply as $\mathrm{VC}(H) := \max\{n \in \mathbb{N} \mid \exists C \subseteq X : |C| = n \wedge H \text{ shatters } C\}$. Since we found *some* set of size 2 that is shattered and proved that *no* set of size 3 is shattered, the largest integer such that a set of that size can be shattered is 2.

(If no maximum exists – i.e. if for any $n \in \mathbb{N}$, there exists a set of size n that is shattered – the VC-dimension is ∞.)

How useful is this concept of VC-dimension? For one, *any class with VC-dimension ∞ is not PAC learnable*. Given such a hypothesis class and any learner A, for each $m \in \mathbb{N}$, one can find a set C of size 2m and a probability distribution D such that the argument used in the No-Free-Theorem goes through, thus establishing a minimum gap between the performance of the optimal classifier and the classifier output by A, thus contradicting the definition of PAC learning. (To be more precise, we first prove that, if the expected error is at least $\frac{1}{4}$, then there must be at least a $\frac{1}{8}$ chance to have an error of at least $\frac{1}{7}$, which then gives us a direct contradiction with the definition of PAC learning. This proof of this is easy – prove the contrapositive.)

The fact that this contradicts PAC learnability depends on the precise definition of PAC learnability. In particular, if the sample complexity were allowed to depend on the probability distribution, the No-Free Lunch argument would no longer work. The third post will discuss two alternative notions of learnability, where the sample complexity is allowed to depend on further elements.

What about classes with finite VC-dimension? Here, the definition of VC-dimension may seem uninformative on first glance – only sets of size five are shattered, but perhaps for any m there exist sets of size m that are almost shattered? However, it turns out this is not possible. One can prove that, for any $m \in \mathbb{N}$ and any subset $C \subset X$ with $|C| = m$, the number of functions in H_C (where $H_C := \{h_{|C} \mid h \in H\}$) is equal to the number of subsets of C that are shattered by H. Since, if the VC-dimension of H is d for some $d \in \mathbb{N}$, only subsets of size at most d can be shattered, and since there are only polynomially many subsets of size at most d, this implies that the size of H_C can be bounded by a function polynomial in $|C|$. This is a pretty strong result, given that there are $2^{|C|}$ possible classifiers for C.

No proof for this result, as I didn't find reading it very instructive. The same will be true for most of the upcoming result.

The Fundamental Theorem of Statistical Learning

The ***fundamental theorem of statistical learning*** (in slightly simplified form) states that, if H is some hypothesis class for any domain set X and target set $Y = \{0, 1\}$, the following statements are equivalent:

1. H has the uniform convergence property
2. A_{ERM} is a successful PAC learner for H
3. H is PAC learnable
4. H has finite VC-dimension

We've proved $1 \implies 2$ in the first section of this post. $2 \implies 3$ is trivial. We've sketched the proof for $3 \implies 4$ in the previous section, by arguing that $\neg 4 \implies \neg 3$. The difficult-and-very-technical-proof-I-will-not-include is the one for $4 \implies 1$. Those four taken together make it into a circle, thus establishing equivalence.

There is also a quantitative version which upper- and lower bounds the sample complexity for PAC learnability and uniform convergence.

As nice as this result is, it is also important to understand that it says nothing whatsoever about runtime. As we've seen, the class of all predictors such that there exists a C program with at most 10000 symbols implementing the predictor is PAC learnable, but actually implementing A_{ERM} is completely infeasible.

Understanding Machine Learning (III)

This is the final part of a 3-part series on the fundamentals of Machine Learning as defined by this book (although it will transition into a longer sequence about the remaining, more practical part of the book). If you're interested, begin here.

Nonuniform Learning

Recall the definition of PAC learnability:

> A hypothesis class H is called PAC learnable iff there exists a learner A and a function $m^* : (0,1)^2 \to \mathbb{N}$ such that, for any probability distribution D over $X \times Y$, any gap $\epsilon \in (0,1)$, and any failure probability $\delta \in (0,1)$, if the training sequence S consists of at least $m^*(\epsilon, \delta)$ i.i.d. examples sampled via D, then with probability at least $1 - \delta$ over the choice of S, it holds that $\ell(A(S)) \leq \ell(h^*) + \epsilon$ for all $h^* \in \text{argmin}_{h \in H}[\ell(h)]$.

One way to weaken this definition so that it applies to a wider range of hypothesis classes is to allow the function m^* which upper-bounds the sample complexity to also depend on the hypothesis we are competing with – so rather than upper-bounding the difference of A(S) and h^*, we upper-bound the difference of A(S) and h' separately for each $h' \in H$. We have the following formal definition:

> A hypothesis class H is called **nonuniformly learnable** iff there exists a learner A and a function $z^* : (0,1)^2 \times H \to \mathbb{N}$ such that, for any probability distribution D over $X \times Y$, any gap $\epsilon \in (0,1)$, any failure probability $\delta \in (0,1)$, and any hypothesis $h \in H$, if the training sequence S consists of at least $z^*(\epsilon, \delta, h)$ i.i.d. examples sampled via D, then with probability at least $1 - \delta$ over the choice of S, it holds that $\ell(A(S)) \leq \ell(h) + \epsilon$.

(There is also a second relaxation, where the sample complexity is allowed to depend on the probability distribution D in addition to the hypothesis and the gap and failure probability. It is called **consistency**, but it turns out to be too weak to be useful. One can prove that every countable hypothesis class is consistent, and that A_{memorize} is a successful "learner.")

For an example of a hypothesis class that is nonuniformly learnable but not PAC learnable, consider the domain $X = \mathbb{R}$, the usual binary label class $Y = \{0, 1\}$, and the hypothesis class H of all ***polynomial classifiers*** – that is, all classifiers h such that there exists a polynomial function p such that $h(x) = 1 \iff p(x) \geq 0$ for all $x \in X$. Arguing with the VC-dimension, one can see that class of polynomial classifiers of degree at most n is PAC learnable (even though it's infinite), and that the entire class of polynomial classifiers is not PAC learnable. While this class is nonuniformly learnable (this will follow from what we prove this chapter), A_{ERM} is no longer a successful learner. This fact is easy to verify: given any finite amount of points, A_{ERM} would always choose a polynomial of

sufficiently high degree in order to classify all of them correctly. As long as no two points are exactly the same, this is always doable – for example, one can choose the polynomial $-(1) \cdot (x - p_1)^2 \cdot (x - p_2)^2 \cdots$ which will be 0 at every p_i and negative everywhere else. You might notice that this is precisely the argument that proves that the class has infinite VC-dimension. It is also a classic example of overfitting – it is not at all going to lead to a small estimation error. In fact, this implementation would mean that A_{ERM} behaves exactly like $A_{memorize}$. A more sophisticated learning algorithm is needed.

Suppose we have such a *nonuniform learner* $A_{???}$. This will mean that, if we fix some distribution D generating the points $x \in X$, then for any one hypothesis h, we can bound the number of sample points we need such that the predictor $A_{???}(S)$ will compete well with the true error of h.

The learner $A_{???}$ will have to be biased toward lower degree polynomials so as not to behave like A_{ERM}. Therefore, if h is a predictor corresponding to a polynomial with small degree, we probably won't need many samples to compete with it. On the other hand, if h comes from a polynomial of degree 50 and the distribution D is such that h is an excellent predictor, we will need a very large number of sample points before $A_{???}$ is convinced enough to seriously consider high-degree polynomials and outputs something almost as good as h. As the degree of polynomials we're competing with goes up, the number of sample points we need will go up, and therefore there is no *uniform* function (namely uniform across all other hypotheses) that can bound the sample complexity.

If D is such that the optimal classifier corresponds to a polynomial of small degree, this will not be an issue – then, while we don't get the theoretical assurance of being able to compete with classifiers from high-degree polynomials, they will, in fact, have large real error (although excellent empirical error), and we will compete with every hypothesis in our class fairly quickly. In that case, the theoretical bound will be pessimistic. However, recall that nonuniform learning is about guarantees that hold for all distributions D simultaneously.

I've previously stated that there are more sophisticated approaches to the evidence-vs-priors problem than to commit to a hypothesis class ahead of time. The learner $A_{???}$ will be an example.

Now recall the definition of uniform convergence:

> A hypothesis class H has the uniform convergence property iff there is a function $u^* : (0,1)^2 \to N$ such that, for every joint probability distribution D over $X \times Y$, every gap $\epsilon \in (0,1)$ and every confidence level $\delta \in (0,1)$, if $|S| \geq u^*(\epsilon, \delta)$, then with probability at least $1 - \delta$ over the choice of S, the inequality $|\ell_S(h) - \ell(h)| \leq \epsilon$ holds for all $h \in H$.

Notice that we called the three bounding functions m^* in case of PAC learnability (for sa**m**ple complexity), u^* (for **u**niform convergence) and z^* for nonuniform learning. Notice also that, unlike the other two, uniform convergence bounds the difference between real and empirical error in both directions (this will become important in a bit).

We've found two properties which are equivalent to PAC learnability: the uniform convergence property and having a finite VC-dimension. We now introduce a property which is equivalent to *nonuniform learnability*, namely

A hypothesis class H is nonuniformly learnable iff there exists a family $\{H_i\}_{i\in N}$ of hypothesis classes, each of which PAC learnable, such that $H = \bigcup_{i\in N} H_i$.

There are two directions to prove in order to establish this result; we will prove both of them. The first one is pretty easy.

" $\Longrightarrow$ " Suppose H is nonuniformly learnable. Let z^* be the function from the definition. For each $n \in N$, set $H_n := \{h \in H \mid z^*(\epsilon, \delta, h) \leq n\}$. Then, $H = \bigcup_{n\in N} H_n$, since for all $h \in H$, we have $h \in H_{z^*(\epsilon,\delta,h)}$. Moreover, given any $n \in N$, if H_n had infinite VC-dimension, we would find a probability distribution D over $X \times Y$ such that the expected value of $\ell(A(S)) - \min_{h\in H}\{\ell(h)\}$ is at least $\frac{1}{4}$. (Apply the No-Free-Lunch argument.) In particular, there would be at least a $\frac{1}{8}$ chance that $\ell(A(S)) - \min_{h\in H}\{\ell(h)\} \geq \frac{1}{4} > \frac{1}{8}$, contradicting the fact that $z^*(\epsilon, \delta, h) \leq n$ for all $h \in H_n$. Therefore, H_n has finite VC-dimension and is thus PAC learnable by the fundamental theorem of statistical learning.

The interesting thing about this proof is that getting both the gap and the failure probability to $\frac{1}{8}$ indirectly implies that they can also go arbitrarily low.

The reverse direction is good deal harder. We shall construct a new learner A_{SRM} that will be our $A_{???}$. The abbreviation SRM stands for ***Structural Risk Minimization***, and the idea is that we take into account both how well a hypothesis performs on the training data (like ERM does) and how easily we can bound the estimation error of its hypothesis class.

Structural Risk Minimization

To begin, we suppose that we are given a family $\{H_n\}_{n\in N}$ of hypothesis classes, each of which PAC learnable. Since each is PAC learnable, each also has the uniform convergence property, and we can find a family of functions $\{u_n^*\}_{n\in N}$ such that u_n^* bounds the sample complexity for class H_n. Among our infinitely many classes, we we will need to prioritize some over others. To do this, we define a ***weighting function*** $w : N \to (0, 1)$; i.e. a function with the property that $\sum_{i\in N} w(i) \leq 1$. The obvious choice here is $w(i) = 2^{-i}$, although many other choices are also possible; it will not matter for the theoretical construction.

To achieve a global guarantee, we shall query each class H_n for some "local" guarantee in the form of some ϵ' and δ'. The gap will not be the same for each class, so we define a function $\epsilon_n : (0, 1) \times N \to (0, 1)$ that, given a certainty δ' and a sample size $m \in N$, will return the best (i.e. lowest) gap between real and empirical error that H_n can guarantee for δ' and m. In symbols, we set $\epsilon_n(\delta, m) := \min\{\epsilon \in (0, 1) \mid u_n^*(\epsilon, \delta) \leq m\}$.

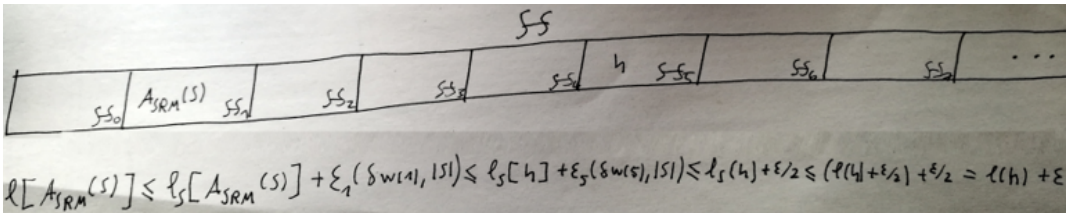
We want A_{SRM} to satisfy the definition of nonuniform convergence, so we assume that we are given two values $\epsilon, \delta \in (0, 1)$ as well as a training sequence S. Now in order to implement A_{SRM}, we could,

for each $n \in N$, compute the gap which H_n guarantees with failure probability δ, namely $\epsilon_n(\delta, |S|)$, as well as the minimal empirical error achievable with that class, namely $\min\{\ell_S(h) \mid h \in H_n\}$. Then, we could output the hypothesis that minimizes the sum of these two values (across all of $\bigcup_{n\in N} H_n$). However, this approach will not allow a proof that we satisfy the definition of nonuniformly learning because we will not be able to upper-bound our risk of failure by δ. Rather, we can upper-bound it by δ *for each class,* but then then only one of these infinitely many guarantees has to fail for our uniform upper-bound to fail. (Despite its name, the δ in the definition of nonuniform learning is very much uniform, i.e., it applies to the entire class $\bigcup_{n\in N} H_n$.)

To remedy this, we will demand successively stronger safety guarantees, and we use this as our mechanism to privilege earlier classes. Namely, we do it as explained above except that we compute the gap which H_n can guarantee with safety $1 - \delta \cdot w(n)$ rather than just $1 - \delta$, i.e., the value $\epsilon_n(\delta \cdot w(n), |S|)$. Clearly $\delta \cdot w(n)$ is a lot smaller than δ, so we get more confident bounds (and particularly confident bounds for more distant classes). Then we still compute the empirical risk, add both numbers, and return the global minimum. To write it down as a formula, A_{SRM} will output a hypothesis in the set $\operatorname{argmin}_{h\in H}\{\ell_S(h) + \epsilon_n(\delta \cdot w(n), |S|)\}$.

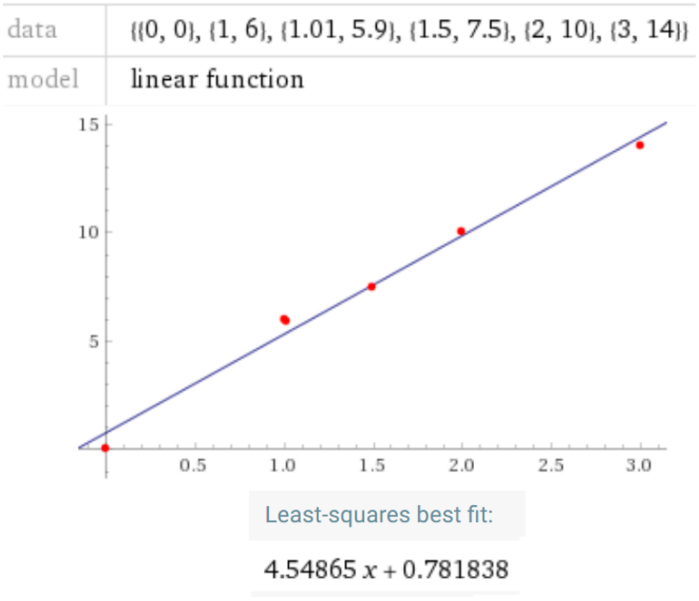
With these progressively more confident bounds, the probability that they all hold is $\prod_{i=0}^{\infty}(1 - \delta w_n) \geq 1 - \sum_{i=0}^{\infty} \delta w_n = 1 - \delta \sum_{i=0}^{\infty} w_n \geq 1 - \delta$.

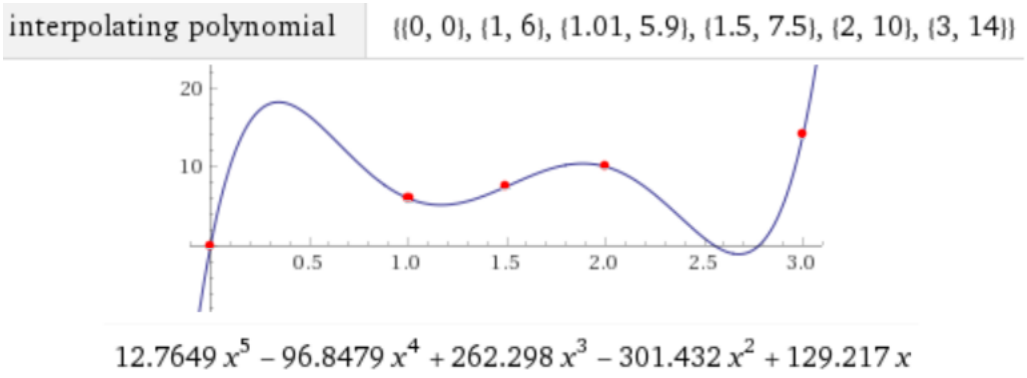
With the definition in place, we can prove $\Longleftarrow$ by showing that A_{SRM} as constructed above is a nonuniform learner of the class H. For given $\epsilon, \delta \in (0, 1)$ and $h \in H$, if we set n to be the index of the (first) class that contains the predictor h, i.e., $n := \min\{n \in N \mid h \in H_n\}$, then we can define $z^*(\epsilon, \delta, h) := u_n^*(\epsilon, \delta \cdot w(n))$, which will guarantee a gap between $\ell(A_{SRM}(S))$ and $\ell(h)$ of at most ϵ by the following hand-written proof:



The first inequality holds because it simply writes out the gap which ϵ_1 guarantees. The second inequality holds because this is how we defined A_{SRM}. The third inequality holds because $|S| \geq u_n^*(\epsilon, \delta \cdot w(n))$ and $\epsilon_n(\delta \cdot w(n), u_n^*(\epsilon, \delta \cdot w(n))) = \epsilon$ (make sure you understand why). In our case, we have that $n = 5$. And the fourth inequality – recall that uniform convergence bounds the difference between empirical and real error in both directions, i.e. it also holds that $\ell_S(h) \leq \ell(h) + \epsilon_5(\delta \cdot w(n), |S|)$. Now the second summand comes out ϵ for the same reason as above.

Note that, in the final (fourth) inequality, we bound the empirical error via the real error plus some maximum bound. This is necessary for the proof, but unlikely to make sense for a real instance – the hypothesis h is in a fairly "far out" class, so it is presumably fairly complex, and likely to overfit the data, i.e., have low empirical error but high real error. Whenever the best predictor is in an early class, all of the later classes are likely to perform poorly in the real world (although they will often perform better on the training data, at least if the classes are actually ordered by increasing complexity). At least this is so in the case of polynomial predictors.





Minimum Description Length

Suppose we are interested in learning a countable hypothesis class H. Without any other requirements, H is nonuniformly learnable, being the countable collection of singleton classes, i.e. $H = \{h_0, h_1, \dots\} = \bigcup_{n \in \mathbb{N}} \{h_n\}$. The idea now is that we can use a "natural" weighting function which is induced by the ***minimum description length*** of each predictor in the class. Formally, we define a description language as a subset $L \subset \{0,1\}^*$. If the description language is ***prefix-free***, i.e. if there

are no two descriptions $x, y \in L$ such that x is a prefix of y, we can define a weighting function $w : N \to (0, 1)$ by $w(h_n) = 2^{-|d(h_n)|}$, where $d(h_n) \in L$ is the description of h_n. To see that this defines a proper weighting function – i.e. that the total sum of weights equals at most 1 – note that can view the set of all descriptions as a probability space, i.e. $\Omega = \{d(h_n)\}_{n \in N}$, and assign each description a probability. Specifically, consider the following experiment:

> Repeatedly throw a coin; whenever it lands heads, write down a 1, whenever it lands tails, write down a 0. Stop when the output string equals some element in L.

Since this language is prefix-free, each element $\ell \in L$ is assigned a probability in this way – and as we know, probabilities sum up to at most 1. Then since $w = Pr$, i.e. the function w just assigns these probabilities as weights, this shows that it is a proper weighting function.

This leads to a natural definition of an alternative learner A_{MDL}, which is simply following the SRM rule while using w as the weighting function. This construction is closely related to Occam's Razor and Kolmogorov Complexity.

Computational Complexity of Learning

A field with fairly strong similarity to Machine Learning is *statistics*, because both are, in some sense, about learning from data. But in addition to theoretical limits, Machine Learning also has to concern itself with the annoying reality of bounded runtime. Thus, even though we've previously pretended that the behavior of A_{ERM} and A_{SRM} is fully determined by their subscript, this is not really so: the subscript is merely a mission statement, and how this is actually achieved has been left unsaid. Different approaches may have drastically different runtimes in practice.

Instead of reinventing the wheel, we will borrow the relevant ideas from the existing literature on *computational complexity theory.*

Computational Complexity Theory: the Landau notation

One generally cannot make precise statements about the runtime of an algorithm, because it depends on the processor speed of the machine it runs on (a problem which persists even if the algorithm is fully specified). Instead, one uses "asymptotic" bounds. Let's demonstrate how this works with an example.

Consider the selection-sort algorithm. It takes as input a sequence of points $(x_i)_{i \in \{1,\dots,n\}}$ such that $x_i \in X$ for all i and there is a total order $<$ on X (so given any two elements, we can say which one is larger). It goes through the list, remembers the smallest element, puts it at the beginning, goes through the list again (this time starting with the second element), remembers the smallest element, and puts it at the second spot, then goes through the list again (this time starting with the third element), puts the smallest element of those in third place and so on. It's the slowest not-intentionally-bad sorting algorithm that I know of, but it's also the simplest, and it's probably a decent description of how a human would sort a list.

So how long does selection-sort need to sort a list of n elements? Well, we sort of need n operations to find the smallest element, then $n - 1$ for the second smallest and so on, leading to $\sum_{i=1}^{n} i = n \cdot \frac{n+1}{2}$.

And then some operations to swap elements, but it's not obvious how many. And maybe there's some overhead, too.

To have something rigorous, we would like to say that it's around (or actually, *not significantly larger than*) n^2 in a principled way. The approach taken here is the O notation or *Landau notation*. Let s be the function describing the runtime of selection-sort as a function of the input size of the list on any machine, and let $f : \mathbb{N} \to \mathbb{N}$ be given by $f(n) = n^2$, where f outputs time in seconds. Then we want to write that $s \in O(f)$ to say exactly that "s is not significantly larger than f", even though we don't know exactly how large s gets (because we don't know for which machine it measures the runtime). The formal definition goes like this:

$$s \in O(f) \iff \exists C \in \mathbb{R}, n_0 \in \mathbb{N} \text{ such that } \forall n \in \mathbb{N}_{\geq n_0} : s(n) \leq C \cdot f(n)$$

So the idea is that, regardless of how slow the machine is that s is measuring runtime for, there should be some C such that every operation on this machine takes at most C seconds. Obviously, for any modern machine, many thousands of operations might be performed in one second, so $C < 0.001$ is quite likely to do the job. But even if we had declared that f measures nano-seconds instead, there would still always be such a C; one could just take the current one and multiply it by 10^9. Thus, if such a C exists, then s should be bounded by C times f. Furthermore, there might be some constant overhead, so we don't demand that this is true for all input values, only all input values from some point onward.

One often sees $f \in O(n^2)$ as shorthand for [$f \in O(g)$ where $g(n) := n^2$]. One also often sees $f = O(n^2)$ rather than $f \in O(n^2)$ even though that notation is nonsensical (function equals set of functions?) and not even shorter. Far worse still is the inexplicable $f(n) = O(g(n))$ which offends me on a remarkably deep level.

To apply these ideas to machine learning, we will need to make three adjustments. 1) we need functions that take real numbers as input, rather than natural numbers. 2) we will need to compare their output as their parameter goes to 0, rather than as their parameter grows. And 3), we will need functions of several variables.

The first two are straight-forward. For $s, f : (0, 1) \to \mathbb{N}$, one can simply alter the definition into

$$s \in O(f) \iff \exists C \in \mathbb{R}, r_0 \in (0, 1) \text{ s.t. } \forall r \in (0, 1) : (r < r_0 \implies s(r) \leq C \cdot f(r))$$

The third one appears to raise at least one question – if r turns to a vector $r = (r_1, \ldots, r_d)$, then what does the condition "$r < r_0$" turn into? The answer is "$r_i < r_0$ for some $i \in \{1, \ldots, d\}$", so it will be enough if one of the coordinates gets small.

Last words on Landau in general: since $f \in O(g)$ for $f(n) = 3n$ and $g(n) = 2^{2^n}$, this notation is fairly imprecise. To have something that bounds runtime in both directions, one can use the symbol Θ which is defined as$f \in \Theta(g) :\iff f \in O(g) \wedge g \in O(f)$,

Applying Landau to Machine Learning tasks

In the context of Machine Learning, one has the problem that there is no obvious input size for a learning task. The size of the training set is a non-starter because having more training data makes a problem easier, not harder. What one does instead is to use the gap and confidence as parameters. We have the following formal definition:

> Given a function $f : (0,1) \to \mathbb{N}$, a learning task defined via X and Y and H and ℓ, and a learner A, we say that A solves the task in time $O(f)$ iff there exists a constant $C \in \mathbb{R}$ such that, for all $\epsilon, \delta \in (0,1)$, if A is given a randomly sampled training sequence S of sufficient length, then
>
> 1. A always puts out a predictor $h := A(S)$ in at most $C \cdot f(\epsilon, \delta)$ seconds
> 2. Whenever h is given any domain point $x \in X$, it puts out a label $y \in Y$ in at most $C \cdot f(\epsilon, \delta)$ seconds
> 3. The equation $\ell(h) \leq \min_{h' \in H}[\ell_S(h')] + \epsilon$ holds with probability at least $1 - \delta$ over the choice of S

The first condition is the logical place we've been working toward: it says that A can't take too long to output a predictor. The second condition makes sure that A doesn't cheat – if it wasn't for this rule, then A could offload all computational work to the predictor. To be precise, we could define $A_{ERM\_cheating}$ as "memorize the training sequence, then output the predictor h that, upon receiving a domain point $x \in X$, runs the code of A_{ERM_honest}, applies that to x and returns its output". Clearly, $A_{ERM\_cheating}$ has excellent runtime, but it's probably not a very useful way to go about defining a predictor. Alas, with the predictor itself being bound by the same term as the learner, this strategy no longer helps. (Even if it could somehow offload half of all work to the predictor, taking half as long doesn't change the complexity class.)

Finally, the third condition is just the usual PAC condition.

There are some cases in which A_{ERM} can be implemented with a reasonable runtime complexity. But usually it can't, and that's why this is only the end of part I and not of the book. There are many different hypothesis classes that are of interest, and the ability to implement A_{ERM} efficiently on them is generally desirable.

UML IV: Linear Predictors

(This is the fourth post in a series on Machine Learning based on this book. Click here for part one. If you have some background knowledge, this post might work as a stand-alone read.)

The mission statement for this post is simple: we wish to study the class of ***linear predictors***. There are linear correlations out there one might wish to learn; also linear stuff tends to be both efficient and simple, so they may be a reasonable choice even if the real world is not quite linear. One can also build more sophisticated classifiers by using linear predictors as building blocks, but not in this post).

In school, a function is linear iff you can write it a $f(x) = ax + c$. In higher math, a function $f : X \to Y$ is linear iff $f(x + y) = f(x) + f(y)$ for all $a, b \in X$. In the the case of $f : R^d \to R$, this condition holds iff f it can be written as $f(x) = \sum_{i=1}^{d} a_i x_i$ for some parameter vector $a \in R^d$. So the requirement is stronger – we do not allow the constant term the school definition has – but one also considers higher dimensional cases. The case where we do allow a constant factor is called affine-linear, and we also say that a function is homogeneous iff , which (in the case of affine-linear functions) is true iff there is no nonzero constant factor.

In Machine Learning, the difference between linear and affine-linear is not as big of a deal as it is in other fields, so we speak of *linear predictors* while generally allowing the inhomogeneous case. Maybe Machine Learning is more like school than like university math ;)

For $X = R^d$ and some $Y \subseteq R$, a class of linear predictors can be written like so:

$$L_{d,\phi} = \{ h : x \mapsto \phi(\langle a , x \rangle + c) \mid a \in R^d , c \in R \}$$

Let's unpack this. Why do we have an inner-product $\langle \cdot \rangle$ here? Well, because any function $f : x \mapsto \sum_{i=1}^{d} a_i x_i$ can be equivalently written as $f : x \mapsto \langle a, x \rangle$, where $a := (a_1, \ldots, a_d)$. The inner-product notation is a bit more compact, so we will prefer it over writing a sum. Also note that, for this post, bold letters mean "this is a vector" while normal letters mean "this is a scalar". Secondly, what's up with the ϕ? Well, the reason here is that we want to catch a bunch of classes at once. There is the class of ***binary linear classifiers*** where $Y = \{-1, 1\}$ but also the class of ***linear regression predictors*** where $Y = R$. (Despite what this sequence has looked like thus far, Machine Learning is not just about binary classification.) We get both of them by changing ϕ. Concretely, we get the linear regression functions by setting $\phi = \phi_{sign} := 1_{R_+} - 1_{R_-}$, i.e., the function that sends all positive numbers to 1 and all negative numbers to -1. The notation 1_M for any set M denotes the *indicator function* that sends all elements in M to 1 and all others in its domain to 0.

For the sake of brevity, one wants to not write the constant term but still have it around, and to this end, one can equivalently write the class as

$$L_{n,\phi} = \{ h : x \mapsto \phi(\langle a', x' \rangle) \quad | \quad a' \in R^{d+1} \}$$

where it is implicit that x' = (x : 1). Then, the final part of the inner product will add the term $a_{d+1} \cdot x'_{d+1} = a_{d+1} \cdot 1 = a_{d+1}$, so a_{d+1} will take on the role of c in the previous definition.

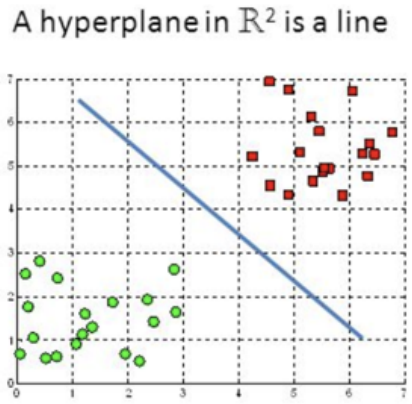
We still need to model how the environment generates points. For this section, we assume the simple setting of a probability function D over X only and a true function $f : X \to Y$ of the environment. We also need to define empirical loss functions for a training sequence $S \in (X \times Y)^*$. For binary classification, we can use the usual one that assigns h the number $\frac{1}{|S|}|\{(x, y) \in S \mid h(x) \neq y)\}|$. Since we will now look at more than one loss function, we will have to give it a more specific name than ℓ_S, so we will refer to it as ℓ_S^{0-1}, indicating that every element is either correctly or incorrectly classified. We call this the ***0-1 loss*** even though the label set is now $Y = \{-1, 1\}$.

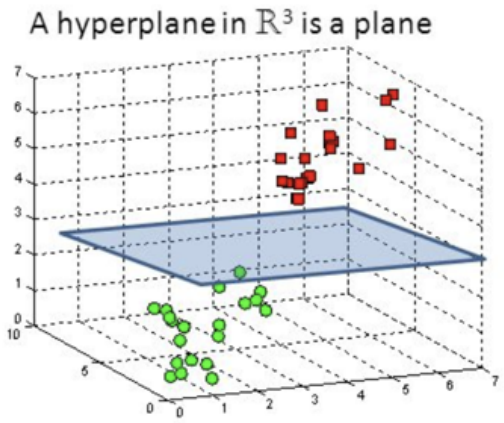
For regression (where Y = R), this is a poor function since hitting the precisely correct element in R is not a reasonable expectation – and if 3.8 is the correct answer, then $3.8 + 10^{-47}$ is a better guess than 17. Instead, we want to penalize the predictor based on *how far* it went off the mark. If $S = ((x_1, y_1), \ldots, (x_n, y_n))$, then we define the ***squared distance loss function*** as $\ell_S^{(2)}(h) := \sum_{i=1}^{n}(h(x_i) - y_i)^2$ and the ***absolute distance loss function*** as $\ell_S^{(1)}(h) := \sum_{i=1}^{n} |h(x_i) - y_i|$.

We begin with binary linear classification.

Binary Linear Classification

A binary linear classifier separates the entire space in two parts along a *hyperplane*. (A hyperplane in R^d is a $d - 1$ dimensional subspace.) This is quite easy to visualize. In the homogeneous case, the hyperplane will go through the origin, whereas in the inhomogeneous case, it may not.





Let's work out why this is so. A point x is sent to $\phi(\langle a', x' \rangle)$, do it gets classified positively iff $\langle a', x' \rangle$ is greater than 0. Suppose we're somewhere in the red area where this is not the case, and now we move into the direction of a'. At some point, we will be at a place where the inner product is exactly 0. Now, if we move into a direction orthogonal to a', the inner product doesn't change. This corresponds to the hyperplane that is visualized in the pictures above.

In linear classification problems, we say that a problem is ***separable*** iff there exists a vector a^* whose predictor gets all points in the training sequence right. In the book, this distinction is also frequently made for other learning tasks, where a problem is called ***realizable*** iff there exists a perfect predictor. For linear predictors, the space might be very high dimensional, which makes this assumption more plausible. As an example, suppose we model text documents as vectors, where there is one dimension for every possible term, and one sets the coordinate for the word "crepuscular" to the number of appearances of the word "crepuscular" in the document. With ≈ 171476 dimensions at hand, it might not be so surprising if a hyperplane classifies all domain points perfectly.

How do we train a linear classifier? The book discusses two different algorithms. For both, we shall first assume that we are in the homogeneous case. We start with the ***Perceptron*** algorithm.

The Perceptron algorithm

Since our classifier is completely determined by the choice of a, we will refer to it as h_a.

Thus $h_a(x) = \phi_{sign}(\langle a, x \rangle)$.

Recall that ha measures how *similar* a label point is to a and classifies it as 1 if it's similar enough (and as -1 otherwise). This leads to a very simple algorithm: we start with $a^0 = 0$; then at iteration t we take some pair (x, y) that is not classified correctly – i.e., where either

$\langle a^t, x\rangle > 0$ even though $y = -1$ or $\langle a^t, x\rangle < 0$ even though $y = 1$ – and we set $a^{t+1} := a^t + yx$.

If x was classified as -1 even though $y = 1$, then we add x, thereby making a^{t+1} more similar to x than a^t, and if x was classified as 1 even though $y = -1$, then we subtract x, thereby making a^{t+1} less similar to x than a^t. In both cases, our classifier updates into the right direction. And that's it; that's the perceptron algorithm.

It's not obvious that this will ever terminate – while updating towards one point will make the predictor better about that point, it might make it worse about other points. However, there is a theorem stating that it does, in fact, terminate provided the problem is separable. The proof is partially interesting but also technical, so I'll present a sketch that includes the key ideas but hides some of the details.

The main idea is highly non-trivial. We assume there is no point directly *on* the separating hyperplane, then we begin by choosing a vector $a*$ whose predictor classifies everything correctly (which exists because we assume the separable case) and also has scalar product at least 1 with every positive point (take one that satisfies the first condition and divide it by the smallest norm of any positively labelled point). Now we observe that the similarity between at and $a*$ increases as t increases. This is so because $\langle a^{t+1}, a^*\rangle = \langle a^t + y \cdot x, a^*\rangle = \langle a^t, a^*\rangle + y\langle x, a^*\rangle$, and the term $y\langle x, a^*\rangle$ is positive since a^* is by assumption such that $\langle x, a^*\rangle > 0 \iff y = 1$. This shows that $\langle a^t, a^*\rangle$ grows as t grows.

The proof then proceeds like this:

- Establish a lower bound on the growth rate of $\langle a^t, a^*\rangle$
- Establish an upper bound on the growth rate of $||a^t||$
- Observe that $||a^*||$ is a constant
- Observe that $\langle a_t, a^*\rangle \leq ||a^t|| \cdot ||a^*||$ must hold because it's the famous *Cauchy-Schwartz inequality*
- Conclude that, as a consequence of the four facts above, the term t can only grow for a limited number of iterations

We obtain a bound that depends on the norm of the smallest vector a^* such that $\langle a^*, x\rangle \geq 1$ for all domain points $x \in X$ and on the largest norm of any domain point. This bound might be good or it might not, depending on the case at hand. Of course, the algorithm may always finish much earlier than the bound suggests.

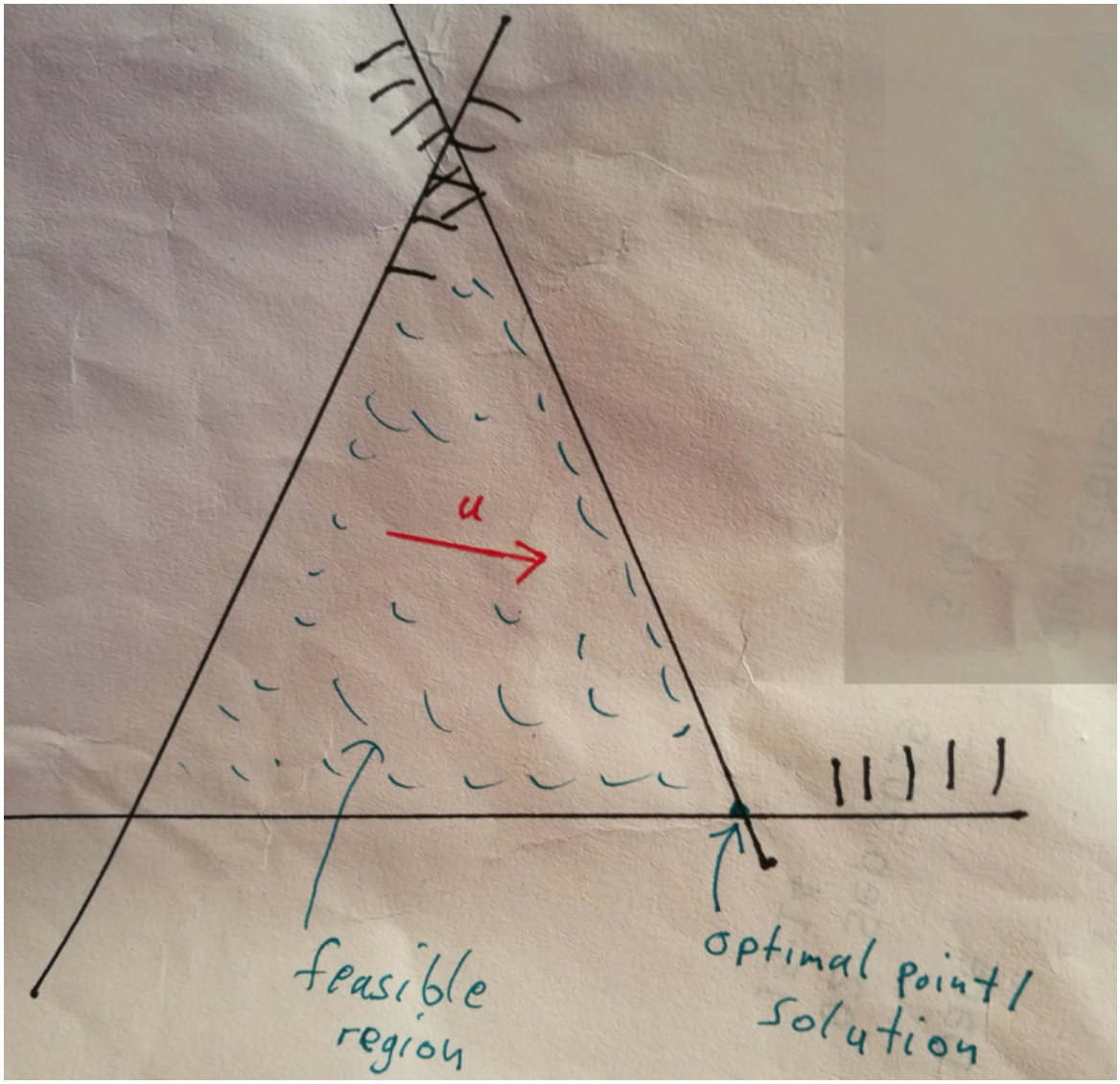
Linear Programming

Linear programming is an oddly chosen name for a problem of the following form:

$$\max_{x \in R^d} \quad \langle u, x \rangle \qquad \text{s.t.} \qquad A x \geq b$$

where $u \in R^d$ and $b \in R^n$ and $R^{n,d}$ are given. So we have a particular direction, given by u, and we want to go as far into this direction as possible; however, we are restricted by a set of *constraints* – namely the n many constraints that follow from the equation $Ax \geq b$. Each constraint is much like the predictor h from the previous section; it divides the entire space R^d into two parts along a hyperplane, and it only accepts points in one of the two halves. The set of points which are accepted by all constraints is called the *feasible region,* and the objective is to find the point in the feasible region that is farthest in the direction of u.

Here is a visualization for d=2 and n=3:



Once we hit the triangle's right side, we cannot go any further in precisely the direction of u, but going downward along that side is still worth it, because it's still "kind of" like u – or to be precise, if w is the vector leading downward along the rightmost side of the triangle, then

$\langle u, w \rangle > 0$, and therefore points which lie further in this direction have a larger inner product

with u. Consequently, the solution to the linear program is at the triangle's bottom-right corner.

The claim is now that finding a perfect predictor for a separable linear binary classification problem can be solved by a linear program. Why is this? Well for one, it needs to correctly classify some number of points, let's say n, so it needs to fulfil n conditions, which sounds similar to meeting n constraints. But we can be much more precise. So far, we have thought of the element a that determines the classifier as a vector and of all domain elements as points, but actually they are the same kind of element in the language of set theory. Thus, we can alternatively think of each domain element $x \in X$ as a vector and of our element a determining the classifier as a point. Under this perspective, each $x \in X$ splits the space in two halves along its own hyperplane, and the point a needs to lie in the correct half for all n hyperplanes) for it to classify all x's correctly. In other words, each $x \in X$ behaves exactly like a linear constraint.

Thus, if $S = ((x_1, y_1), \ldots, (x_n, y_n))$, then we can formulate our set of constraints as $Xa \geq 1$ (not ≥ 0 because we want to avoid points on the hyperplane), where

$$X = \begin{bmatrix} y_i x_1 \\ \vdots \\ y_n x_n \end{bmatrix}$$

i.e. the matrix whose row vectors are either the elements x_i (if $y_i = 1$) or the elements x_i scaled by -1 (if $y_i = -1$). The i-th coordinate of the vector Xa equals precisely $y_i \langle x_i, a \rangle$, and this is the term which is positive iff the point is classified correctly, because then $\langle x_i, a \rangle$ has the same sign as y_i.

We don't actually care where in the feasible region we land, so we can simply set some kind of meaningless direction like $u = 0$.

So we can indeed rather easily turn our classification problem into a linear program, at least in the separable case. The reason why this is of interest is that linear programs have been studied quite extensively and there are even free solvers online.

The inhomogeneous case

If we want to allow a constant term, we simply add a 1 at the end of every domain point, and search for a vector $a \in R^{d+1}$ that solves the homogeneous problem one dimension higher. This is why the difference of homogeneous vs inhomogeneous isn't a big deal.

Linear Regression

Linear regression is where we first need linear algebra and vector calculus.

Recall that, in linear regression, we have $X = R^d$ and $Y = R$ and a *predictor* h_a for some $a \in R^d$ is defined by the rule $h_a(x) = \langle a, x \rangle$. Finally, recall that the *empirical squared loss function* is defined as $\ell_S^{(2)}(h) = \frac{1}{n}\sum_{i=1}^{n}(h(x_i) - y_i)^2$. (We set $n := |S|$.) The $\frac{1}{n}$ is not going to change where the minimum is, so we can multiply with n to get rid of it; then the term we want to minimize looks like this:

$$n \cdot \ell_S^{(2)}(h_a) = \sum_{i=1}^{n}(\langle a, x_i \rangle - y_i)^2$$

In order to find the minimum, we have to take the derivative with regard to the vector a. For a fixed $i \in \{1, \ldots, n\}$, the summand is $(\langle a, x_i \rangle - y)^2$. Now the derivative of a scalar with respect to a vector is defined as

$$\frac{\partial y}{\partial a} := \left(\frac{\partial y}{\partial a_1} \cdots \frac{\partial y}{\partial a_d}\right)$$

so we focus on one coordinate $j \in \{1, \ldots, d\}$ and compute the derivative of the such a summand term with regard to a_j. By applying the chain rule, we obtain $2(\langle a, x_i \rangle - y_i) \cdot (x_i)_j$. This is just for one summand; for the entire sum we have $2\sum_{i=1}^{n}(\langle a, x_i \rangle - y_i)(x_i)_j$. So this is the j-th coordinate of a vector that needs to be zero everywhere. The entire vector then looks like this:

$$2\sum_{i=1}^{n}(\langle a, x_i \rangle - y_i)x_i.$$

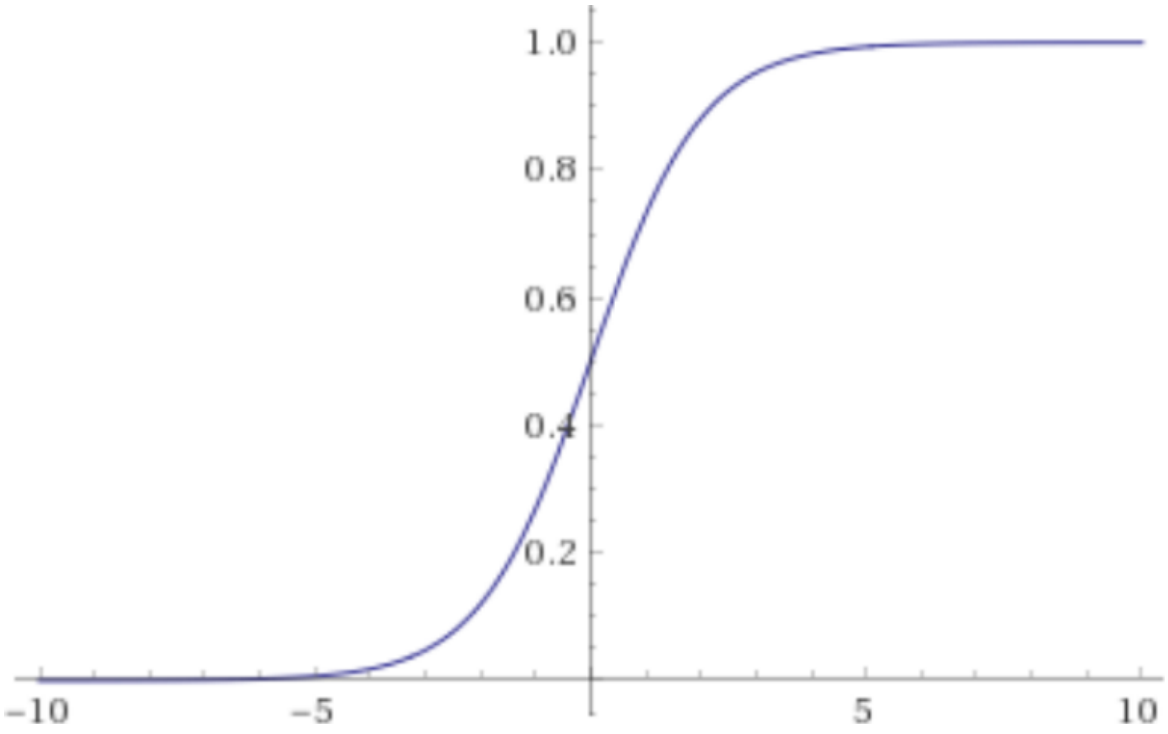
Now, through a process that I haven't yet been able to gain any intuition on, one can reformulate this as $XX^T a = b$, where $X = (x_1 \cdots x_n)$ is the matrix whose column vectors are the x_i, and $b = \sum_{i=1}^{n} y_i x_i$.

Now if XX^T is invertible the problem is easy; if not then it is still solvable, since one can prove that b is always in the range of XX^T (but I'll skip the proof). It helps that XX^T is symmetric.

Logistic Regression

Binary classification may be inelegant in sort of the same way that committing to a hypothesis class ahead of time is elegant – we restrict ourselves to a binary choice, and throw out all possibility of expressing uncertainty. The difference is that it may actually be desirable in the case of binary classification – if one just has to go with one of the labels, then we can't do any better. But quite often, knowing the degree of certainty might be useful. Moreover, even if one does just want to throw out all knowledge about the degree of certainty for the final classifier, including it might still be useful during training. A classifier that gets 10 points wrong, all of which firmly in the wrong camp, might be worse choice than a classifier which gets 11 points wrong, all of which near the boundary (because it is quite plausible that the first predictor just got "lucky" about its close calls and might actually perform worse in practice).

Thus, we would like to learn a hypothesis which, instead of outputting a *label,* outputs a *probability* that the label is 1, i.e. a hypothesis of the form h:X→Y where Y=[0,1]. For this, we need ϕ to be of the form $\phi : \mathrm{R} \to [0, 1]$, and it should be monotonically increasing. There are many plausible candidates; one of them is the *sigmoid function* ϕ_{sigmoid} defined by the rule $\phi_{\text{sigmoid}}(x) := \frac{1}{1+e^{-x}}$. Its plot looks like this:

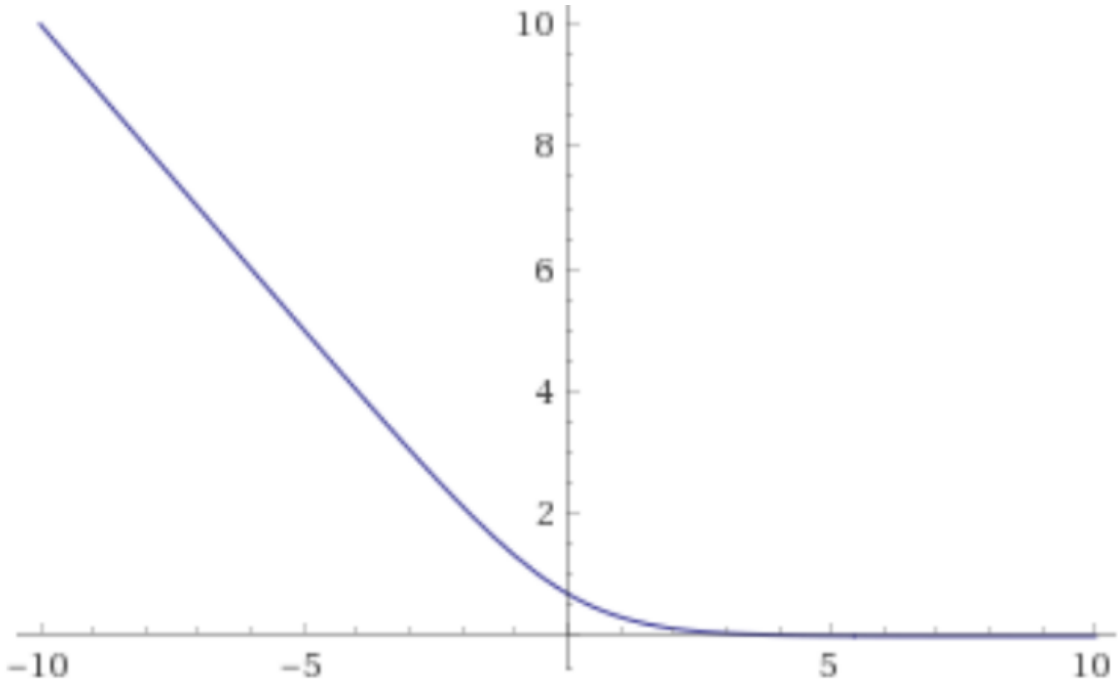


In practice, one could put a scalar in front of the $-x$ to adjust how confident our predictor will be.

How should our loss function be defined for logistic regression? In the case of y=1, we want to penalize probability mass, and in the case of $y = -1$, we want to penalize the *missing* probability mass to 1. Both is achieved by setting

$$\ell_S^{\text{logistic}}(h_a) := \sum_{j=1}^{n}(1 + e^{-y_j\langle x_j, a\rangle})$$

This is how it looks in the case of y=1:



And the case of y = −1 is symmetrical.

UML V: Convex Learning Problems

(This is part five in a sequence on Machine Learning based on this book. Click here for part 1.)

The first three posts of this sequence have defined PAC learning and established some theoretical results, problems (like overfitting), and limitations. While that is helpful, it doesn't actually answer the question of how to solve real problems (unless the brute force approach is viable). The first way in which we approach that question is to study *particular classes* of problems and prove that they are learnable. For example, in the previous post, we've looked (among other things) at *linear regression*, where the loss function has the form $\ell : R^d \to R$ and is linear. In this post, we focus on ***convex learning problems,*** where the loss function also has the above form and is convex.

Convexity

We begin with sets rather than functions.

Convex sets

> A set M (that is part of a vector space) is called *convex* iff for any two points in that set, the line segment which connects both points is a subset of M.

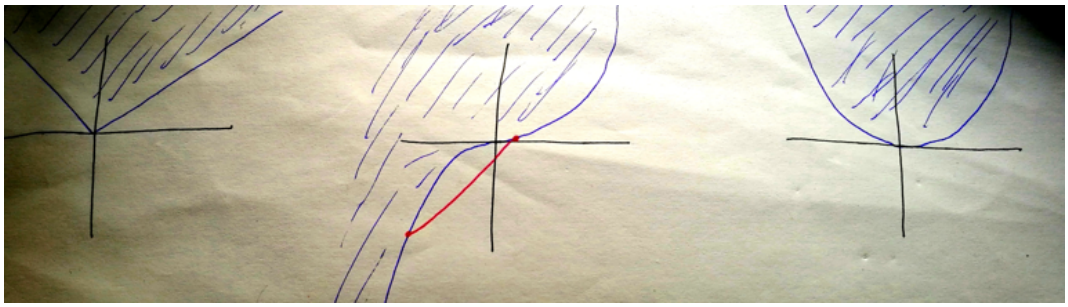
The condition that M is part of a vector space is missing in the book, but it is key to guarantee that a line between two points exists. Convexity cannot be defined for mere topological spaces, or even metric spaces. In our case, all of our sets will live in R^d for some $d \in N_+$.

As an example, we can look at letters as a subset of the plane. None of the letters in this font are quite convex – l and I are closest but not quite there. The only convex symbols in this post that I've noticed are . and ' and | and –.

Conversely, every regular filled polygon with n corners is convex. The circle is not convex (*no* two points have a line segment which is contained in the circle), but the disc (filled circle) is convex. The disc with an arbitrary set of points on its boundary (i.e. the circle) taken out remains convex. The disc with any other point taken out is not convex, neither is the disc with any additional point added. You get the idea. (To be precise on the last one, the mathematical description of the disc is $D = \{x \in R^2 \mid ||x|| \leq 1\}$, so there is no way to add a single point that somehow touches the boundary.)

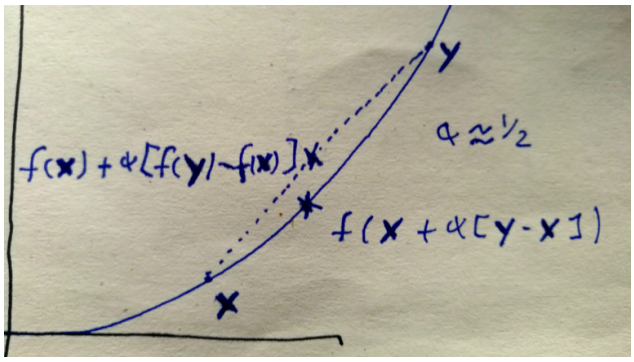
Convex functions

Informally, a function $f : R^d \to R$ is convex iff the set of all points *on and above the function* is convex as a subset of R^{d+1}, where the dependent variable goes up/downward.



(Here, the middle function (x^3) is not convex because the red line segment is not in the blue set, but the left ($|x|$) and the right (x^2) are convex.)

The formal definition is that a function $f : R^d \to R$ is convex iff for all $x, y \in R^d$, the equation $f(x + \alpha(y - x)) \leq f(x) + \alpha[f(y) - f(x)]$ holds for all $\alpha \in [0, 1]$. This says that a line segment connecting two points on the graph always lies above the graph.



If $d = 1$ as was the case in all of my pictures, then f is convex iff the little pixie flying along the function graph never turns rightward. This is the case iff f' is monotonically increasing, which is the case iff $f''(x) \geq 0$ for all $x \in R$.

The main reason why convexity is a desirable property is that, for a convex function, *every local minimum is a global minimum.* Here is a proof:

Suppose that $x \in R^d$ is a local minimum. Then we find some ball $B_d(x, \epsilon) := \{p \in R^d \mid ||p - x|| \leq \epsilon\}$ around x such that $f(y) \geq x$ for all y in the ball (this

is what it means to be a local minimum in R^d). Now let z be an arbitrary point in R^d; we show that its function value can't lie below that of x. Imagine the line segment from x to z. A part of it must lie in our ball, so we find some (small) $\delta \in R_+$ such that $x + \delta[z - x] \in B_d(x, \epsilon)$. Then (because x is our local minimum), we have that $f(x) \leq f(x + \delta[z - x])$. By convexity of f we have $f(x + \delta[z - x]) \leq f(x) + \delta[f(z) - f(x)]$, so taken together we obtain the equation

$$f(x) \leq f(x) + \delta[f(z) - f(x)]$$

Or equivalently $\delta[f(z) - f(x)] \geq 0$ which is to say that $\delta f(z) \geq \delta f(x)$ which is to say that $f(z) \geq f(x)$.

If there are several local minima, then there are several global minima, then one can draw a line segment between them that inevitably cannot go up or down (because otherwise one of the global minima wouldn't be a global minimum), so really there is just one global minimum. This is all about the difference between ≤ and <. The simplest example is a constant function – it is convex, and everywhere is a global minimum.

Jensen's Inequality

The key fact about convex functions, I would argue, is *Jensen's inequality:*

> Given $\alpha_1, \ldots, \alpha_n \in R_+$ with $\sum_{i=1}^{n} \alpha_i = 1$, if $f : R^d \to R$ is convex, then for any sequence $(x_1, \ldots, x_n) \in (R^d)^n$, it holds that $f(\sum_{i=1}^{n} \alpha_i x_i) \leq \sum_{i=1}^{n} \alpha_i f(x_i)$.

If you look at the inequality above, you might notice that it is almost the definition of linearity, except for the condition $\sum_{i=1}^{n} \alpha_i = 1$ and the fact that we have ≤ instead of =. So convex functions *fulfill the linearity property as an inequality rather than an equality* (almost). In particular, linear functions are convex. Conversely, concave functions (these are functions where the ≤ in the definition of convex functions is a ≥) also fulfill the above property as an inequality, only the the sign does again turn around. In particular, linear functions are concave. To refresh your memory, here is the definition of convexity:

$$f(x + \alpha(y - x)) \leq f(x) + \alpha[f(y) - f(x)] \quad \forall x, y \in X, \alpha \in [0, 1]$$

So to summarize: convex functions never turn rightward, concave functions never turn leftward, and the intersection of both does neither, i.e., always goes straight, i.e., is linear. Looking at convexity and concavity as a generalization of linearity might further motivate the concept.

Terms of the form $x + a(y - x)$, which one sees quite often (for example in defining points on a line segment), can be equivalently written as $(1 - a)x + ay$. I think the first form is more intuitive; however, the second one generalizes a bit better. We see that x and y are given weights, and those weights sum up to 1. If one goes from 2 weighted values to n weighted values (still all non-negative), one gets Jensen's inequality. Thus, the statement of Jensen's inequality is that if you take any number of points on the graph and construct a weighted mean, that resulting point still lies above the graph. See wikipedia's page for a simple proof via induction.

Guaranteeing learnability

Recall that we are trying to find useful solvable special cases of the setting *minimize a loss function of the form* $\ell : \mathrm{R}^d \to \mathrm{R}$. This can be divided into three tasks:

(1) define the special case

(2) demonstrate that this special case is indeed solvable

(3) apply the class as widely as possible

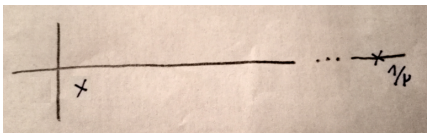
This chapter is about (1). (When I say "chapter" or "section", I'm referring to the level-1 and level-2 headlines *of this post* as visible in the navigation at the left.) The reason why we aren't already done with (1) is that convexity of the loss function alone turns out to be insufficient to guarantee PAC learnability. We'll discuss a counter-example in the context of *linear regression* and then define additional properties to remedy the situation.

A failure of convexity

In our counter-example, we'll have $X = Y = R$ (note that this is a convex set). Our hypothesis class H will be that of all (small) *linear predictors*, i.e. just $H = \{f_\alpha : x \mapsto \alpha \cdot x \mid \alpha \in [-1, 1]\}$. The reason that we only allow small predictors is that our final formulation of the learnable class will also demand that H is a bounded set, so this example demonstrates that even boundedness + convexity is still not enough.

We've previously defined *real loss functions* as taking a hypothesis and returning the *real error*, and *empirical loss functions* as taking a hypothesis and a training sequence and returning the *empirical error*. Now we'll look *point-based loss functions* (not

bolded as it's not an official term, but I'll be using it a lot) which measure the error of a hypothesis on a single point only, i.e. they have the form $\ell_{(x,y)} : H \to R$ for some $(x, y) \in X \times Y$. To be more specific, we will turn the *squared loss function* defined in the previous post into a point-based loss function. Thus we will have $\ell^{(2)}_{(x,y)}(h_\alpha) = ||\alpha \cdot x - y||^2 = (\alpha x - y)^2$, where the last equality holds because we're in the one-dimensional case. We will only care about two points (all else will have probability mass 0), namely these two:



That's the point $(1/-1)$ at the left and one all the way to the right at $(1/\mu, 0)$. With μ, think of an extremely small positive number, so that $1/\mu$ is quite large.

If this class were PAC learnable, then there would be a learner A such that, for all $\epsilon, \delta \in (0, 1)$, if the size of the training sequence is at least $m^*(\epsilon, \delta)$, then for all probability distributions over $X \times Y$, with probability at least $1 - \delta$ over the choice of S, the error of A(S) would be at most ϵ larger than that of the best classifier.

So to prove that it is not learnable, we first assume we have some learner A. Then we get to set some ϵ and δ and construct a probability distribution D_A based on A. Finally, we have to prove that A fails on the problem given the choices of ϵ and δ and D_A. That will show that the problem is not PAC learnable.

We consider two possible candidates for D_A. The first is D_L which has all probability mass on the point $(1/-1)$ on the left. The second is $D_?$, which has almost all probability mass on the point $(1/-1)$, but also has μ probability mass on the point $(1/\mu, 0)$. As mentioned, $\mu \in R_+$ will be extremely small; so small that the right point will be unlikely to ever appear in the training sequence.

If the right point doesn't appear in the training sequence, then the sequence consists of only the left point sampled over and over again. In that case, A cannot differentiate

between D_L and $D_?$, so in order to succeed, it would have to output a hypothesis which performs well with both distributions – which as we will show is not possible.

Given our class H, the hypothesis A(S) must be of the form h_α for some $\alpha \in R$. Recall that the classifier is supposed to predict the y-coordinate of the points. Thus, for the first point, $\alpha = -1$ would be the best choice (since $-1 \cdot 1 = -1$) and for the second point, $\alpha = 0$ would be the best choice (since $0 \cdot 1/\mu = 0$).

Now if $\alpha \leq -0,5$, then we declare that $D_A = D_?$. In this case (assuming that the second point doesn't appear in the training sequence), there will be a μ chance of predicting the value $\alpha \cdot 1/\mu = \alpha/\mu \leq -1/2\mu$, which, since we use the squared loss function, leads to an error of at least $\frac{1}{4\mu^2}$, and thus the *expected error* is at least $\mu \frac{1}{4\mu^2} = \frac{1}{4\mu}$, which, because μ is super tiny, is a very large number. Conversely, the best classifier would be at least as good as the classifier with $\alpha = 0$, which would only have error $1 - \mu$ (for the left point), which is about 1 and thus a much smaller number.

Conversely, if $\alpha > -0.5$, we declare that $D_A = D_L$, in which case the error of A(S) is at least $(-0.5 - (-1))^2 = \frac{1}{4}$, whereas the best classifier (with $\alpha = -1$) has zero error.

Thus, we only need to choose some $\epsilon < \frac{1}{4}$ and an arbitrary δ. Then, given the sample size m, we set μ small enough such that the training sequence is less than δ likely to contain the second point. This is clearly possible: we can make μ arbitrarily small; if we wanted, we could make it so small that the probability of sampling the second point is $< \delta^{100}$. That concludes our proof.

Why was this negative result possible? It comes down to the fact that we were able to make the error of the first classifier with $\alpha < -0.5$ large via a super unlikely sample point with super^2 high error – so the problem is the growth rate of the loss function. As long as the loss function grows so quickly that, while both giving a point less probability mass and moving it further to the right, the expected error goes up, well then one can construct examples with arbitrarily high expected error. (As we've seen, the expected error in the case of $\alpha < -0.5$ is at least $\frac{1}{4\mu}$, i.e. a number that grows arbitrarily large as $\mu \to 0$.)

Lipschitzness & Smoothness

There are at least two ways to formalize a requirement such that the loss function is somehow "bounded". They're called *Lipschitzness* and *Smoothness,* and both are very simple.

Lipschitzness says that a function cannot grow too fast, i.e.:

A function $f : R^d \to R$ *is* ρ-*Lipschitz* iff $|f(y) - f(x)| \leq \rho||y - x|| \quad \forall x, y \in R^d$

If f is differentiable, then a way to measure maximum growth is the gradient, because the gradient points into the direction of fastest growth. Thus, one has the equivalent characterization:

A differentiable function $f : R^d \to R$ *is* ρ-*Lipschitz* iff $||\nabla x|| \leq \rho$ for all $x \in R^d$

However, non-differentiable functions can be Lipschitz; for example, the absolute value function |x| on the real numbers is 1-Lipschitz.

Conversely, *smoothness* is about the change of change. Thus, |x| is definitely not smooth since the derivative jumps from -1 to 1 across a single point (smoothness is only defined for differentiable functions). On the other hand, the function x^2 is smooth on all of R. The formal definition simply moves Lipschitzness one level down, i.e.

A differentiable function $f : R^d \to R$ *is* β-*smooth* iff its gradient is β-Lipschitz

Which is to say, iff $||\nabla f(y) - \nabla f(x)|| \leq \beta||y - x||$ for all $x, y \in R^d$. In the one-dimensional case, the gradient equals the derivative, and if the derivative is itself differentiable, then smoothness can be characterized in terms of the second derivative. Thus, a twice-differentiable function $f : R \to R$ is β-smooth iff $|f''(x)| \leq \beta$ for all $x \in R$.

One now defines the class of ***convex Lipschitz bounded*** problems and that of ***convex smooth bounded*** problems. They both require that H has a structure as a familiar set like $B_d(0, M)$, that it is *convex* and *bounded* (so $H = R^d$ would not suffice), and that, for all $(x, y) \in X \times Y$, the point-based loss function $\ell_{(x,y)} : H \to R$ is ρ-Lipschitz (in the former case) or β-smooth *and nonnegative* (in the latter case). If all this is

given, the class is called convex Lipschitz bounded with parameters (M, ρ); or convex smooth bounded with parameters (M, β).

In the previous example, the hypothesis could be represented by the set $[0, 1]$, which is both convex and bounded. (In that example, we thought of it as a set of functions, each of which fully determined by an element in $[0, 1]$; now we think of it as the set $[0, 1]$ itself.) Each point-based loss function $\ell^{(2)}_{(x,y)}$ is convex (and non-negative). However, for any number $x \in R_+$, the loss function $\ell^{(2)}_{(x,y)}$ is defined by the rule $\ell^{(2)}_{(x,y)}(\alpha) = (\alpha \cdot x - y)^2$, and the gradient of this function with respect to α (which equals the derivative since we're in the one-dimensional case) is $2(\alpha \cdot x - y)$. Since this gets large as α gets large, the function is not Lipschitz. Furthermore, the second derivative is $2x$. This means that each particular function induced by the point (x, y) is $2x$-smooth, but there is no parameter β such that all functions are β-smooth.

Surrogate Loss Functions

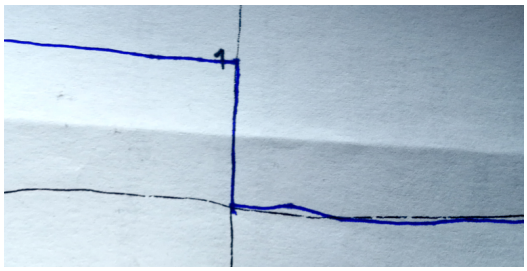
We are now done with task (1), defining the special case. Task (2) would be demonstrating that both convex Lipschitz bounded and convex smooth bounded problems are, in fact, PAC learnable with no further conditions. This is done by defining an algorithm and then proving that that the algorithm works (i.e., learns any instance of the class with the usual PAC learning guarantees). The algorithm we will look at for this purpose (which does learn both classes) is an implementation of *Stochastic Gradient Descent*; however we'll do this in the next post rather than now. For this final chapter, we will instead dive into (3), i.e. find an example of how the ability to learn these two classes is useful even for problems that don't naturally fit into either of them.

Recall the case of *binary linear classification*. We have a set of points in some high-dimensional space $X = R^d$, a training sequence where points are given binary labels (i.e. $Y = \{-1, 1\}$) and we wish to find a *hyperplane* that performs well in the real world. We've already discussed the *Perceptron* algorithm and also reduced the problem to *linear programming*; however, both approaches have assumed that the problem is separable.

We're not going to find a perfect solution for the general case, because one can show that the problem is NP-hard. However, we can find a solution that approximates the

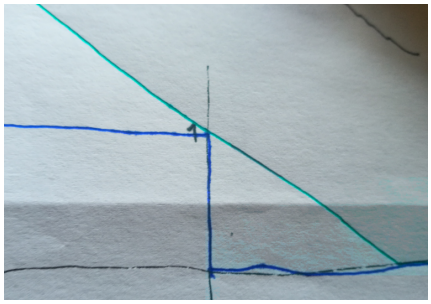
optimal predictor. The approach here is to define a ***surrogate loss function***, which is a loss function ℓ^* that (a) upper-bounds the real loss ℓ^{0-1}, and (b) has nicer properties than the real loss, so that minimizing it is easier. In particular, we would like for it to be a member of one of the two learnable classes we have introduced. Our point-based loss function for ℓ^{0-1} has the form $\ell^{0-1}_{(x,y)}(h_a) := 1_{h_a(x)\neq y}$, where 1_B for a boolean statement B is 1 iff B is true and 0 otherwise.

Recall that each hyperplane is fully determined by one vector in R^d, hence the notation h_a. If we represent H directly as R^d and assume d = 1, then the graph of $\ell^{0-1}_{(x,y)}$ looks like this...



... because in d = 1 and the homogeneous case, the classifier determined by a single number; if this number is positive it will label all positive points with 1; if it's negative, it will label all negative points with 1. If the x-coordinate of the point in question is positive with label 1 or negative with label −1 (i.e. x > 0 and y = 1; or x < 0 and y = −1), then the former case is the correct one and we get this loss function. Otherwise, the loss function would jump from 0 to 1 instead.

Obviously, d = 1 is silly, but it already demonstrates that this loss function is not convex (it makes a turn to the right, and it's easy to find a segment which connects two points of the graph and doesn't lie above the graph). But consider the alternative loss function $\ell^*_{(x,y)}$:



This new loss function can be defined by the rule $\ell^*_{(x,y)}(h_a) := \max(0, 1 - \langle a, x \rangle y)$. In the picture, the horizontal axis corresponds to a and we have $x > 0$ and $y = 1$. This loss function is easily seen to be convex, non-negative, and not at all smooth. It is also $||x||$-Lipschitz. Thus, the problem with $X = R^d$ is not convex Lipschitz bounded, but if we take $X = B_d(0, \rho)$ and also $H = B_d(0, M)$ for some $M, \rho \in R_+$, then it does become a member of the convex-Lipschitz-bounded class with parameters M and ρ, and we can learn via e.g. stochastic gradient descent.

Of course, this won't give us exactly what we want (although penalizing a predictor for being "more wrong" might not be unreasonable), so if we want to bound our loss (empirical or real) with respect to ℓ^{0-1}, we will have to do it via $\ell^{0-1}(h) = \ell^*(h) + [\ell^{0-1}(h) - \ell^*(h)]$, where the second term is the difference between both loss functions. If $\ell^{0-1}(h) - \ell^*(h)$ is small, then this approach will perform well.

UML VI: Stochastic Gradient Descent

(This is part six in a sequence on Machine Learning based on this book. Click here for part 1.)

Stochastic Gradient Descent is the big Machine Learning technique. It performs well in practice, it's simple, and it sounds super cool. But what is it?

Roughly, ***Gradient Descent*** is an approach for function minimization, but it generally isn't possible to apply it to Machine Learning tasks. Stochastic Gradient Descent, which is Gradient Descent plus noise, is a variant which can be used instead.

Let's begin with regular Gradient Descent.

Gradient Descent

Given a function $f : R^d \to R$ and a point $x \in R^d$, the *gradient of* f *at* x is the vector of partial derivatives of f at x, i.e.

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_d}(x)\right)$$

So the *gradient at a point* is an element of R^d. In contrast, the gradient itself (not yet applied to a point) is the function $\nabla f : R^d \to R^d$ defined by the above rule. Going forward, *gradient* always means "gradient at a point".

If d = 1, then the gradient will be $f'(x)$, a number in R. It can look like this:

→

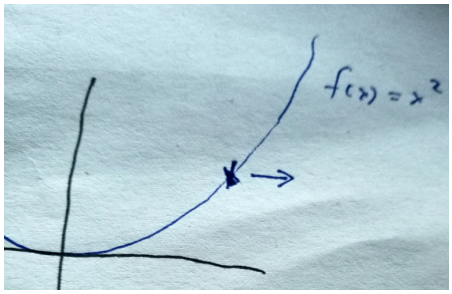
Or like this:

←

Or also like this:

⟶

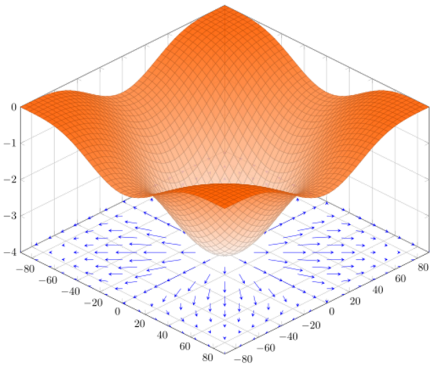
Point being, if $f'(x) > 0$ it'll point rightward and if $f'(x) < 0$ it'll point leftward, also it can have different lengths, but that's it. The idea of gradient descent is that the gradient points into the direction of fastest positive change, thus the opposite of the gradient is the direction of fastest negative change. It follows that, to minimize a function, one can start anywhere, compute the gradient, and then move into the opposite direction.



In the above example, the function goes up, therefore the derivative is positive, therefore the gradient points rightward. Gradient Descent tells us to go into the opposite direction, i.e. leftwards. Indeed, leftward is where the function decreases. Clearly, this is a display of utter brilliance.

Importantly, note that the picture is misleading insofar as it suggests there are more directions than two. But actually, *the gradient lives in the domain space*, in this case, R, *not* in the Cartesian product of domain space and target space, in this case, R^2. Therefore, it cannot point upward or downward.

As silly as the one-dimensional case is, it quickly becomes less trivial as we increase the dimension. Consider this function (picture taken from Wikipedia):



Again, the gradient lives in the domain space, which is R^2, as this image illustrates nicely. Thus it cannot point upward or downward; however, it can point into any direction within the flat plane. If we look at a point on this function, it is not immediately obvious in which direction one should move to obtain the fastest possible decrease of its function value. But if we look at the little arrows (the gradients at different domain points), they tell us the direction of the fastest positive change. If we reverse them, we get the direction of the fastest negative change.

The following is important to keep in mind: in the context of Machine Learning, *the domain space for our loss function is the space of all possibe hypotheses*. So the domain is itself a function space. What we want to do is to minimize a function (a loss function) that takes a function (a predictor, properly parametrized), as an argument, and outputs the performance of that predictor on the real world. For example, if we allow predictors of the form $h(x) = ax^2 + bx + c$ for a regression problem, i.e. $X = Y = R$, our function space could be $H = R^3$, where each $(a, b, c) \in H$ defines one predictor $h_{(a,b,c)}$. Then, there is some well-defined number $\ell((a, b, c))$ that evaluates the performance of $h_{(a,b,c)}$ in the real world. Furthermore, if ℓ is differentiable, there is also some well-defined three-dimensional vector $\nabla\ell((a, b, c))$. The vector, which lives in H, tells us that the real error increases most quickly if we change the predictor in the direction of the gradient – or equivalently, that it decreases most quickly if we change the predictor in the direction opposite to the gradient. This is why we will refer to our elements of H by the letters a, b, c, a, b, c and such; not x or x.

If the loss function were known, one could thus minimize it by starting with an arbitrary predictor parametrized by $a^{(0)}$, computing the gradient $\nabla\ell(a^{(0)})$, and then "moving" into the direction opposite to the gradient, i.e. setting $a^{(1)} := a^{(0)} - \eta \cdot \nabla\ell(a^{(0)})$, where $\eta \in R_+$ is a parameter determining the step size. There is no telling how long moving into the direction opposite to the gradient will decrease the function, and therefore the choice of η is nontrivial. One might also decrease it over time.

But, of course, the loss function isn't known, which makes regular Gradient Descent impossible to apply.

Stochastic Gradient Descent

As always, we use the training sequence $S = ((x_1, y_1), \ldots, (x_m, y_m))$ as a substitute for information on the real error, because that's all we have. Each element (x, y) defines the *point-based loss function* $\ell_{(x,y)} : H \to R$, which we can actually use to compute a gradient. Here, H has to be some familiar set – and to apply the formal results we look at later, it also has to be bounded, so think of $H = B_d(0, M) = \{x \in R^d \mid ||x|| \leq M\}$.

Now we do the same thing as described above, except that we evaluate the performance of our predictor at only a single point at a time – however, we will use a different point for each step. Thus, we begin with some $a^{(0)} \in H$ which defines a predictor $h_{a^{(0)}}$, and we will update it each step so that, after step t, we have the predictor $h_{a^{(t)}}$. To perform step t, we take the loss function $\ell_{(x_t,y_t)} : H \to R$ which maps each $a \in H$ onto the error of the predictor h_a on (x_t, y_t). We compute the gradient of this loss function at our current predictor, i.e. we compute $\nabla\ell_{(x_t,y_t)}(a^{(t-1)})$. Then we update $a^{(t-1)}$

by doing a small step in the direction opposite to this gradient. Our complete update rule can be expressed by the equation

$$a^{(t)} := a^{(t-1)} - \eta \cdot \nabla \ell_{(x_t, y_t)}(a^{(t-1)})$$

The reason why, as the book puts it, Stochastic Gradient Descent can be used to "minimize the loss function directly" is that, given the i.i.d. assumption, the point (x_t, y_t) is an *unbiased* estimate of the real distribution, and therefore, one can prove that the *expected value* – with respect to the point (x_t, y_t) – of the gradient we compute equals the gradient of the real loss function ℓ. Thus, we won't always update into the right direction, but we will update into the right direction + noise. The more steps one does, the lower the expected total noise will be, at least if η is decreased over time. (This is so because, if one does random steps from some origin, then as the number of steps increases, even though the expected absolute distance from the origin will increase, the expected relative distance decreases. The expected relative distance is the term $\frac{\text{expected absolute distance}}{\text{number of steps}}$, which is an analog to $\frac{\text{expected net total noise}}{\text{number of steps}}$ in our case. Then, since we have this consistent movement into the correct direction that grows linearly with "number of steps", the term $\frac{\text{expected net total noise}}{\text{expected net total movement}}$ also decreases, making us likely to converge toward the solution.)

In the context of *supervised learning*, i.e. where training data is available at all, the only things that needs to hold in order for Stochastic Gradient Descent to be applicable is that (a) the hypothesis class can be represented as a familiar set, and (b) we can compute gradients on the point-based loss functions (in practice, they don't even necessarily need to be differentiable at every point). The remaining question is whether or not it will perform well. The most important property here is certainly the convexity of the loss function because that is what guarantees that the predictor will not get stuck in a local minimum, which could otherwise easily happen (the negative gradient will just keep pushing us back toward that local minimum).

In the previous chapter, we introduced the classes of *convex Lipschitz bounded* and *convex smooth bounded* problems. For both of these, one can derive upper-bounds on the expected error of a classifier trained via stochastic gradient descent for a certain number of steps. In the remainder of this chapter, we will do one of those proofs, namely the one for convex smooth bounded problems. This is an arbitrary choice; both proofs are technical and difficult, and unfortunately also quite different, so we're not doing both.

Deriving an upper-bound on the Error

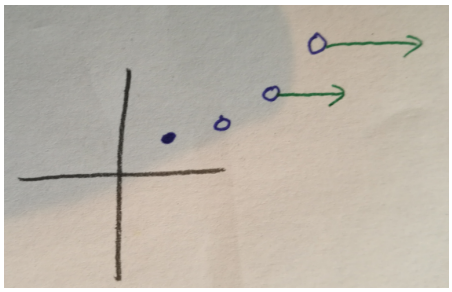
The proof which upper-bounds the expected real error is of the kind that I usually don't work out in detail because it just doesn't seem worth it. On the other hand, I don't want difficult proofs to be excluded in principle. So this will be an experiment of how my style of dissecting each problem until its pieces are sufficiently small works out if applied to a proof that seems to resist intuitive understanding. In any case, working through this proof is not required to understand Stochastic Gradient Descent, so you might end this chapter here.

Now let us begin.

The plan

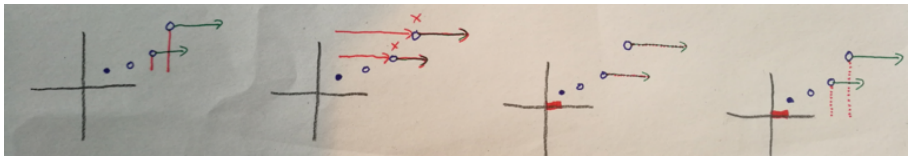
Recall that we wish are given a problem out of the class of convex smooth bounded problems (with parameters β, M), and we run the algorithm A_{SGD} defined earlier in this post on our training sequence, S. We wish to bound the (expected) real error of $A_{SGD}(S)$.

Meet the problem instance:



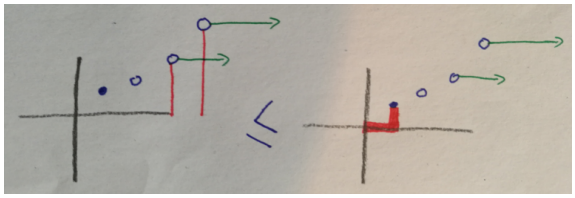
Of course, this is merely the problem instance within the images-world; the symbolic world where the actual proof happens will make no assumptions about the nature of the problem. That said, in the problem instance, we have $d = 1$ and $m = 2$, because that's enough to illustrate the steps. Our training sequence (*not* shown!) consists of two elements, i.e. $S = ((x_0, y_0), (x_1, y_1))$, this time indexed from 0 to avoid having to write a lot of "-1". For our hypothesis class, think of $H = [-10, 10]$, where each $a \in H$ defines a simple linear predictor h_a – although going forward, we will pretend as if a itself is the predictor. What the picture does show is the predictors $a^{(0)} \to a^{(1)} \to a^{(2)}$ through which the algorithm iterates (the highest one is $a^{(0)}$, the second-highest $a^{(1)}$, and the one without a gradient arrow is $a^{(2)}$). They live in 1-dimensional space (on the x-axis). Their real error corresponds to their position on the y-axis. Thus, both the predictors themselves and their errors are monotonically decreasing. The green arrows denote the gradients $\nabla \ell_{(x_0,y_0)}(a^{(0)})$ and $\nabla \ell_{(x_1,y_1)}(a^{(1)})$. The filled point denotes the *optimal* predictor a^*.

For unspecified reasons – probably because it makes analysis easier, although it could also reduce variance in practice – the precise algorithm we analyze puts out the predictor $\frac{1}{m}\sum_{i=0}^{m-1} a^{(i)}$ rather than $a^{(m)}$, even though $a^{(m)}$ naively looks like the better choice. In our case, that means we put out $\frac{1}{2}(a^{(0)} + a^{(1)})$. With that in mind, now meet the proof roadmap:



The proof roadmap illustrates what terms we're concerned with (red), and which upper-bounds we're deriving. Dashed lines mean a small (< 1) factor in front of the term; fat lines mean a large (> 1) factor. In the beginning, we look at the difference between the error of our output predictor $\frac{1}{2}(a^{(0)} + a^{(1)})$ and that of the optimal predictor a^* (leftmost picture). Then we do three steps; in each step, we bound our current term by a different term. First, we bound it in terms of the inner product between the predictors themselves and the gradients. Then, we bound that by δ times the norm of the gradients plus many times the norm of the optimal predictor, where $\delta < 1$. Then we bound that by δ times the total error of our output predictor plus a lot of times the norm of the optimal predictor.

At that point, we've made a circle. Well – not quite, since we started with the difference between the error of the output predictor and that of the optimal predictor (hence why the lines don't go all the way down in picture one) and ended with the total error of the output predictor. But it's good enough. Through algebraic manipulations (adding the error of the optimal predictor on both sides and rescaling), we obtain a bound on the error of our output predictor, in terms of the error of the optimal predictor and the norm of the optimal predictor, i.e.:

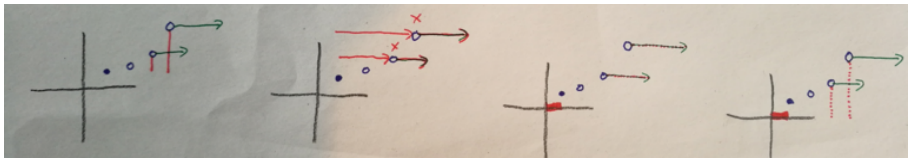


And the equation corresponding to this picture will be the theorem statement.

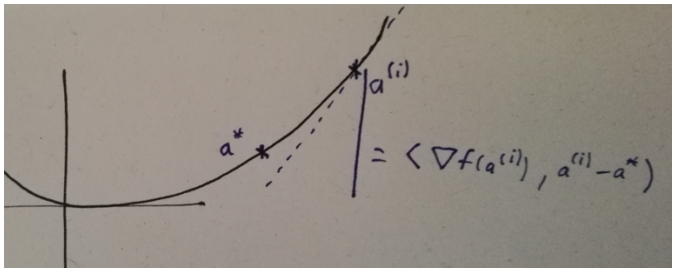
Now we just have to do the three steps, and then the algebraic stuff at the end.

Step 1

We start by bounding the term $\sum_{t=0}^{T-1} [\ell_{(x_t,y_t)}(a^{(t)}) - \ell(a^*)]$, which is closely related to but not identical to the real error of the output predictor (it is defined in terms of the point-based loss functions rather than the real error). T will be the number of steps, so $T = m$. Recall the proof roadmap:



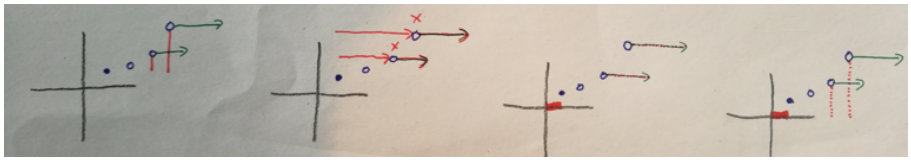
The first stop (second picture) is the term $\sum_{t=0}^{T-1} \langle a^{(t)} - a^*, \nabla_t \rangle$, where we write ∇_t for $\nabla \ell_{(x_t, y_t)}(a^{(t)})$. This bound applies for each summand individually, and it relies on a fundamental property of convex functions, namely that every tangent remains below the function:



This should be easy enough to believe; let's skip the formal proof and continue.

Step 2

Our current term is $\sum_{t=0}^{T-1} \langle a^{(t)} - a^*, \nabla_t \rangle$. Recall the proof roadmap:



The next stop (third picture) is the term $\frac{1}{2\eta}||a^*||^2 + \frac{\eta}{2}\sum_{t=0}^{T-1} ||\nabla_t||^2$. This will be the hardest step – let's begin by reflecting on what it means.

We wish to bound the inner product of predictors and corresponding gradients with the norm of the gradients and the norm of the final predictor. So what we lose is the values of the predictors themselves. However, these are implicit in the gradients, since they encode how we move toward our final predictor (or equivalently, how we move away from the final predictor). Consequently, we will need to use the update rule, $a^{(t+1)} = a^{(t)} - \eta\nabla_t$, to prove this result.

Our approach will be to reduce the inner product $\langle a^{(t)} - a^*, \nabla_t \rangle$ to the difference between $||a^{(t)} - a^*||^2$ and $||a^{(t+1)} - a^*||^2 = ||a^{(t)} - \eta\nabla_t - a^*||^2$. Alas,

$$||a^{(t)} - a^*||^2 - ||a^{(t)} - \eta\nabla_t - a^*||^2 = 2\langle a^{(t)}, \eta\nabla_t \rangle - 2\langle a^*, \eta\nabla_t \rangle - \langle \eta\nabla_t, \eta\nabla_t \rangle$$

Or differently written, $\langle a^{(t)} - a^*, 2\eta\nabla_t \rangle - \eta^2||\nabla_t||^2$. Thus, we have

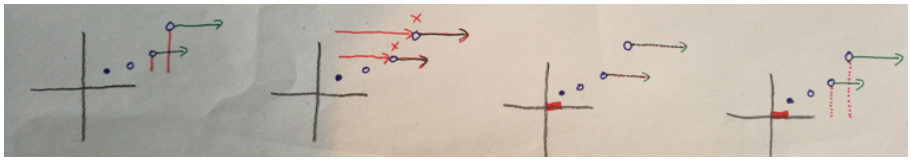
$$\langle a^{(t)} - a^*, \nabla_t \rangle = \frac{1}{2\eta} \langle a^{(t)} - a^*, 2\eta\nabla_t \rangle = \frac{1}{2\eta} \left(||a^{(t)} - a^*||^2 - ||a^{(t+1)} - a^*||^2 + \eta^2||\nabla_t||^2 \right)$$

Then, $\sum_{t=0}^{T-1} \langle a^{(t)} - a^*, \nabla_t \rangle = \frac{1}{2\eta}\sum_{t=1}^{T-1} ||a^{(t)} - a^*||^2 - ||a^{(t+1)} - a^*||^2 + \eta^2||\nabla_t||^2$, and the first two elements of this are a *telescopic sum,* i.e. each segment negates part of the next segment. What remains is $\frac{1}{2\eta}(||a^{(0)} - a^*||^2 - ||a^{(T)} - a^*||^2 + \sum_{t=0}^{T-1} \eta^2||\nabla_t||^2)$, and we can upper bound it as $\frac{1}{2\eta}||a^*||^2 + \sum_{t=0}^{T-1} \frac{\eta}{2}||\nabla_t||^2$, which is the

bound we wanted to have. Here, the $a^{(0)}$ disappeared because the algorithm assumes we start with $a^{(0)} = 0$, which I conveniently ignored in my drawings.

Step 3

Our current term is $\frac{1}{2\eta}||a^*||^2 + \frac{\eta}{2}\sum_{t=0}^{T-1} ||\nabla_t||^2$. Recall the proof roadmap:



The next stop (last picture) is the term $\frac{1}{2\eta}||a^*||^2 + \beta \cdot \eta \sum_{t=0}^{T-1} \ell_{(x_t,y_t)}(a^{(t)})$. This one will also work for each summand separately. We wish to prove that

$$||\nabla_t||^2 = ||\nabla \ell_{(x_t,y_t)}(a^{(t)})||^2 \leq 2\beta \cdot \ell_{(x_t,y_t)}(a^{(t)}),$$

where β is from the smoothness definition of our point-wise loss function. So this bound is all about smooth functions. It even has a name: functions with this property are called *self-bounded*, because one can bound the value of the gradient in terms of the value of the same function.

If one were to prove this formally, one would do so for arbitrary smooth functions, so one would prove that, if $O \subseteq R^d$, where O is convex, and $f : O \to R$ is a β-smooth and nonnegative function, then

$$||\nabla f(x)||^2 \leq 2\beta \cdot f(x) \quad \forall x \in O$$

Recall that f is β-smooth iff $||\nabla f(x) - \nabla f(y)|| \leq \beta||x - y||$ for all x and y. Now, why is this statement true? Well, if we imagine f to model the position of a space ship, then smoothness says that f cannot accelerate too quickly, so in order to reach a certain speed, it will gradually have to accelerate towards that point and thus will have to bridge a certain distance. Let's take the one-dimensional case, and suppose we start from 0 and consistently accelerate as fast as we're allowed to (this should be the optimal way to increase speed while minimizing distance). Then $f(x) = 0$ and $f''(x) = \beta$, which means that $f'(x) = \beta x$ and $f(x) = \frac{1}{2}\beta x^2$; then $f'(x)^2 = (\beta x)^2 = 2\beta \cdot \frac{1}{2}\beta x^2 = 2\beta f(x)$. This is reassuring, because we've tried to construct the most difficult case possible, and the statement held with equality.

The formal proof, at least the one I came up with, is quite long and relies on line integrals, so we will also skip it.

The algebraic stuff

We have established that

$$\sum_{t=0}^{T-1} [\ell_{(x_t,y_t)}(a^{(t)}) - \ell(a^*)] \leq \frac{1}{2\eta}||a^*||^2 + \beta \cdot \eta \sum_{t=0}^{T-1} \ell_{(x_t,y_t)}(a^{(t)})$$

which can be reordered as

$$\sum_{t=0}^{T-1} \ell_{(x_t,y_t)}(a^{(t)}) \leq \frac{1}{2\eta}||a^*||^2 + \beta \cdot \eta \sum_{t=0}^{T-1} \ell_{(x_t,y_t)}(a^{(t)}) + \sum_{t=0}^{T-1} \ell(a^*).$$

At this point, we better have that $\beta\eta < 1$, otherwise, this bound is utterly useless. Assuming $\eta\beta < 1$ does indeed hold, we can further reformulate this equation as

$$(1-\beta\eta)\sum_{t=0}^{T-1} \ell_{(x_t,y_t)}(a^{(t)}) \leq \frac{1}{2\eta}||a^*||^2 + T\cdot\ell(a^*)$$

Rescaling this, we obtain

$$\sum_{t=0}^{T-1} \ell_{(x_t,y_t)}(a^{(t)}) \leq \frac{1}{(1-\beta\eta)}(T\cdot\ell(a^*) + \frac{1}{2\eta}||a^*||^2)$$

Dividing this by T to have the left term closer to our actual error, we get

$$\frac{1}{T}\sum_{t=0}^{T-1} \ell_{(x_t,y_t)}(a^{(t)}) \leq \frac{1}{(1-\beta\eta)}(\ell(a^*) + \frac{1}{2\eta T}||a^*||^2).$$

Now we take expectations across the randomization of the training sequence S on both sides. Then the left side becomes the expected real error of our output predictor $A_{SGD}(S) = \frac{1}{m}\sum_{i=0}^{m-1} a^{(i)}$. The right term doesn't change, because it doesn't depend on S. Thus, we have derived the bound

$$E(A_{SGD}(S)) \leq \frac{1}{(1-\beta\eta)}(\ell(a^*) + \frac{1}{2\eta T}||a^*||^2)$$

and this will be our final result. By choosing η correctly and rearranging, it is possible to define a function s^* such that, for any β and M upper-bounding the hypothesis class ($||a^*||$ cannot be an input since we don't know the optimal predictor a^*) and any arbitrarily small $\epsilon \in R_+$, if $T \geq s^*(\beta, M, \epsilon)$, then the above is upper-bounded by $\ell(a^*) + \epsilon$. To be specific, s^* will be given by $s^*(\beta, M, \epsilon) = 12\cdot M^2\cdot\beta\cdot\frac{1}{\epsilon^2}$.

Reflections

Recall that β is the parameter from the smoothness of our loss function, T is the number of training steps we make (which equals the number of elements in our training sequence), and η is the step size, which we can choose freely. In the result, we see that a smaller β is strictly better, a larger T is strictly better, and η will have its optimum at some unknown point. This all makes perfect sense – if things had come out any other way, that would be highly surprising. Probably the most non-obvious qualitative property of this result is that the norm of the optimal predictor plays the role that it does.

The most interesting aspect of the proof is probably the "going around in a circle" part. However, even after working this out in detail, I still don't have a good sense of why we chose these particular steps. If anyone has some insight here, let me know.

UML VII: Meta-Learning

(This is the seventh post in a sequence on Machine Learning based on this book. Click here for part I.)

Meta

A more accurate title for this post would be "*Meta, Error bounds, Test Data, Meta-Learning, Error decomposition, and an optional introduction to general probability spaces.*" The first item is me trying to categorize the stuff we've been doing in this sequence so far, which is meta-learning but nonetheless different from the Meta-Learning item, which is about a Machine Learning technique.

Categorizing Machine Learning Insights

Although the book doesn't do this, I think taking some time to reflect on how exactly we've been making progress is very worthwhile. I would categorize the material we've covered so far as follows:

- (1) Carving out relevant and learnable classes out of ML problem space
  - (1.1) Defining such classes and establishing results about their limits
    - **Examples:** Linear Predictors (post 4), Convex (bounded, Lipschitz / smooth) problems (post 5), the negative result on convex problems (post 5)
  - (1.2) Finding algorithms that learn these classes and prove that they work.
    - **Examples:** Perceptron (post 4), Linear Programming (post 4), applied linear algebra (post 4), Stochastic Gradient Descent (post 6)
- (2) Extending the usability of such classes
    - **Examples:** Primarily surrogate loss functions (post 5); also the mapping of inhomogeneous linear problems to linear problems (post 4), even if it's just a detail. There will be more stuff here in future posts.
- (3) General (as in, widely applicable) theoretical work
    - **Examples:** The general learning model, work on overfitting, the PAC learning framework, the basic error decomposition, the ERM approach (all post 1), the uniform convergence property, the No-Free-Lunch Theorem, the VC dimension, the fundamental theorem of statistical learning (all post 2), the nonuniform learning framework, the SRM approach, the minimal description length, the basics of computational complexity (all post 3), meta-learning, error bounds via test data, and advanced error decomposition (later in this post)

Not quite covered: coming up with learning algorithms. So far, that's just Stochastic Gradient Descent, which I have listed as a solution to solve convex Lipschitz/smooth bounded problems, but it's not restricted to that. Also, neural networks supposedly have an impressive track record of fitting all sorts of functions, and I don't expect there to be analogous theoretical results.

Based on this categorization (which is not perfect – much of (3) is only about binary classification), the entire first part of the book which was covered in posts I-III is only

about broad theoretical work, hence "foundations". The remaining book jumps around between all three. Note that my adaption isn't always linear.

Error Bounds

One way to motivate this chapter is that we *wish to find better guarantees for the quality of predictors.* We usually either care about the term $\ell(A_{ERM}(S)) - \ell(h^*)$ (where h^* is the predictor with minimal real error in H) or about $\max_{h \in H}[\ell(h) - \ell_S(h)]$. It turns out that both are closely related. We'll call either of these the "gap". Now one can alternatively ask

- "given that I have m data points, what is the smallest upper-bound on the gap that I can prove?"; or
- "given that I want to establish an upper bound of ϵ on the gap, how many data points do I need?"

where, in both cases, the guarantee will hold with some probability $1 - \delta$, rather than with certainty. The first would be an error bound, the second a sample complexity bound, but they're equivalent: given an error bound, one can take the corresponding equation, which will have the form $\ell(A_{ERM}(S)) - \ell(h^*) \leq \langle$some term which depends on m$\rangle$, set the left part to ϵ and solve it for m, which will yield a sample complexity bound. Going into the opposite direction is no harder.

Musings on past approaches

The punchline of this section is that all bounds we have looked at so far are based on the performance of the classifier *on the data it's been trained on*. Given that the output predictor may be highly dependent on patterns that only exist in the training data but not in the real world, this is difficult – and, as we know from the No-Free-Lunch Theorem, even impossible without making further assumptions.

This implies that we must have made such assumptions, and indeed we have. In addition to the empirical error, our current bounds have been based on

- (1) the hypothesis class
- (2) the properties of the classifier that the learner guarantees.

By (2), I mean that the classifier is in the set $\text{argmin}_{h \in H}[\ell_S(h)]$ in the case of A_{ERM}, in the class $\text{argmin}_{h \in H}[\ell_S(h) + \epsilon_n(\delta, h)]$ in the case of A_{SRM}; and in the case of A_{SGD} we've used the update rule in the proof that derived the error bound.

Deriving error bounds with test data

By testing the classifier on data that the learner didn't have access to, we can toss out (1) and (2) in favor of a purely statistical argument, which results in a bound that is more general, much easier to prove, and stronger. The downside is that it requires additional data, whereas the previous bounds were "free" in that regard.

To be more precise, the idea now is that we split our existing data into two sequences S and T; we train a learner on S and we test the output predictor $A(S)$ on T to obtain an estimate of the real error. In symbols, we use $\ell_T(A(S))$ as our estimate for $\ell(A(S))$, where ℓ_T is defined just like ℓ_S only, of course, using T instead of S as the set (this definition is implicit if one simply regards the definition of ℓ_S as applying to ℓ with any set S as a subscript). So $\ell_T(h) = \frac{1}{|T|}|\{(x, y) \in T \mid h(x) \neq y\}|$. We will call the output of ℓ_T the *test error*.

In general, the real error does not equal the test error (in symbols, $\ell(A(S)) \neq \ell_T(A(S))$ because the test error is based on the test sequence T which may be unrepresentative. However, the real error equals the *expected test error* (we'll prove this in the optional chapter at the end of the post). Therefore, we only need an upper-bound on the difference between the expected value and the actual value of random variables (this is the aforementioned statistical argument), and a theorem to that end has already been established by relevant literature: *Hoeffding's Inequality*.

> Let $\theta_1, \ldots, \theta_m$ be i.i.d. RVs that range over the interval $[0, 1]$. Then, for all $\epsilon \in \mathbb{R}_+$, it holds that $\Pr(|\bar{\theta} - E[\bar{\theta}]| > \epsilon) \leq 2e^{-2\epsilon^2 m}$, where $\bar{\theta} := \frac{1}{m}\sum_{i=1}^{m} \theta_i$.

This uses the same notational shortcut that is used all the time for Random Variables (RVs), namely pretending as if RVs are *numbers* when in reality they are *functions*. Importantly, note that $\bar{\theta}$ depends on the input randomness, whereas $E[\bar{\theta}]$ is a constant.

In our case, the random variables we are concerned with are the *point-based loss functions applied to our predictor* for our instances in T, i.e., if $T = ((x_1, y_1), \ldots, (x_t, y_t))$, then our random variables are $\theta_1 = \ell_{(x_1,y_1)}(h), \ldots, \theta_t = \ell_{(x_t,y_t)}(h)$. They are random variables if we regard the points they're based on as unknown (otherwise they'd just be numbers). Their mean, $\bar{\theta} = \frac{1}{|T|}\sum_{i=1}^{|T|} \ell_{(x_i,y_i)}(h)$ equals the test error, $\ell_T(h)$, whose randomness ranges over the choice of all of T. So $\bar{\theta} = \ell_T(h)$ and $E(\bar{\theta}) = \ell(h)$. Given this,

Hoeffding's inequality tells us that $\Pr(|\ell(h) - \ell_T(h)| > \epsilon) \leq 2e^{-2\epsilon^2 m}$, provided that all loss functions range over $[0, 1]$ (if not, then as long as they are still bounded, a generalized version of this theorem exists). Furthermore, if one drops the $|\ |$, the failure probability halves, i.e. $\Pr(\ell(h) - \ell_T(h) > \epsilon) \leq e^{-2\epsilon^2 m}$. Setting $\delta := e^{-2\epsilon^2 m}$ and solving for ϵ, we obtain $\epsilon = \sqrt{\frac{\ln(1/\delta)}{2m}}$. Thus, we have established the following:

> If T is a sequence of i.i.d examples sampled by the same distribution that generated S, but is disjoint from S, then the inequality $\ell_T(A(S)) - \ell(A(S)) \leq \sqrt{\frac{\ln(1/\delta)}{2|T|}}$
>
> holds with probability at least $1 - \delta$ over the choice of T.

For a brief comparison: a bound provided by the (quantitative version of) the fundamental theorem of statistical learning states that $|\ell_S(h) - \ell(h)| \leq \sqrt{C\frac{d + \ln(1/\delta)}{m}}$, provided that H has VC-dimension $d \in \mathbb{N}$. So this bound relies on a property of the hypothesis class (and if that property doesn't hold, it doesn't tell us anything), whereas our new bound always applies. And as a bonus, the new bound doesn't have the constant C but instead has a nice 2 in the denominator.

Meta-Learning

The book now goes on to talk about how this idea can be used for ***model selection.*** Let's begin by summarizing the argument.

Model Selection ...

Suppose we have several possible models for our learning task, or alternatively one model with different parameters (maximal polynomial degree, step size of stochastic gradient descent, etc). Then we can proceed as follows: we split our training data into three sequences, the training sequence S, the *validation sequence* V, and the test sequence T. For each model we look at, we train the corresponding learner A on S, then we test all output hypotheses $A_1(S), \ldots, A_k(S)$ on V, i.e. we compute $\ell_V(A_1(S)), \ldots, \ell_V(A_k(S))$, and pick one with minimal validation error, i.e. we pick $A_j(S)$

such that $j \in \operatorname{argmin}_{i \in \{1,\dots,k\}} \ell_V(A_i(S))$. Finally, we compute $\ell_T(A_j(S))$ and use that value to argue that the predictor performs well (but not to change it in any way).

Since the learners $A_1(S), \dots, A_k(S)$ haven't been given access to V, the validation error will be unbiased feedback as to which learner performs best, and since even *that* learner has been computed without using T, the test error is yet again unbiased feedback on the performance of our eventual pick $A_j(S)$. That is, provided that there are no spurious correlations across our data sets, i.e. provided that the i.i.d. assumption holds.

One can be even more clever, and trade some additional runtime for more training data. The test sequence T remains as-is, but rather than partitioning the remaining data in S and V we partition it into a bunch of blocks of equal size, say k blocks, and then, for each such block i, we

- train our learner A on all blocks $S_{\neq i}$
- compute the error on block i, i.e. $\ell_{S_i}(A(S_{\neq i}))$

This way, we obtain k different validation errors (one for each classifier which was trained on nine blocks and tested on the tenth); now we take the mean of all of them as our ultimate validation error. We do this entire thing for each learner A, take the one that performed best, and then we can even retrain that learner on all k blocks to make sure we fully utilize all data we have.

Finally, just as before, we test that final predictor using the test data and output the result. This entire process is called k***-fold cross validation.***

... is just meta-learning

While this is all well and good, I think it's important to understand that the specifics are non-fundamental. What is novel here is the concept of obtaining unbiased feedback by computing the empirical error of a predictor based on a sequence that has not been used to learn that predictor. Beyond that, the observation is that it might be useful to do meta-learning, i.e. have several levels of learning, i.e. partition all possible hypotheses in a bunch of different classes, take the best from each class, and then take the best across all classes.

But the notion of exactly two levels is arbitrary. Suppose we have a bunch of models, and each model has some parameters. Then we can partition our training data in *four* sequences S, V_1, V_2, T. For each model, we choose a bunch of plausible parameter

values, so say we have n models and each one has m parameter settings. For each such setting, we train the learner on S and test it on V_1. We pick the optimal one among those m; this will be the predictor with the optimal parameters for our model. We do this for all n models, leading to n predictors, all of whom use the optimal parameters for their model. We test these n predictors using V_2 and output the optimum yet again. Finally, we test that predictor on T.

Similarly, one could imagine four levels, or however many. The crucial thing is just that choosing the optimal predictor of the next lower level has to be done with data that hasn't been used to learn that predictor. So in the above example, for each [model and particular parameter setting], the respective output hypothesis just depends on S. If we then compute the optimal predictor of that model by looking at which of the m predictors (corresponding to the m parameter settings) performed best on V_1, this output predictor also depends on V_1. In fact, the procedure I just described is just one kind of learning algorithm that uses both S and V_1 as training data. Similarly, if we take the predictor with the smallest error on V_2 across all n models to obtain our final predictor, then this predictor depends on S and V_1 and V_2 – in fact, the procedure I just described is just one kind of learning algorithm that uses S and V_1 and V_2.

Differently put, to derive a predictor for a particular problem, one might choose to apply meta-learning, i.e. choose k different models, train a predictor in each model, and then select the best of those k predictors, based on their performance on training data which hasn't been used to train them – and each of those k smaller problems is itself just another problem, so one might again choose to apply meta-learning, and so on.

The trick from the last section for saving data can also be used with more than 2 levels, although it quickly increases runtime.

The idea of using a test sequence is more fundamental (i.e. not just another level), since it's only used to obtain a certificate of the quality of the final predictor, but *not* for learning (in any way). As soon as one violates this rule (for example by choosing another predictor after the first one showed poor performance on the test data), the bound from Hoeffding's inequality is no longer applicable, since the test error is no longer an unbiased estimate of the real error.

Error Decomposition

So far, we've decomposed the error of a predictor like so:

$$\ell(h) = (\ell(h) - \ell(h^*)) + \ell(h^*)$$

where $h^* \in \operatorname{argmin}_{h \in H}[\ell(h)]$. The term $\ell(h^*)$ is the *approximation error*, which is the lowest error achievable by any predictor in our hypothesis class ("how well does this class approximate the problem?") and the term $(\ell(h) - \ell(h^*))$ is the *estimation error* ("how well did we estimate the best predictor?").

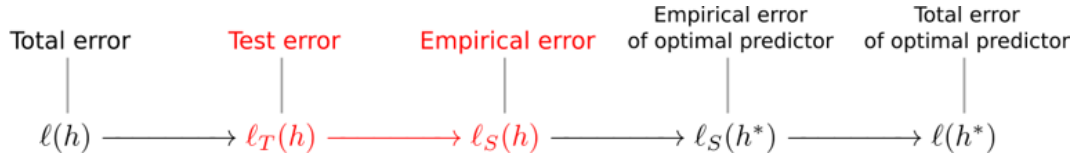
The usefulness of this approach is limited. One doesn't actually know the approximation error – if learning fails, it's not clear which of the two errors is at fault. Furthermore, both overfitting and underfitting is part of the estimation error, which suggests that it should be divided up further (whereas the approximation error can stay as-is). This is what we do now. Our new decomposition will rely heavily on having the kind of unbiased feedback from some independent data which we've been talking about in the previous chapter. Note, however, that while I'll only talk about the test sequence T going forward, everything applies equally if one uses a validation sequence instead (i.e. training data which has not been given to the learner). In fact, using such data is the way to go if one intends to change the learner in any way upon seeing the various parts of the decomposition.

I find the notation with differences not very illustrative, which is why I'm making up a different one. The above decomposition can be alternatively written as

$$\ell(h) \longrightarrow \ell(h^*)$$

where we decompose the left term into the value of the arrow plus that of the right term. (Each error corresponds to adding "+[right term − left term]". If there are more than two terms, we decompose the leftmost term into the sum of all errors plus the rightmost term (actually, any term in the chain equals the sum of all errors to its right plus the rightmost term).

That said, here is a more sophisticated error decomposition:



The red terms are the ones we have access to. Note that it is not the case that each step is necessarily negative, so the values are not monotonically decreasing. Now let's look at this more closely.

"$\ell(h) \longrightarrow \ell_T(h)$" can be bounded using Hoeffding's inequality.

Let's pause already and reflect on that result. Our only terminal value is to maximize the leftmost term; the first arrow can be bounded tightly; and the second term is

known. It follows that, if the test error is small we can stop right there (because the real error is probably also small). In fact, that was the point of the previous chapter. The remaining decomposition is only necessary if the test error is larger than what is acceptable.

"$\ell_T(h) \longrightarrow \ell_S(h)$" is the difference between the unbiased and the biased error estimate – the error on the data the learner had no access to minus the error on the data it did have access to. It measures how much we overfit – which is quite useful given that this is another term we have access to. Thus, if the test error is large but the empirical error small, we might want to change our learner such that fits the training data less closely. However, we don't have any guarantee in this case that overfitting is the only problem. It's possible that we overfit *and* the approximation error is large.

If the test error and the empirical error are both large, that's when we're interested in the remaining part of this decomposition.

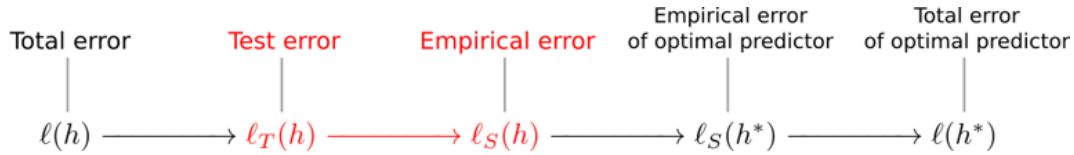
"$\ell_S(h^*) \longrightarrow \ell(h^*)$" is usually negative, at least if we're using A_{ERM} or something close to it.

"$\ell_S(h^*) \longrightarrow \ell(h^*)$" can also be bounded by Hoeffding's inequality, because h* does not depend on S (unless we've messed with the hypothesis class based on the training data).

Finally, $\ell(h^*)$ is the approximation error from our previous decomposition. It doesn't make sense to decompose this further – if the approximation error is large, our hypothesis class doesn't contain any good predictors and learning it is hopeless.

Thus, based on the previous 3 paragraphs, the empirical error is unlikely to be significantly larger than the total error of the output predictor, i.e. the approximation error. (It might be significantly smaller, though.) It follows that if the empirical error is high, then so is the approximation error (but not vice-versa), *with the important caveat that one needs to be confident in the learner's ability to minimize empirical risk.* If the problem is learning starcraft, then it's quite plausible that minimizing the empirical risk is so difficult that the empirical risk could be large even if the approximation error is very small.

To summarize...



... one can reason like this:

- Test error...
  - is small → real error is probably small too, so everything's good
  - is large → empirical error...

- is small → the learner probably suffers from overfitting[1]
- is large → the empirical error seems...
  - easy to minimize → approximation error is probably large
  - hard to minimize → ??? (the problem is just hard)

If the approximation error is large, then the problem is unrelated to our learner. In that case, the hypothesis class H may be "too small"; perhaps it contains only linear predictors and the real function just isn't linear. Alternatively, it could also be the case that the real function is approximately linear, but we just didn't choose features that represent meaningful properties of the data points. For example, if the problem is learning a function that classifies emails as spam / not spam, one could imagine that someone was just really wrong about which features are indicative of spam emails. This is the earliest point of failure; in this case, learning might be impossible (or at least extremely difficult) even with a good hypothesis class and a good learner.

[1] While this is technically true, one might actually be fine with some amount of overfitting. If so, then this case isn't quite satisfactory, either; in particular, one doesn't know whether the approximation error is large or not. Another way to approach the question is to train the learner on parts of the training sequence first, say $S_1 \subsetneq \cdots \subsetneq S_k = S$, and examine the test error for each learner $A(S_j)$. If it goes down as we use more data, that's some evidence that the approximation error is low.

In general, this chapter is far from a full treatment of error decomposition.

Proof that E(test error) = real error

As mentioned, this chapter is completely optional (also note that the post up to this point is roughly as long as previous posts in this sequence). It's long bothered me that I didn't understand how a general probability space was defined, so if you share that frustration, this chapter will provide an answer. However, the connection to Machine Learning is very loose; one needs general probability spaces to prove the result, but the actual proof is quite easy, so this will be a lot of theory with a fairly brief application.

Preamble: General Probability Spaces

Before the general case is introduced, one is generally being taught two different important special cases of probability spaces. The first is the *discrete* one. Here, a probability space is given by a pair (Ω, p), where Ω is a countable sample space and $p : \Omega \to [0, 1]$ a function that assigns to each *elementary event* $\omega \in \Omega$ a probability. For any subset $A \subseteq \Omega$, the probability $P(A)$ is then simply defined as $\sum_{\omega \in A} p(\omega)$. And for a random variable $X : \Omega \to R$, we have the definition

$$(1): E(X) := \int_{-\infty}^{\infty} xf(x)\,dx.$$

This is super intuitive, but it can't model how an event happens at a random point throughout a day. For that, we need the *absolutely continuous* case. This is where no one point has nonzero probability, but intervals do. In other words, one is in the absolutely continuous case whenever the probability distribution admits of a *probability density function* such that, for each interval, the probability of that interval is equal to the integral of that function over that interval. If $\Omega = \mathbb{R}$ (covering only that case here), we can write the probability space as (Ω, f) where for any interval $[a, c] \subseteq \Omega$, we have $P([a, c]) = \int_a^c f(x)\,dx$. This is also rather intuitive – one can just draw an analogy between probability mass and area. A single point has an "infinitely thin" area below it, so it has zero probability mass (and in fact the same is true for any countable collection of points because even the tiniest interval is uncountable), but an interval does have real area below it. The height of the area is then determined by f.

Given a random variable $X : \Omega \to \mathbb{R}$, we have

$$(2): E(X) := \int_{-\infty}^{\infty} xf(x)\,dx.$$

But neither of these is the general case, because one assumes that every point has nonzero probability and the other assumes that no point does. In the general case, a probability space looks like this:

$$(\Omega\ ,\ \Sigma\ ,\ P\ )$$

The first element Ω is the same as before – the sample space. But the Σ is something new. It is called a σ-*algebra,* which is a *set of subsets* of Ω, so $\Sigma \subseteq P(\Omega)$, and it *consists of all those subsets which have a probability*. It needs to fulfill a bunch of properties like being closed under complement and like $\Omega \in \Sigma$, but we don't need to concern ourselves with those.

The probability distribution is a function $P : \Sigma \to [0, 1]$. So it's a function that assigns subsets of Ω a probability, but only those subsets that we declared to have a probability, i.e. only those subsets that are elements of our σ-algebra Σ. It must have a bunch of reasonable properties such as $P(\Omega) = 1$ and $P(A) + P(B) = P(A \sqcup B)$, where the symbol $\sqcup$ is meant to indicate that A and B are disjoint.

(In the discrete case, we don't need to specify Σ because we'd simply have $\Sigma = P(\Omega)$, i.e. every subset has a probability. I'm not sure how it would look in the continuous case, but maybe something like the set all of all subsets that can be written as a countable union of intervals plus a countable union of single points.)

The question relevant for us is now this: given a general probability space (Ω, Σ, P) and a random variable $X : \Omega \to P$, how do we compute the expected value of X? And the answer is this:

$$(3): E(X) := \int_\Omega X(\omega)\ dP(\omega)$$

where $\int$ is the *Lebesgue integral* rather than the Riemann integral. But how does one compute a Lebesgue integral? Fortunately, we don't require the full answer to this question, because some of the easy special cases will suffice for our purposes.

Computing Lebesgue Integrals of Simple Functions

The setting to compute a Lebesgue integral is a bit more general than that of a general probability space, but since we're only trying to do the easy cases anyway, we'll assume our space has the same structure as before. However, we will rename our probability distribution P into μ and call it a *measure,* so that our space now looks like this:

$$(\Omega ,\ \Sigma ,\ \mu)$$

And we'll also rename our random variable into f to make it look more like a general function. So $f : \Omega \to R$ is the function whose Lebesgue integral $\int_X f\ d\mu$ we wish to compute.

Recall that P assigns a probability to the subsets of Ω that appear in Σ, and that the entire space has probability 1. Analogously, μ assigns a *measure* to subsets of Ω, and the entire space has measure 1. So let Y be a subset that is "half" of Ω, then $\mu(Y) = \frac{1}{2}$.

One now proceeds differently than with Riemann integrals. Rather than presenting the general case right away, we begin with the simplest kind of function, namely *indicator functions*. That is a function of the form 1_S for some $S \in \Sigma$, and it is simply 1 on S and 0 everywhere else. And now, even though we don't understand how Lebesgue integrals work in general, in this particular case the answer is obvious:

$$\int_\Omega 1_S \, d\mu := \mu(S).$$

The function 1_S just says "I count S once and I count nothing else", so the integral has to just be one times the size, i.e. the *measure,* of S. Which is μ(S).

So for example, $\int_\Omega 1_Y \, d\mu = \mu(Y) = \frac{1}{2}$ and $\int_\Omega 1_\Omega \, d\mu = \mu(\Omega) = 1$.

Next, if f is not itself an indicator function, but it can be written as a finite weighted sum of indicator functions, i.e. $f = \sum_{i=1}^{m} a_i 1_{S_i}$, then f is called a *simple function* and we define

$$\int_X f \, d\mu := \sum_{i=1}^{m} a_i \mu(S_i)$$

So for example, if $f = 2 \cdot 1_Y$, i.e. the function that is 2 on one half of the space and 0 on the other, then $\int_X f \, d\mu = 2\mu(Y) = 2 \cdot \frac{1}{2} = 1$.

And this where we stop because integrating simple functions is all we need.

An example

Let's return to our setting of general probability spaces. Here, the probability distribution becomes the measure. So let's say our space is $\Omega = [0, \frac{1}{2}] \cup \{10\}$. Our σ-algebra Σ includes all sub-intervals of $[0, \frac{1}{2}]$ (closed or open or half-open), and all such intervals plus the point 10. Our probability distribution P says that $P(\{10\}) = \frac{1}{2}$ and the probability mass of an interval in $[0, \frac{1}{2}]$ is just the size of the interval, so if $[a, c] \subseteq [0, \frac{1}{2}]$, then $P([a, c]) = c - a$. Then, $P(\Omega) = 1$. Recall that P has to assign each set $M \in \Sigma$ a probability, which it now does.

Let's say $X : \Sigma \to R$ is a random variable given by $X = 5 \cdot 1_{[0, \frac{1}{3}]} + 7 \cdot 1_{\{10\}}$. Then, X is a simple function – it is the weighted sum of two indicator functions – and we can compute the expected value of X as

$E(X) = \int_\Omega X \, dP = 5 \cdot P([0, \frac{1}{3}]) + 7 \cdot P(\{10\}) = 5 \cdot \frac{1}{3} + 7 \cdot \frac{1}{2} = \frac{5}{3} + \frac{7}{2} = \frac{31}{6}$.

The actual proof

The real error of h := A(S) is

$$\ell\,(\,h\,)\;=\;D\,(\,\{\,(\,x\,,\;y\,)\;\in\;X\;\times\;Y\;\;\mid\;\;h\,(\,x\,)\;\neq\;y\,\}\,)$$

The test error $\ell_T(h)$ depends on the test sequence. Let m be the length it the sequence, so that $T = ((x_1, y_1), \ldots, (x_m, y_m))$; then to compute the *expected* test error, we have to compute the Lebesgue integral

$$\int_\Omega \ell_T(h)\, dD^m \text{ where } \Omega = (X \times Y)^m$$

So our measure for the space Ω is now D^m. We wish to write $\ell_T(h)$ as a sum of indicator functions. Recall that $\ell_T(h) = \frac{1}{m}|\{(x, y) \in T \mid h(x) \neq y\}|$, i.e. $\ell_T(h)$ counts the number examples which h got wrong. Thus, by defining the sets $W_k := \{((x_1, y_1), \ldots, (x_m, y_m)) \in \Omega \mid h(x_k) \neq y_k\}$ for all $k \in \{1, \ldots, m\}$, we get the equality $\ell_T(h) = \sum_{k=1}^{m} \frac{1}{m} \cdot 1_{W_k}$.

Thus, the integral simply equals the probability mass of the respective sets (multiplied by their respective weights). In symbols,

$$\int_\Omega \ell_T(h)\, dD^m = \sum_{k=1}^{m} \frac{1}{m} D^m(W_k)$$

The i.i.d. assumption tells us that $D^m(W_k) = D(\{(x, y) \in X \mid h(x) \neq y\}) = \ell(h)$ (in fact, this or something similar is probably how it should be formally defined), and therefore the above sum equals $\sum_{k=1}^{m} \frac{1}{m} \ell(h) = \ell(h)$.

UML VIII: Linear Predictors (2)

(This is the eight post in a sequence on Machine Learning based on this book. Click here for part I. Alternatively, click here for part IV, which covers the basics of linear predictors.)

The mission statement for this post is to

- continue the study of linear predictors
- widen the applicability of this class

The latter will be quite successful, which is why more tools to learn linear predictors are desirable. In particular, most ways to reduce non-linear problems to linear ones will cause the dimension to blow up, which is why we want techniques that can handle high-dimensional instances (but we won't get to those in this post).

Before diving into the Machine Learning material, we need a tool from Linear Algebra.

Orthogonal Decomposition

The orthogonal decomposition is something my favorite textbook taught me, and we'll need it both for this post for the next (?) one.

Let V be a vector space and U be a subspace. Let $x \in V$ and $u \in U$ be two nonzero vectors. We wish to write x as a multiple of u plus a vector orthogonal to u. That's the orthogonal decomposition. (Is this always possible? Think about it for R^2 and R^3.) We can write

$$x = \alpha u + (x - \alpha u)$$

which is clearly correct, and if we can choose α such that u and $(x - \alpha u)$ are orthogonal; this is the desired orthogonal decomposition. As one does in math, we now write down what we want to have as an equation and solve it for the desired variable, i.e.

$$\langle u, x - \alpha u \rangle \overset{!}{=} 0 \iff \langle u, x \rangle - \alpha ||u||^2 = 0 \iff \alpha = \frac{\langle u, x \rangle}{||u||^2}$$

So the answer is affirmative – it is always possible. The thing to remember is that, if x is our given vector and u the first vector of the decomposition, we have to put the inner product between both in the nominator and the squared norm of u in the denominator; that will be the α from the decomposition equation.

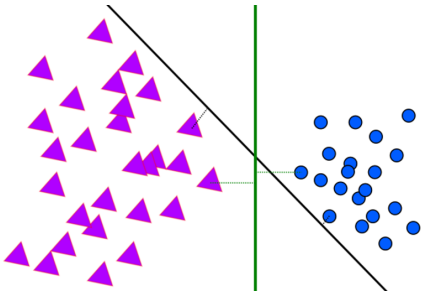
Orthogonal decomposition can also be used for the simplest and most intuitive proof of the Cauchy-Schwartz inequality that I've seen (simplest even if one has to derive the orthogonal decomposition as part of the proof), but that's not important for this post. Let's do Machine Learning.

Support Vector Machines

Support Vector Machines is an odd name for hyperplanes that are selected in a way that attempts to maximize the distance between the hyperplane and the nearest point(s). We differentiate between ***Hard Support Vector Machines*** which are hyperplanes that separate the instance space perfectly and where (among all hyperplanes which do that) the distance to the nearest point is maximal, and ***Soft Support Vector Machines*** which try to simultaneously maximize the distance to the nearest point(s) and the number of correctly classified points; without the assumption that they'll get the second one perfectly right. Thus, Hard Support Vector Machines only exist if the instance space is linearly separable, but Soft Support Vector Machines always exist.

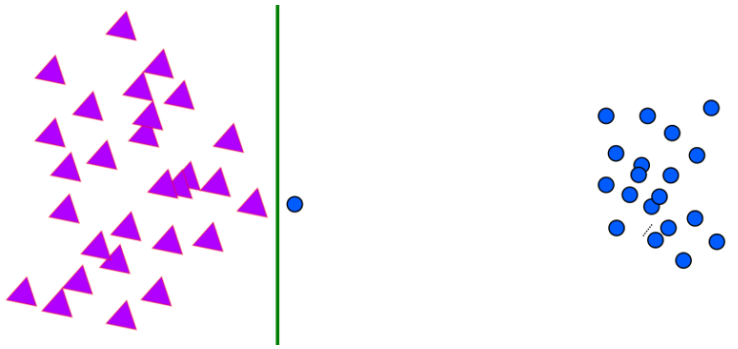
Illustrating the concept

In the 2-dimensional classification problem depicted below,



both the black and the green hyperplanes classify all instances correctly, but only the green one is (approximately) a Hard Support Vector Machine, because the distance to the nearest points (dotted lines) is maximized. Note that for a true Hard Support Vector Machine, generally there will be a bunch of nearest points with identical distance (because if there was just one, then the hyperplane could very likely be moved in such a way that distance to that point increases while distance to other points might decrease, leading to a higher minimal distance). In two-dimensional space, there will likely be two such points. On the other hand, the black hyperplane is not a Hard Support Vector Machine, and indeed the minimal margin could be slightly improved by moving it to the bottom left, such that the margin to the nearest triangle decreases but that to the nearest circle increases.

Given that Hard Support Vector Machines have an extremely discontinuous "caring function" (i.e. only care about the distance to a couple of points and are entirely indifferent to all others), it is easy to construct examples where they look silly:



You get the idea. The difference between Hard and Soft Support Vector Machines is somewhat analogous between the difference between committing to a hypothesis class ahead of time, and utilizing *Structural Risk Minimization* which has a weighting function over different hypothesis classes. In the above example, a Soft Support Vector Machine would probably bite the bullet and classify the one blue point incorrectly, in exchange for a much larger margin to the nearest correctly classified points.

Distance between a point and a hyperplane

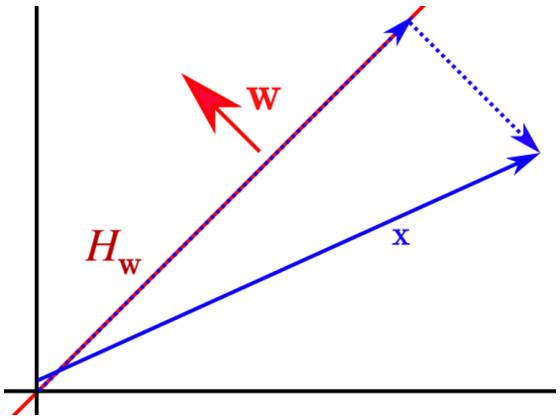
In order to implement either form of Support Vector Machines, we need to know how to compute the distance between a point and a hyperplane. Let's assume we're in R^d. In the homogeneous case (where hyperplanes have to pass through the origin), a hyperplane can be parametrized by a single vector, i.e. for $w \in R^d$, the predictor h_w is determined by the rule $h_w(x) = 1_{\langle x,w\rangle>0} \ \forall x \in R^d$ (where 1_B for a boolean statement B is 1 iff the statement is true and 0 otherwise). Let's write H_w for the actual hyperplane that corresponds to w and h_w, i.e. the set of points $\{p \in R^d \mid \langle w, p\rangle = 0\}$.

Now, what is the *distance* between a point x and H_w? It turns out that if $||w|| = 1$ (which does not restrict the set of possible hyperplanes), then it is the simplest answer one could hope for, namely $|\langle x, w\rangle|$. Let's prove this claim.

In general metric spaces, the distance between a point p and a set of points S is defined as $\inf\{d(p, q) \mid q \in S\}$. If S is a linear subspace of a larger vector space with an inner product, then this definition is still valid (inner-product spaces *are* metric spaces), but the distance *also* equals the length of the unique vector which connects S and p and is *orthogonal* to S – this is a basic result out of linear algebra. The vector orthogonal to the hyperplane H_w is simply w (see post IV), which means that the

shortest distance between H_w and x is achieved by going from x to H_w in the direction of w. Furthermore, the vectors which are orthogonal to w are exactly those on H_w.

This means that if we apply orthogonal decomposition to write x as a multiple of w plus a vector orthogonal to w, then the former be precisely the distance of x and H_w.



Now if you recall, orthogonal decomposition has the form $x = \alpha w + (x - \alpha w)$, where $\alpha = \frac{\langle x,w\rangle}{||w||^2}$ – which is just $\langle x, w\rangle$ in our case since $||w|| = 1$ – and hence the distance between x and H_w is $||\alpha w|| = ||\langle x, w\rangle w|| = |\langle x, w\rangle| \cdot ||w|| = |\langle x, w\rangle|$.

Learning Hard Support Vector Machines

As usual, let $S = ((x_1, y_1), \ldots, (x_m, y_m))$ be our training sequence. Notice that, for Hard Support Vector Machines, we wish to find a hyperplane that is *optimal* in some sense (it maximizes the distance to the nearest point) under the *constraint* that it classifies every point directly. Furthermore, everything is linear. These three concepts together sound a lot like *linear programming* (again, see post IV), and indeed we can describe the problem of finding a Hard Support Vector Machine as a linear program.

First attempt:

$$\max_{w \in R^d} (\min\{\langle x_k, w_k\rangle \mid k \in [m]\})$$

$$\text{s.t.} \quad y_k\langle w, x_k\rangle > 0 \ \ \forall k \in [m]$$

where $[m] := \{1, \ldots, m\}$. (The condition works because $y_k\langle w, x_k\rangle > 0$ iff w classifies x_k correctly according to y_k.) Unfortunately, despite our wanting to find a hyperplane, this objective function is non-linear and thus we don't have a linear program.

To remedy this, we will now perform an incredible magic trick. Recall that we assumed that $||w|| = 1$, and in that case, the distance between a point and the hyperplane is $\langle w, x \rangle$. Now if $||w|| \neq 1$, then a look at the proof we did shows that the general term for the distance is $\frac{\langle w, x \rangle}{||w||}$. Thus, if we measure the distance in terms of $\langle w, x \rangle$ while letting $||w||$ be something else, say smaller, then our measured distance (i.e. $\langle w, x \rangle$) will become smaller (it would have to be divided by a small term to make it grow into the real distance). It is as if we're zooming out of the image, but keep measuring distance with the same ruler, without normalizing it.

Thus, instead of demanding that the minimal distance be as large as possible, we can simply demand that it must be at least 1 (which also fixes the second bug in our linear program, where the constraints have a $>$ sign rather than a $\geq$ sign), and we're maximizing how *far we zoom out*. Thereby magic happens and our new program looks like so:

$$\min_{w \in R^d} \quad ||w||^2$$

$$\text{s.t.} \quad y_k \langle w, x_k \rangle \geq 1 \quad \forall k \in [m]$$

which is a proper linear program and can, therefore, be solved efficiently. And we use $||w||^2$ instead of $||w||$ just because it's simpler.

Learning Soft Support Vector Machines

Instead of demanding that all points be classified correctly, we can penalize incorrectly classified points based on *how* wrong the classification is. This can be done by changing the program to

$$\min_{w \in R^d} \quad \lambda ||w||^2 + \frac{1}{m} \sum_{k=1}^{m} \xi_k$$

$$\text{s.t.} \quad y_k \langle w, x_k \rangle \geq 1 - \xi_k \quad \forall k \in [m]$$

$$\xi_k \geq 0 \quad \forall k \in [m]$$

This reads "instead of classifying a point correctly, we're allowing you to be off by ξ_k, but try to keep the ξ_k as small as possible". We don't allow negative ξ_k, i.e. we don't reward the classifier for getting points correct with a particularly large margin, rather everything above 1 is equally fine (it will have $\xi_k = 0$ and thus won't add to the term we're trying to minimize). Furthermore, it's not clear how important zooming further

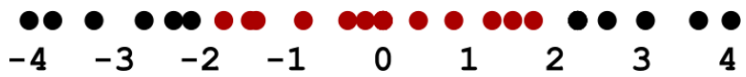
out (i.e. increasing margin) is vs having small ξ_k, so one introduces a parameter to control the tradeoff. That parameter is λ. It doesn't matter whether one puts it in front of $||w||^2$ or ~~in~~ $\sum_{k=1}^{m} \xi_k$ except that it looks prettier this way. Finally, I don't know why we're minimizing the arithmetic mean of the ξ_k rather than the sum, but given that we have λ it doesn't matter, so I decided to roll with the rule from the book.

Widening Applicability

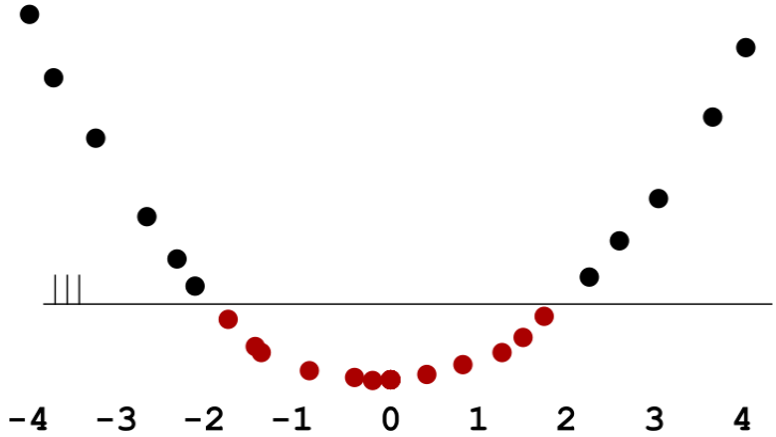
This chapter is where we look into applying linear learners to non-linear problems. Let's begin with the basic idea.

The basic idea

The basic idea is this: rather than solving non-linear problems using non-linear methods, one can map them into a linear space and then apply linear methods. The book motivates this as follows: consider the classification problem depicted below in $X = R$:



where black indicates a positive label and red a negative one. This training sequence is not linearly separable, but if we map it onto R^2 by the rule $\psi(x) = (x, x^2)$, then $\psi(S)$ looks like this:



And thus our data set has become linearly separable (via an inhomogeneous hyperplane). The black line indicates this hyperplane and the three strokes indicate that it classifies points above as positive.

In general, given any domain set X, we can choose an arbitrary embedding $\psi : X \to R^d$, or perhaps $\psi : X \to B_d(0, M)$ (the set of elements in R^d with norm at most M), and construct a linear classifier $h_w : R^d \to \{0, 1\}$ which will define a classifier $h_w^* : X \to \{0, 1\}$ by the rule $h_w^*(x) = h_w(\psi(x))$. This same construction can also be used to recover more information than just the label.

Wait but what?

Reading a description of the above gave me this familiar feeling that I don't quite get what is happening and need to rework it bottom-up. This section is me trying to describe that process.

In the above example, the new data set is indeed linearly separable. However, if ψ is allowed to be an arbitrary function, it can actually do much more work than just lifting the points onto a parabola. For example, the y-dimension isn't even used in the case above. Clearly, what really matters in that problem is the distance to the origin: every point is classified positively iff that distance is at least 2. Thus, we could just have $\psi : R \to R$ be given by $\psi(x) = |x|$, leading to the transformed 1-dimensional problem

which is separable by an even easier hyperplane. Or why not directly choose $\psi(x) = |x| - 2$? Then the data set looks like this:

and the linear classifier can just be the sign function.

(In the homogeneous case, the only two possible 1-dimensional classifiers based on hyperplanes are equivalent to the sign function and the inverse sign function, so the previous ψ does not actually lead to a separable instance, and neither does the original one which lifted the points onto the parabola. However, this is just a technicality: we are in fact allowing inhomogeneous hyperplanes, we're just pretending otherwise because inhomogeneous hyperplanes can be learned through homogeneous methods, which is actually a special case of what this chapter is about. More on this in a bit.)

What we're *actually* doing here is simply declaring that our classifier be of the form

$$h \circ \psi$$

where h is linear and ψ is an *arbitrary function.* Putting it like this, it's obvious that h doesn't have to do anything: concatenating an arbitrary function with a linear function is itself an arbitrary function; no new options are gained by this construction.

This does not imply much about the usefulness of the approach, but I think it helps to understand what is really going on. Similarly, another thing that seems important to realize is that, for learning to be successful, ψ needs to preserve meaningful properties of the data. If we just let ψ randomize some point in $\mathbb{R}^{100000}$ for each new input point it receives, then the resulting problem in $\mathbb{R}^{100000}$ will almost certainly be linearly classifiable (because the space is so high-dimensional), but the resulting classifier will not have learned how to generalize beyond the training data.

Despite those two caveats, this approach is quite useful. For one, as mentioned (and as you might recall, although I reduced the explanation to a single sentence), it can be used to learn inhomogeneous hyperplanes, by setting

$$\psi : \mathbb{R}^d \to \mathbb{R}^{d+1} \qquad \psi : (x_1, \ldots, x_d) \mapsto (x_1, \ldots, x_d, 1)$$

where the weight vector $w \in \mathbb{R}^{d+1}$ and its homogeneous hyperplane H_w found for ψ(S) will correspond to the inhomogeneous hyperplane $b + H_{w'}$ for S, where $w' := (w_1, \ldots, w_d)$ and $b := w_{d+1}$.

But there are more impressive examples, too. Let's look at one of them.

Example: polynomial regression

Consider a nonlinear regression problem, i.e. $X = \mathbb{R} = Y$ with some training sequence $S = ((x_1, y_1), \ldots, (x_m, y_m))$, and suppose we want to fit a polynomial function of reasonable degree. Thus, we choose our hypothesis class H to be the set of all (one-dimensional) *polynomial functions* of degree at most k. Rather than studying polynomials, one can now simply transform this problem into a linear one by setting

$$\psi : \mathbb{R} \to \mathbb{R}^{k+1}, \qquad \psi : x \mapsto (1, x, x^2, \ldots, x^k)$$

Then we train some linear predictor on this set, which (because it is linear) will be fully defined by some weight vector $w \in \mathbb{R}^{k+1}$, i.e. $h_w(x) = \langle w, x \rangle$. And thereby we obtain our polynomial predictor $h^*_w : \mathbb{R} \to \mathbb{R}$; it is defined by the rule $h^*_w(x) = \sum_{i=0}^{k} w_i x^i$. We have

$$h_S^{(2)}(h_w^*) = \sum_{i=1}^{m}(h_w^*(x_k) - y_k)^2 = \sum_{i=1}^{m}(\sum_{j=0}^{k} w_j^* x_k^j - y_k)^2$$

$$= \sum_{i=1}^{m}(\sum_{j=0}^{k} w_j^* \psi(x_k)_j - y_k)^2 = \sum_{i=1}^{m}(\langle w^*, \psi(x)\rangle - y_k)^2$$

$$= \ell_{\psi(S)}^{(2)}(h_w^*)$$

where $\ell_S^{(2)}$ is the empirical squared loss function. These equations show that the empirical error of both predictors are the same, which means that if h_w^* is optimal for the transformed problem $\psi(S)$, then h_w^* is optimal for the original problem.

Thus, this technique can learn polynomial predictors. This *particular* trick is only slightly more general than that: we can learn any family of functions $\{f_w\}_{w \in R^k}$ where $f_w(x) = \sum_{i=1}^{k} w_k f_k(x)$ by setting $\psi(x) = (f_1(x), \ldots, f_k(x))$ and learning $\psi(S)$ via linear regression (post IV).

Further Musings

In complexity theory, one says that problem A can be *reduced* to problem B iff, given a general solver for problem B, one can solve any instance of problem A. Usually this is done by "translating" the problem instance of A into one of B and then looking at the solution there. If this can be done, it shows that A is *at most as hard* as B.

In that sense, we have just proved that the problem of learning a polynomial function of degree at most k with minimal empirical error can be reduced to the problem of learning a linear function in k + 1 dimensional space with minimal empirical error. Put differently, we have shown that learning such a polynomial function in 1-dimensional space is *at most as hard* as learning a linear function.

So does this prove that there must exist a simple algorithm that learns polynomial predictors? Well –

– the answer to this question is trivial because we have just constructed such an algorithm. It's just that this algorithm happens to make use of ψ. Now, is that a fundamentally meaningful statement? In simple reduction arguments, one tends to

get the feeling that both problem solvers kinda do the same thing anyway. So if one understood linear predictors and 1-dimensional polynomial predictors to a point that all theory about them felt trivial, would it still feel like embedding the training data via ψ and then learning it with a linear predictor is different than learning it directly – or would it feel as if the algorithm just ends up doing the same thing anyway? If it's the latter, then ψ is just a hack that is useful for confused people, but not fundamentally necessary.

Moving away from that line of thought – given that we are all confused about Machine Learning (in the sense that we haven't "solved" the field yet), how general is this method? Are there any other Machine Learning problems where one can just split the function to approximate in two parts in a way that will somehow make learning it more feasible?

And for a final thought: it seems non-obvious that one should only use training data to learn h (in the composition $h \circ \psi$). The book mentions that ψ is a way to encode prior knowledge, and that's fine, but the same criticism of discontinuity also applies here: it's probably not realistic to say that we have perfect confidence in ψ and zero confidence in any other mapping. Furthermore, if ψ can be improved, then (because training data has diminishing returns) it seems doubtful that it is optimal to use *all* training data on h. At least, there ought to be some amount of data for which it stops being optimal. In practical terms, one might choose a family $\{\psi_i\}_{i \in [n]}$ of possible embeddings, and apply *meta-learning* to choose between them.

The next (?) post will cover *kernels,* which are a sophisticated tool to learn high-dimensional problems. They were originally meant for this post, which instead ended up reaching full length without ever getting there.

UML IX: Kernels and Boosting

(This is the ninth post in a sequence on Machine Learning based on this book. Click here for part I.)

Kernels

To motivate this chapter, consider some training sequence $S = ((x_1, y_1), \dots, (x_m, y_m))$ with instances in some domain set X. Suppose we wish to use an embedding $\psi : X \to R^d$ of the kind discussed in the previous post (i.e., to make the representation of our points more expressive, so that they can be classified by a hyperplane). Most importantly, suppose that d is significantly larger than m. In such a case, we're describing each point $\psi(x_i)$ in terms of d coordinates, even though our space only has m points, which means that there can, in some sense, only be m "relevant" directions. In particular, let

$$U := \text{span}(\psi(S_x)) = \{p \in R^d \mid \exists a \in R^m : p = \sum_{i=1}^{d} \alpha_i \psi(x_i)\}$$

where S_x is the training sequence without labels, so that $\psi(S_x) = (\psi(x_1), \dots, \psi(x_m))$. Then U is an (at most) m-dimensional subspace of R^d, and we would like to prove that we can work in U rather than in R^d.

I

As a first justification for this goal, observe that $\psi(x_i) \in U$ for all $i \in [m]$. (The symbol $[n]$ for any $n \in N$ denotes the set $\{1, \dots, n\}$.) Recall that we wish to learn a hyperplane parametrized by some $w \in R^d$ that can then be used to predict a new instance $\psi(y)$ for some $y \in X$ by checking whether $\langle w, \psi(y) \rangle > 0$. The bulk of the difficulty, however, lies in *finding* the vector w; this is generally much harder than computing a single inner product $\langle w, \psi(y) \rangle$.

Thus, our primary goals are to show that

(1) w will lie in U

(2) w can somehow be computed by only working in U

To demonstrate this, we need to look at how w is chosen, which depends on the algorithm we use. In the case of *Soft Support Vector Machines* (previous post), we choose

$$w \in \underset{w \in R^d}{\operatorname{argmin}} \left(\lambda ||w||^2 + \frac{1}{m} \sum_{k=1}^{m} \max[0, 1 - y_k \langle w, \psi(x_k) \rangle]\right)$$

This rule shows that we only care about the inner product between w and our mapped training points, the $\psi(x_i)$. Thus, if we could somehow prove (1), then (2) would seem to follow: if $w \in U$, then, according to the rule above, we would only end up caring about inner products between points that are both in U.

Therefore, we now turn to proving (1) formally. To have the result be a bit more general (so that it also applies to algorithms other than Soft Support Vector Machines), we will analyze a more general minimization problem. We assume that

$$w \in \underset{w \in R^d}{\operatorname{argmin}} \left[f_{(y_1,\ldots,y_m)}(\langle w, \psi(x_1) \rangle, \ldots, \langle w, \psi(x_m) \rangle) + R(||w||^2)\right]$$

where f is any function and R is any *monotonically non-decreasing* function. (You might verify that the original problem is an instance of this one.) Now let w^* be a solution to the above problem. Then we can use extended orthogonal decomposition[1] to write $w^* = \pi(w^*) + q$, where $\pi : R^d \to U$ is the projection onto U that leaves vectors in U unchanged and q is orthogonal to every vector in U. Then, for any $u \in U$, we have

$$\langle w^*, u \rangle = \langle \pi(w^*) + q, u \rangle = \langle \pi(w^*), u \rangle + \langle q, u \rangle = \langle \pi(w^*), u \rangle.$$

In particular, this is true for all the $\psi(x_i)$. Furthermore, since R is non-decreasing and the norm of w^* is at least as large as the norm of $\pi(w^*)$ (note that $||w^*||^2 = ||\pi(w^*)||^2 + ||u||^2$ due to the Pythagorean theorem), this shows that $\pi(w^*)$ is a solution to the optimization problem. Moreover, if R is *strictly* monotonically increasing (as is the case for Soft Support Vector Machines), then if $q > 0$, it would also be *better* than w^*, which is impossible since w^* is by assumption optimal. Thus, q must be 0, which implies that not only some but *all* solutions lie in U.

[1] Regular orthogonal decomposition, as I've formulated in the previous post, only guarantees that u is orthogonal to $\psi(w^*)$ rather than to every vector in U. But the extended version is no harder to prove. Choose some *orthonormal* basis B of U, extend it to an orthonormal basis B' of all of R^d (amazingly, this is always possible), and define π by $\pi(\sum_{i=1}^{|B'|} \alpha_i b_i) = \sum_{i=1}^{|B|} \alpha_i b_i$; i.e., just discard all basis elements that belong to B' but not B. That does the job.

II

We've demonstrated that only the inner products between mapped training points matter for the training process. Another way to phrase this statement is that, if we have access to the function

$$K_\psi : X \times X \to R \qquad\qquad K_\psi : (x, y) \mapsto \langle \psi(x), \psi(y) \rangle$$

we no longer have any need to represent the points $\psi(x_k)$ explicitly. The function K is what is called the ***kernel function***, that gives the chapter its name.

Note that K takes two arbitrary points in X; it is not restricted to elements in the training sequence. This is important because, to actually *apply* the predictor, we will have to compute $\langle w, \psi(y) \rangle$ for some $y \in X$, as mentioned above. But to *train* the predictor, we only need inner products between mapped training points, as we've shown. Thus, if we set

$$g_{k,\ell} := K(x_k, x_\ell) = \langle \psi(x_k), \psi(x_\ell) \rangle \qquad \forall k, \ell \in [m]$$

then we can do our training based solely on the $g_{k,\ell}$ (which will lead to a predictor that uses K to classify domain points.) Now let's reformulate all our relevant terms to that end.

Recall that we have just proved that $w^* \in U$. This implies that $w^* = \sum_{i=1}^{m} \alpha_i \psi(x_i)$ for the right α_i. Also recall that our objective is to find w^* in the set

$$\operatorname{argmax}_{w \in U} f(\langle w, \psi(x_1) \rangle, \ldots, \langle w, \psi(x_m) \rangle) + R(||w||^2)$$

Now we can reformulate

$$\langle w, \psi(x_k) \rangle = \langle \sum_{i=1}^{m} \alpha_i \psi(x_i), \psi(x_k) \rangle = \sum_{i=1}^{m} \alpha_i \langle \psi(x_i), \psi(x_k) \rangle = \sum_{i=1}^{m} \alpha_i g_{i,k}$$

for all $k \in [m]$, and

$$||w||^2 = \langle w, w \rangle = \langle \sum_{i=1}^{m} \alpha_i \psi(x_i), \sum_{i=1}^{m} \alpha_i \psi(x_i) \rangle = \sum_{k,\ell=1}^{m} \alpha_k \alpha_\ell g_{k,\ell}.$$

Plugging both of those into the term behind the argmax, we obtain

$$f(\sum_{i=1}^{m} \alpha_i g_{i,1}, \ldots, \sum_{i=1}^{m} \alpha_i g_{i,m}) + R(\sum_{k,\ell=1}^{m} \alpha_k \alpha_\ell g_{k,\ell})$$

This is enough to establish that one can learn purely based on the $g_{k,\ell}$. Unfortunately, the Machine Learning literature has the annoying habit of writing everything that can possibly be written in terms of matrices and vectors in terms of matrices and vectors, so we won't quite leave it there. By setting $\alpha := (\alpha_1, \ldots, \alpha_m)$ (a row vector), we can further write the above as

$$f([\alpha G]_1, \ldots, [\alpha G]_m) + R(\alpha G \alpha^T) \qquad \text{where} \qquad G = (g_{k,\ell})_{\substack{1 \le k \le m \\ 1 \le \ell \le m}}$$

or even as $f((\alpha G)) + R(\alpha G \alpha^T)$, at which point we've successfully traded any conceivable intuition for compactness. Nonetheless, the point that G is sufficient for learning still stands. G is also called the ***Gram matrix***.

And for predicting a new point $\psi(y)$, we have

$$\langle w, \psi(y) \rangle = \langle \sum_{i=1}^{m} \alpha_i \psi(x_i), \psi(y) \rangle = \sum_{i=1}^{m} \alpha_i \langle \psi(x_i), \psi(y) \rangle = \sum_{i=1}^{m} \alpha_i K(x_i, \psi(y)).$$

At this point, you might notice that we never represented U explicitly, but just reformulated everything in terms of inner products. Indeed, one could introduce kernels without mentioning U, but I find that thinking in terms of U is quite helpful for understanding *why* all of this stuff works. Note that the above equation (where we predict the label of a new instance) is not an exception to the idea that we're working in U. Even though it might not be immediately apparent from looking at it, it is indeed the case that we could first project $\psi(y)$ into U without changing anything about its prediction. In other words, it is indeed the case that $\langle w, \psi(y) \rangle = \langle w, \pi(\psi(y)) \rangle$ for all $y \in X$. This follows from the definition of π and the fact that all basis vectors outside of U are orthogonal to everything in U.

III

Kernels allow us to deal with arbitrarily high-dimensional data (even infinitely dimensional) by computing m^2 distances, and later do some additional computations to apply the output predictor – under the essential condition that we are able to evaluate the kernel function K. Thus, we are interested in embeddings ψ such that K_ψ is easy to evaluate.

For an important example, consider an embedding for *multi-variable polynomials*. Suppose we have such a polynomial of the form $p : R^n \to R$, i.e. something like

$$p(x, y, z) = x^2yz^2 + 3xyz^2 - 2x^3z^2 + 12y^2$$

where the above would be a 3-variable polynomial of degree 5. Now recall that, to learn one-dimensional polynomials with linear methods, we chose the embedding $\psi : x \mapsto (1, x, x^2, \ldots, x^k)$. That way, a linear combination of the image coordinates can do everything a polynomial predictor can do. To do the same for an arbitrary n-dimensional polynomial of degree k, we need the far more complex embedding

$$\psi : R^n \to R^{(n+1)^k} \qquad \psi : (x_1, \ldots, x_n) \mapsto \left(\prod_{i=1}^{k} x_{w(i)}\right)_{w \in \{0, \ldots, n\}^k}$$

An n-dimensional polynomial of degree k may have one value for each possible combination of its n variables such that at most k variables appear in each term. Each $w \in \{0, \ldots, n\}^k$ defines such a combination. Note that this is a sequence, so repetitions are allowed: for example, the sequence $(1, 2, \ldots, 2) \in \{0, \ldots, n\}^k$ corresponds to the term $x_1 x_2^{k-1}$. We set $x_0 = 1$ so that we also catch all terms with degree less than k: for example, the sequence $(0, 0, 0, 3, \ldots, 3)$ corresponds to the term x_3^{k-3} and the sequence $(0, \ldots, 0)$ to the absolute value of the polynomial.

For large n and k this target space is extremely high-dimensional, but we're studying kernels here, so the whole point will be that we won't have to represent it explicitly.

Now suppose we have two such instances $\psi(x)$ and $\psi(x')$. Then,

$$\begin{aligned}\langle \psi(x), \psi(x') \rangle &= \langle (\textstyle\prod_{i=1}^{k} x_{w(i)})_{w \in \{0,\ldots,n\}^k}, (\prod_{i=1}^{k} x'_{w(i)})_{w \in \{0,\ldots,n\}^k} \rangle \\ &= \textstyle\sum_{w \in \{0,\ldots,n\}^k} \langle \prod_{i=1}^{k} x_{w(i)}, \prod_{i=1}^{k} x'_{w(i)} \rangle \\ &= \textstyle\sum_{w \in \{0,\ldots,n\}^k} \prod_{i=1}^{k} x_{w(i)} x'_{w(i)}\end{aligned}$$

And for the crucial step, the last term can be rewritten as $(\sum_{i=0}^{n} x_i x'_i)^k$ – both terms include all sequences $x_i x'_i$ of length k where $i \in \{0, \ldots, n\}$. Now (recall that $x_0 = x'_0 = 1$) this means that the above sum simply equals $(1 + \langle x, x' \rangle)^k$. In summary, this calculation shows that

$$K(x, x') := (1 + \langle x, x' \rangle)^k = \langle \psi(x), \psi(x') \rangle \qquad \forall x, x' \in X$$

Thus, even though ψ maps points into the very high-dimensional space $R^{(n+1)^k}$, it is nonetheless feasible to learn a multi-polynomial predictor through linear methods, namely by embedding the values via ψ and then ignoring ψ and using K instead. The gram matrix G will consist of m^2 entries, where for each, a term of the form $(1 + \langle x, x' \rangle)^k = (1 + \sum_{i=1}^{n} x_i x'_i)^k$ has to be computed. This doesn't look that scary! Even for relatively large values of d, k, and m, it should be possible to compute on a reasonable machine.

If we do approach learning a multi-dimensional polynomial in this way, then (I think) there are strong reasons to question in what sense the embedding ψ actually *happens* – this question is what I was trying to wrap my head around at the end of the previous post. It seemed questionable to me that ψ is fundamental even if the problem is learned without kernels, but even more so if it is learned with them.

And that is all I have to say about kernels. For the second half of this post, we'll turn to a largely independent topic.

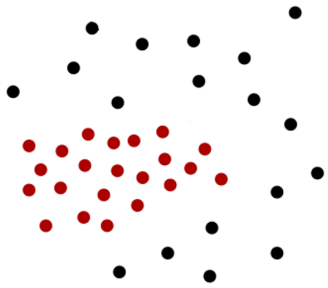
Boosting

Boosting is another item under the "widening the applicability of classes" category, much like the ψ from earlier.

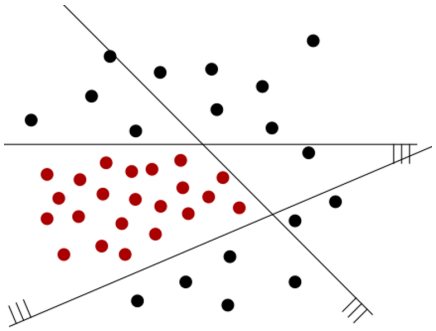
I

This time, the approach is not to expand the representation of data and then apply a linear classifier on that representation. Instead, we wish to construct a complex classifier as a *linear combination of simple classifiers*.

When hyperplanes are visualized, it is usually understood that one primarily cares about hyperplanes in higher-dimensional spaces where they are much more expressive, despite the illustration depicting an instance in 2-d or 3-d. But this time, think of the problem instance below in literal 2-d space:

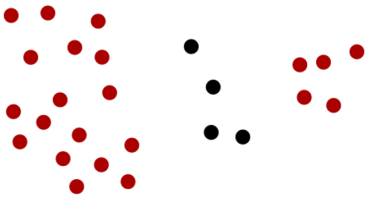


No hyperplane can classify this instance correctly, but consider a combination of these three hyperplanes:

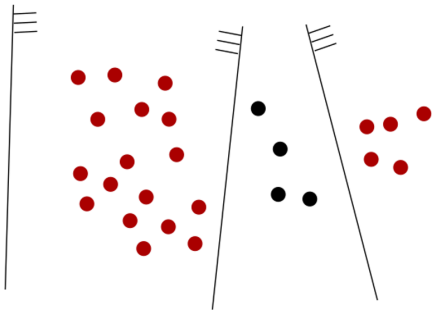


By letting $h(p) = \sigma_{sign}(h_1(p) + h_2(p) + h_3(p) - 2.5)$ where h_i is the predictor corresponding to the i-th hyperplane and σ_{sign} is the sign function, we have constructed a predictor h which has zero empirical error on this training instance.

Perhaps more surprisingly, this trick can also learn non-convex areas. The instance below,



will be classified correctly by letting $h(p) = \sigma_{sign}(h_1(p) + 2h_2(p) + 2h_3(p))$, with the h_i (ordered left to right) defined like so:



These two examples illustrate that the resulting class is quite expressive. The question is, how to learn such a linear combination?

II

First, note that hyperplanes are just an example; the framework is formulated in terms of a learning algorithm that has access to a ***weak learner***, where

> An algorithm A is called a γ-weak learner for a hypothesis class H iff there is a function $w^* : (0, 1) \to \mathbb{N}$ such that, for any probability distribution D over $X \times Y$ and any failure probability $\delta \in (0, 1)$, if S consists of at least $w^*(\delta)$ i.i.d. points sampled via D, then with probability at least $1 - \delta$ over the choice of S, it holds that $\ell(A(S)) \leq \frac{1}{2} - \gamma$.

If you recall the definition of PAC learnability back from chapter 1, you'll notice that this is very similar. The only difference is in the error: PAC learning demands that it be arbitrarily close to the best possible error, while a weak learner merely has to bound it away from $\frac{1}{2}$ by some fixed amount γ, which can be quite small. Thus, a weak learner is simply an algorithm that puts out a predictor that performs a little bit better than random. In the first example, the upper hyperplane could be the output of a weak learner. The term "boosting" refers to the process of upgrading this one weak learner into a better one, precisely by applying it over and over again under the supervision of a smartly designed algorithm –

– which brings us back to the question of how to define such an algorithm. The second example (the non-convex one) illustrates a key insight here: repeatedly querying the weak learner on the unaltered training instance is unlikely to be fruitful, because the third hyperplane by itself performs worse than random, and will thus not be output by a γ-weak learner (not for any $\gamma \in \mathbb{R}_+$). To remedy this, we somehow need to *prioritize the points we're currently getting wrong.* Suppose we begin with the first two hyperplanes. At this point, we have classified the left and middle cluster correctly. If we then weigh the right cluster sufficiently more strongly than the other two, eventually, h_3 will perform better than random. Alas, we wish to adapt our weighting of training points dynamically, and we can do this in terms of a probability distribution over the training sequence.

Now the roadmap for defining the algorithm which learns a predictor on a binary classification problem via boosting is as follows:

- Have access to a training sequence S and a γ-weak learner A_γ
- Manage a list of weak predictors which A_γ has output in previous rounds
- At every step, hand A_γ the training sequence S along with some distribution $D^{(t)}$ over S, and have it output a γ-weak predictor h_{t+1} on the problem $(S, D^{(t)})$, where each point in S is taken into account proportional to its probability mass.
- Stop at some point and output a linear combination of the h_i

The particular algorithm we will construct is called ***Ada-Boost***, where "Ada" doesn't have any relation to the programming language, but simply means "adaptive".

III

Let's first look into how to define our probability distribution, which will be the most complicated part of the algorithm. Suppose we have our current distribution $D^{(t)}$ based on past predictors $h_1, \dots, h_{t-1}$ output by A_γ, and suppose further that we have computed weights $w_1, \dots, w_{t-1}$ such that w_i measures the quality of h_i (higher is better). Now we receive a new predictor h_t with quality w_t. Then we can define a new probability distribution $D^{(t+1)}$ by letting

$$D^{(t+1)}((x_i, y_i)) \propto D^{(t)}((x_i, y_i)) \cdot e^{-w_t y_i h_t(x_i)} \quad \forall i \in [m]$$

where we write $\propto$ rather than $=$ because the term isn't normalized; it will equal the above scaled such that all probabilities sum to 1.

The term $y_i h_t(x_i)$ is 1 iff predictor h_t classified x_i correctly. Thus, the right component of the product equals e^{-w_t} iff the point was classified correctly, and e^{w_t} if it wasn't. If h_t is a bad predictor and w_t is small, say 10^{-3}, the two terms are both close to 1, and we don't end up changing our weight on (x_i, y_i) very much. But if h_t is good and w_t is large, the old weight $D^{(t)}((x_i, y_i))$ will be scaled significantly upward (if it got the point wrong) or downward (if it got the point right). In our second example, the middle hyperplane performs quite well on the uniform distribution, so w_2 should be reasonably high, which will cause the probability mass on the right cluster to increase and on the two other clusters to decrease. If this is enough to make the right cluster dominate the computation, then the weak learner might output the right hyperplane next. If not, it might output the second hyperplane again. Eventually, the weights will have shifted enough for the third hyperplane to become feasible.

IV

Now let's look at the weights. Let $\epsilon_t = \ell_S^{0-1}(h_t)$ be the usual empirical error of h_t, i.e., $\epsilon_t = \frac{1}{m}|\{(x, y) \in S \mid h_t(x) \neq y\}|$. We would like w_i to be a real number, which starts close to 0 for ϵ_t close to $\frac{1}{2}$ and grow indefinitely for ϵ_t close to 0. One possible choice is $w_t := \frac{1}{2}\ln(\frac{1}{\epsilon_t} - 1)$. You can verify that it has these properties – in particular, recall that h_t is output by a weak learner so that its error is bounded away from $\frac{1}{2}$ by at least γ. Because of this, $\frac{1}{\epsilon_t}$ is larger than 2 so that $\frac{1}{\epsilon_t} - 1$ is larger than 1 and w_t is larger than 0.

V

To summarize,

AdaBoost (A_γ : weak learner, S : training sequence, T : N_+)

$D^{(0)} \leftarrow (\frac{1}{m} \cdots \frac{1}{m})$

for (t ← 1 to T) **do**

$h_t \leftarrow A_\gamma(S, D^{(t-1)})$

$\epsilon_t \leftarrow \frac{1}{m}|\{(x, y) \in S \mid h_t(x) \neq y\}|$

$w_t \leftarrow \frac{1}{2} \ln\left(\frac{1}{\epsilon_t} - 1\right)$

$D^{(t)} \leftarrow \text{normalize}\left(\left(D_i \cdot e^{-w_t y_i h_t(x_i)}\right)_{i \in [m]}\right)$

endfor

return $f : x \mapsto \sigma_{sign}(\sum_{t=1}^{T} w_t h_t(x))$

end

VI

If one assumes that A_γ always returns a predictor with error at most $\frac{1}{2} - \gamma$ (recall that it may fail with probability δ), one can derive a bound on the error of the output predictor. Fortunately, the dependence of the sample complexity on δ is only logarithmic, so δ can probably be pushed low enough that A_γ is unlikely to fail even if it is called T times.

Now the error bound one can derive is $e^{-2\gamma^2 T}$. Looking at this, it has exactly the properties one would expect: a higher γ pushes the error down, and so do more rounds of the algorithm. On the other hand, doing more rounds increases the chance of overfitting to random quirks in the training data. Thus, the parameter T allows one to balance the overfitting vs. underfitting tradeoff, which is another nice thing about AdaBoost.The book mentions that Boosting has been successfully applied to the task of classifying gray-scale images into 'contains a human face' and 'doesn't contain a human face'. This implies that human faces can be recognized using a set of quantitative rules – but, importantly, rules which have been generated by an algorithm rather than constructed by hand. (In that case, the weak learner did not return hyperplanes, but simple predictors of another form.) In this case, the result fits with my intuition (that face recognition is the kind of task where a set-of-rules approach will work). It would be interesting to know how well boosting performs on other problems.

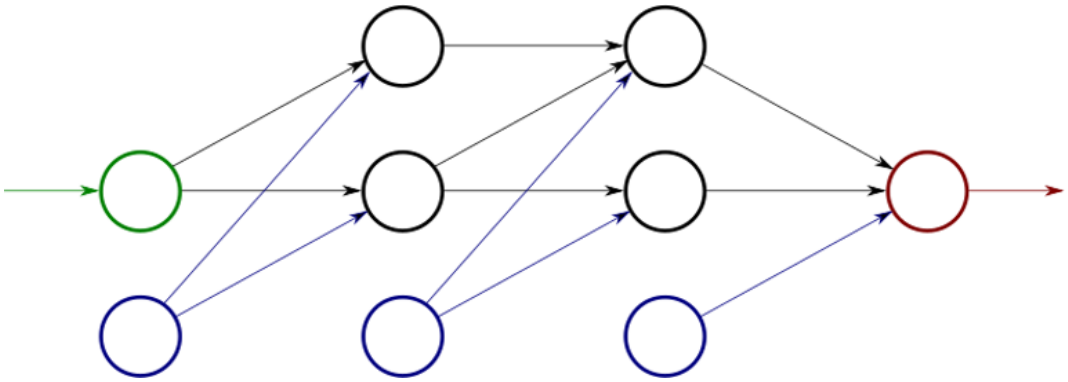
A Simple Introduction to Neural Networks

This post will discuss *what* neural networks are (chapter I), *why* they work (chapter II), and *how* they are trained (chapter III). It serves as the tenth entry in a sequence on Machine Learning, but it's written to be accessible without having read any previous part. Most of chapters I and II also doesn't require any advanced math.

Note that ***bold and italics*** means "I am introducing a new Machine Learning term."

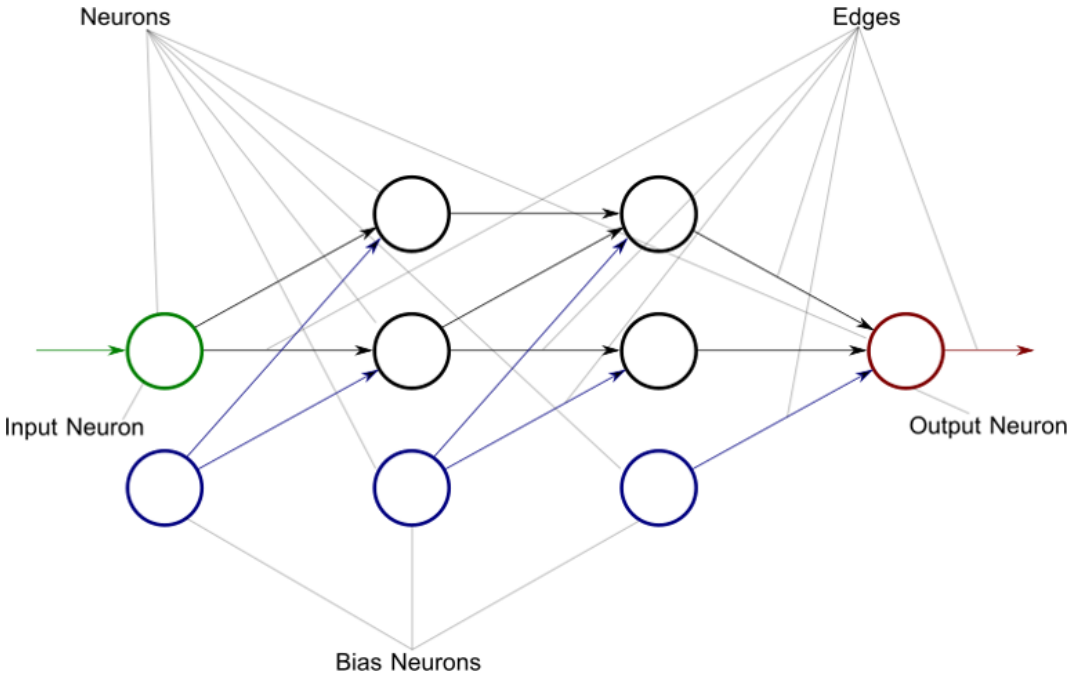
I

Meet the ***neural network***:

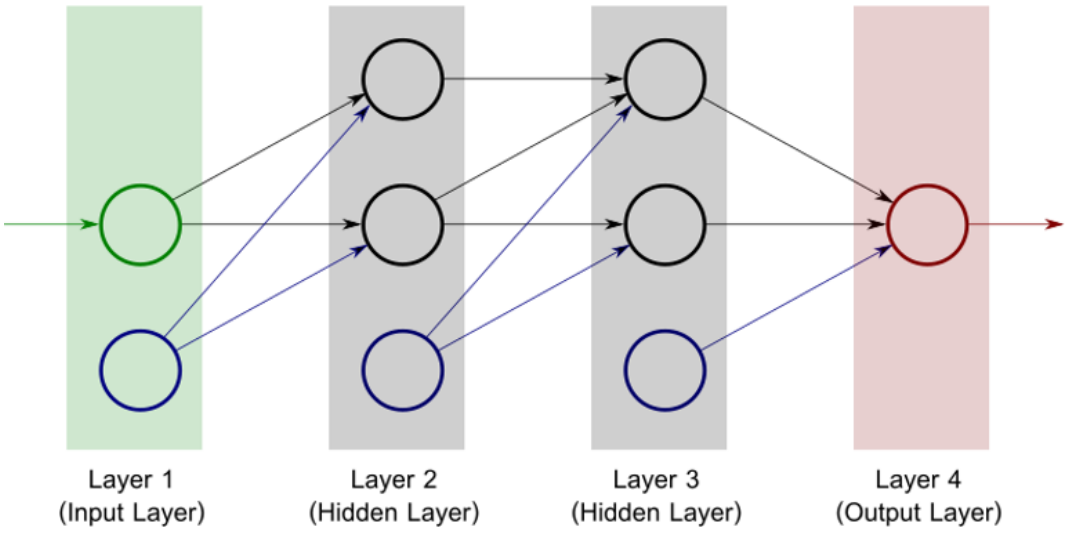


The term "neural network" describes a class of Machine Learning predictors which is inspired by the architecture of the human brain. It is an extremely important class: AlphaGo, AlphaGo Zero, AlphaStar, and GPT-2 are all based on neural networks.

The network above will be our running example throughout this post. Let's begin by defining its components. The blobs are called ***neurons***. The arrows are called **edges**. Together, they define the *graph underlying the neural network.* (This usage of the term "graph" has nothing to do with a function graph – instead, a graph is simply a bunch of nodes with arrows between them.) If this graph has no *cycles* (= edges going around in a circle), the neural network is also called ***feed-forward.*** For this post, you can assume networks are always feed-forward. The green neurons are called ***input neurons,*** the red neurons are called ***output neurons,*** and the blue neurons are called ***bias neurons***.

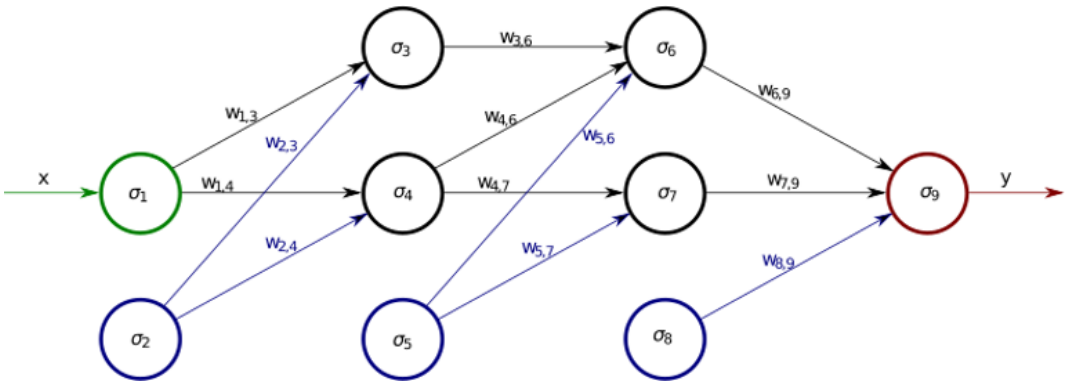


If it is possible to divide the entire set of neurons into an ordered list of subsets, such that each neuron only points towards neuron in the subset next in line, the network is called ***layered***. For this post, you can assume networks are always layered. However, note that non-layered networks do exist: a network with edges A → B and A → C and B → C cannot be divided into layers.



As you can see, each neuron only points towards neurons in the next higher layer. Note that the first blue neuron, despite being in the input layer, isn't an input neuron.

So far, our view has been simplified. In reality, there's more going on:



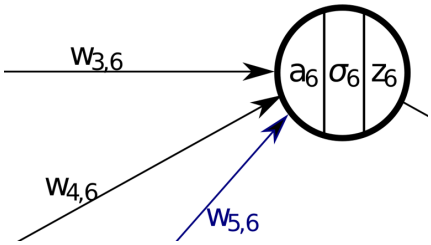
The $w_{\bullet,\bullet}$ written on the edges are numbers called *weights*. They will determine how important the value of the incoming edge is. The $\sigma_\bullet$ written inside of the neurons are *functions* (in the classical sense: they take a value as input and output something else). From here on, if I talk about "the fifth neuron," I mean the neuron with σ_5 written in it.

This picture is a good guide to explain how a neural network works.

First, an input value x (think of a number) is fed into the input neuron, which forwards it to neurons three and four. At the same time, neuron two (the bias neuron) sends the value 1 to neurons three and four (sending the number 1 is all that bias neurons ever do). At this point, the third neuron has received two values – it now multiplies them with the weights on the respective edges, so it multiplies the value x coming from the first neuron with $w_{1,3}$ and the number 1 coming from the bias neuron with $w_{2,3}$. Then it adds both values together and applies the function σ_3 to them. The result of this process is $\sigma_3(w_{1,3} \cdot x + w_{2,3} \cdot 1)$. The fourth neuron has also received two values, and it does the same thing (using the weights $w_{1,4}$ and $w_{2,4}$), leading to the term $\sigma_4(w_{1,4} \cdot x + w_{2,4} \cdot 1)$. With that done, the third neuron sends its value (the one we just computed) to the sixth neuron, the fourth sends its value to both the sixth and the seventh, and the fifth neuron (the bias neuron) sends the number 1 to both the sixth and the seventh neuron. Now they apply their weights, then their functions, and so on. Eventually, the ninth neuron receives a value, applies σ_9 to it, and outputs the result, which is the output of the network.

Recall that the blue neurons are called *bias neurons* – that is because they have no incoming edges. Instead of computing something, they always output 1 and thus "bias" the weighted sum, which arrives at the neurons in the next layer, by a constant factor. Each hidden layer has exactly one bias neuron, and the input layer also has one. That way, each neuron except the input neurons has an incoming edge from a bias neuron. Also, note that the input neuron doesn't apply any function to the input it receives (but all other neurons do).

To describe this process more formally, we need some additional notation for the value that goes into a neuron (the weighted sum of the values on its incoming edges) and the value which comes out (after the function is applied to it). Let's zoom into the sixth neuron:

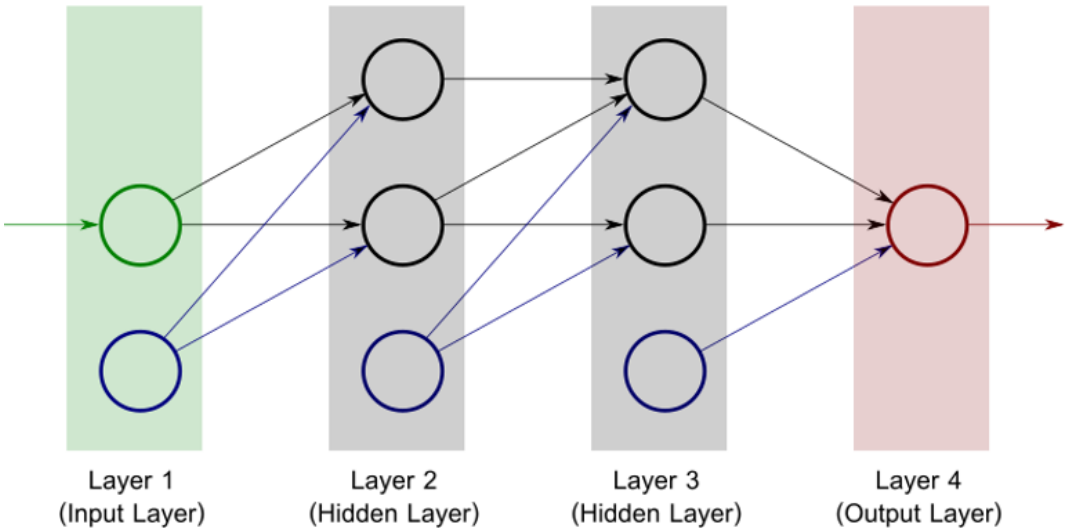


When the input values arrive, first the weighted sum a_6 is computed, then σ_6 is applied to it, and that will be the output value z_6. Thus,

$$z_6 = \sigma_6(a_6) = \sigma_6(w_{3,6} \cdot z_3 + w_{4,6} \cdot z_4 + w_{5,6}).$$

The new thing here is that we express the output of any neuron in terms of the outputs of neurons in the previous layer; thus, the equation can describe the output of any neuron in the network. Notice that the value z_5 is missing – that's because the fifth neuron is a bias neuron, which means it puts out the value 1, which means that $w_{4,5} \cdot z_5$ is the same as $w_{4,5}$. Also, note that the image shows the weights of the incoming edges, but not the weight of the outgoing edge: the weights of an edge should always be thought of as belonging to the neuron the edge is heading into.

Recall that our neural network can be divided into layers:



This is useful for evaluating the network – it allows us to do so layer-by-layer. In the beginning, we just know the input value(s) x_i. We set

$$z_k = x_k$$

for each neuron k in layer 1 (the input layer). This is just a relabeling – nothing has happened yet. But now, assuming we have all the output values for some layer ℓ, we compute the output values for each neuron k in layer $\ell + 1$ by setting

$$z_k = \sigma_k(\sum_{i \in I_k} w_{i,k} z_i)$$

where I_k is the set of indices of neurons that have an edge pointing towards neuron k. Since this equation applies for any layer, it allows us to evaluate the entire network, one layer at a time. The output values z_k of the final layers will be the output values y_k of the network. If you're unfamiliar with the $\sum$ notation, then never mind this – the equation merely states what we have already discussed, namely that the output of each neuron is computed in terms of the outputs of neurons in the previous layer. The purpose of writing it down as a single equation is to make it less ambiguous and more compact.

II

Now you know what a neural network is and how it is computed. But what are they good for, and why do they work so well?

Let's start with the first question. If you already know what they're good for, you can skip forward to section II.II for the second one. Otherwise, here's a quick overview.

II.I

If all values (input values, output values, and values sent between neurons) are regular numbers, then the entire network *implements a function* $f : \mathbb{R}^n \to \mathbb{R}^m$, where n is the number of neurons in the input layer and m the number of neurons in the output layer. In our case, $n = m = 1$. The notation $f : \mathbb{R}^n \to \mathbb{R}^m$ means that f always takes an element in $\mathbb{R}^n$, which is a vector of n numbers (all numbers fed to input neurons), and returns an element in $\mathbb{R}^m$, which is a vector of m numbers (all numbers returned by output neurons). And it does this for any possible input – so for any n input numbers, it returns m output numbers. To say that the network "implements a function" means that we can abstract away from how exactly it works: in the end, its behavior is fully described by such a function $f : \mathbb{R}^n \to \mathbb{R}^m$. This is nice because functions are more well-understood objects than neural networks.

So the utility of a neural network is that it lets us evaluate this function f. This can be valuable if f does useful things. As an example, we can suppose that f takes the source code of an image as input and outputs a single bit indicating whether or not the image contains a cat. In that case, if we suppose the image is gray-scale with 1 byte per pixel and 200 by 200 pixels, the function is of the form $f : \{0,1\}^{8\cdot 200\cdot 200} \to \{0,1\}$. This notation means that it takes a vector of $8 \cdot 200 \cdot 200$ bits as input (which determines an image), and outputs a single bit, where 1 means "cat" and 0 means "no cat". And again, it does this for every possible vector of $8 \cdot 200 \cdot 200$ bits. Having access to this function would be useful, and thus, having a neural network that implements this function – or more realistically, which implements a *similar* function (with some flaws) – would also be useful.

Another example could be the function $f : \{0,1\}^{8\cdot 280} \to \{0,\dots,10\}$. This function takes a vector of $8 \cdot 280$ bits as input which encodes a tweet (280 character limit, each character is encoded by 1 byte, weird symbols are ignored), and outputs a number between 0 and 10 that rates how likely this tweet is to violate twitter's terms of service. This function would be very useful to learn – and, in fact, Jack Dorsey mentioned that they are using Machine Learning for stuff like this (whether they use neural networks, I don't know, but they probably do).

How to train a neural network is something we discuss in detail in chapter III. For now, I'll only mention that it works based on *training data.* That is, we have some sequence of examples $(x_1, y_1), \dots, (x_n, y_n)$ where the x_i are inputs to the neural network (in the twitter example, tweets), and the y_i are outputs (in the twitter example, scores from 0 to 10 that have been assigned by a human). We then train the network to do well on the training data, and hope that this will lead it to also do well in the real world (in the twitter example, we hope that it also labels new tweets accurately). However, this description is not specific to neural networks – it applies to all instances of ***supervised learning***, which is a big part of Machine Learning – and there are many different machine learning techniques out there. So what is it about neutral networks in particular that make them so powerful? We know that they are inspired by the human brain, but that is hardly an answer.

II.II

This feels like an apt time to add the disclaimer that I am very much not an expert. The following is a result of my searching the existing literature for the best answers, but it is an incomplete search evaluated with limited knowledge.

That said, one possible answer is to point towards the *expressibility theorems* we have on neural networks. Most notably, there is the universal approximation theorem – it states that a feed-forward neural network with only a single hidden layer (recall that, in our example, we had two hidden layers) can approximate any continuous & bounded function under some reasonably mild conditions. Sounds pretty good – until you realize that the proof basically consists of brute-forcing the approximation. This

makes it kind of like pointing out that the class of all hypotheses such that there is a C program of size at most 1 GB implementing them can also solve a wide variety of practically relevant problems. A fairly damning fact is that the neural networks constructed in the proof have exponentially many neurons in the hidden layer – just like there are exponentially many C programs of with at most n bits. ("Exponential" means that the number is an exponential function of n, in this case, 2^n.) Results stating that something can be done with exponential resources are generally unimpressive.

A much simpler result that can be rigorously proved in just a few lines states that a neural network (with only one hidden layer) can implement any function of the kind $f : \{0,1\}^n \to \{0,1\}$. This sort of function is called a *boolean function*, and it can be thought of as taking in n input bits and outputting a binary yes/no. More precisely, the theorem states that there is *an architecture for a neural network* (i.e., a graph) such that, for each possible function $f : \{0,1\}^n \to \{0,1\}$, there is a set of weights such that the neural network defined by that architecture & that set of weights implements that function. Moreover, every neuron in this network applies the same, very simple function σ. But once again, the number of neurons in the hidden layer of that architecture is exponential in n. And there is also a second theorem which shows that this is a *lower bound* – an (architecture for a) neural network implementing this kind of function *has to have* an exponential number of neurons in n, no matter how smartly it is designed.

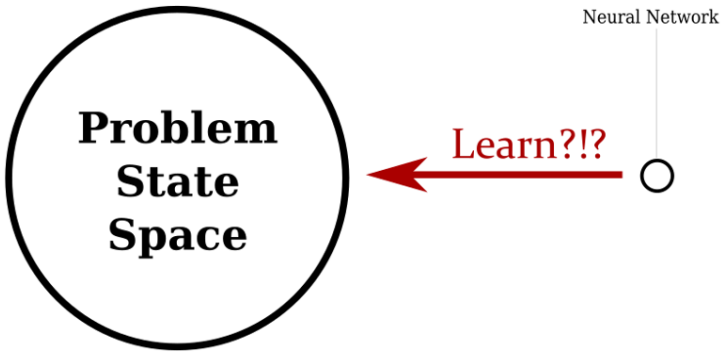
Verdict: unsatisfying.

II.III

If the universal approximation theorem cannot answer the question, what then? The best explanation I've found is from a paper co-authored by Max Tegmark – the guy who wrote Our Mathematical Universe and is the president of the Future of Life Institute. It's titled "Why does deep and cheap learning work so well?" and it begins with the observation that even simple tasks such as image-classification are, in some sense, impossible to solve by a neural network. As mentioned, a 200-by-200 bit gray-scale image with just a single byte for each pixel defines the input space $\{0,1\}^{8\cdot 200\cdot 200}$. That space has size $2^{320.000}$ (i.e., it consists of $2^{320.000}$ encodings for images), which is roughly $10^{10.000}$. (Note that the dot is not a decimal point.) But there's no reason to be so modest; neural networks are, in fact, used to classify much larger images. Let's say our images are 1000-by-1000 pixels with 4 bytes per pixel. In that case, the space has $2^{32.000.000}$ elements, which is about $10^{10.000.000}$. On the other hand, a neural network with, say, 1000 neurons only has n^{1000} possible combinations, where n is the number of different values each weight can take. Even if we're overly generous and assume that each weight can lead to a 100 meaningfully different configurations, that would only provide us with the ability to differentiate between at

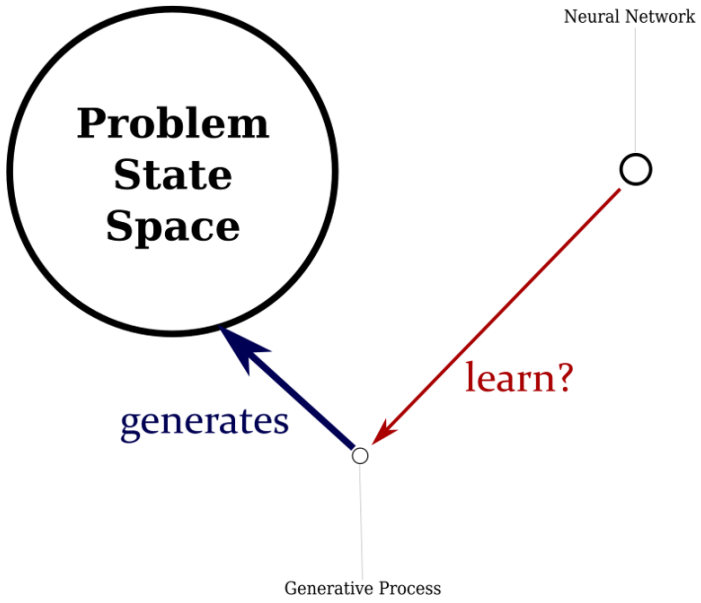
most $10^{10.000}$ different cases – barely enough for tiny gray-scale images, but nowhere near enough for decently sized images. This follows because a neural network cannot possibly differentiate more images than it has different configurations. And we didn't even talk about the amounts of training data that are available in practice, which are nowhere near as large.

This suggests that the situation in practice looks something like this:



And one is left to wonder how this could ever work out.

The solution the paper suggests is to replace the above image with this one:



That is, the problem state space has actually been generated by a physical process that is far simpler to specify than the state space itself and usually even simpler than the neural network. For example, cats (or even pictures of cats) are pretty complicated mathematical objects, but the process which has generated cats (namely evolution) is relatively simple. Similarly, the phrase "draw cat pictures," which one can

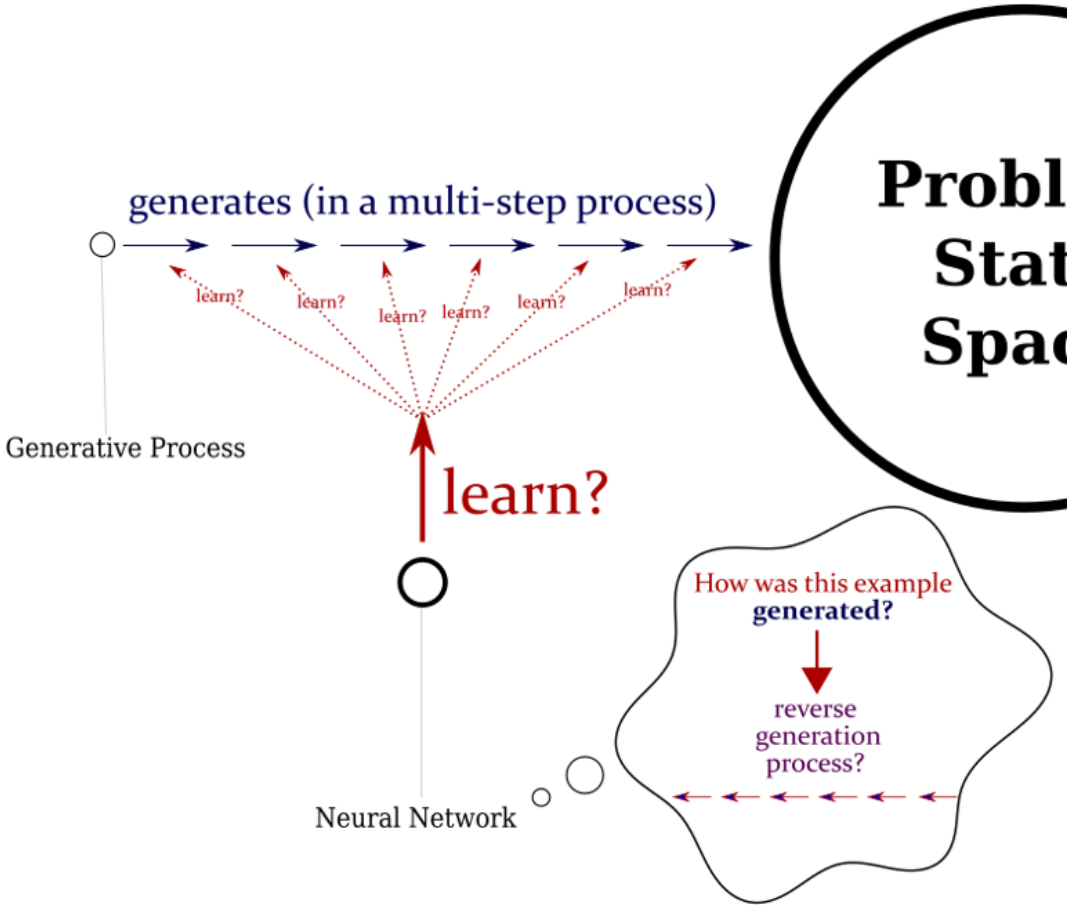
give to a bunch of artists, is far simpler than the images they will produce. Thus, while it is true that a neural network with ~1000 neurons can only give reasonable answers to a tiny subset of images in image-space, this is acceptable: most of the images we care about are contained in that subset. Perhaps most strikingly, the standard model of physics only has 32 parameters, according to the paper.

Now, you might recall that the title of the paper mentioned "deep learning." This term refers to a neural network with a lot of hidden layers, as supposed to just one or two. The fact that deep learning has had massive success in practice stands in contrast to the universal approximation theorem, which argues using a network of only a single hidden layer. It is probably fair to say that "shallow" networks are more similar to other kinds of predictors and less analogous to what the human brain does.

So why neural nets – and more specifically, deep learning – rather than any other technique? On this point, Lin et al. (Tegmark is part of the "et al.") argue that that nature is fundamentally *hierarchical*:

> One of the most striking features of the physical world is its hierarchical structure. Spatially, it is an object hierarchy: elementary particles form atoms which in turn form molecules, cells, organisms, planets, solar systems, galaxies, *etc*. Causally, complex structures are frequently created through a distinct sequence of simpler steps.

So we can rethink the picture yet again to arrive at something like this:



So the idea is not just that the small complexity of the generation process makes it feasible for neural networks to accomplish tasks such as recognizing faces, but also that it may *learn by reversing this generative process*. Furthermore, since this process naturally takes place in multiple steps, the most effective neural networks will themselves be performing their computations in multiple steps. The paper also proves something to the effect that generative hierarchies can be "optimally reversed one step at a time," but I can't tell you what exactly that means since the statement is formalized in information theory.

Now, do "multiple steps" translate into "multiple layers"? The answer to that is a pretty clear yes (here is where we need to introduce a bit of math). Notice that, even if a neural network has many input notes, it can be considered to modify just a single object – namely a vector $x \in \mathbb{R}^n$, where n is the number of input neurons. Now the entire function f which the neural network computes can be written down like so:

$$f = \sigma_\ell \circ A_\ell \circ \cdots \circ \sigma_2 \circ A_2$$

where ℓ is the number of layers, the A_i are linear functions ($\to$ matrices), and the σ_i are the functions that the neurons implement, applied coordinate-wise. The symbol $\circ$ means that we first apply the right function, then the left one. If the input vector x is fed into the neural network, it first passes unchanged to the input neurons (recall that, unlike all other neurons, they don't compute a function). Then they are scaled according to the weight values between layers 1 and 2, which corresponds to the linear transformation A_2, our first transformation. (It's indexed with 2 because it leads into layer 2.) Now the input neurons in the second layer (the first hidden layer) apply their functions, which corresponds to the component-wise application of σ_2 (provided all neurons on the layer implement the same function, which is an assumption we make here). Then the outputs are scaled according to the weight values between layers 2 and 3, which corresponds to the linear transformation A_3, and so on. In the end, the output neurons apply their function σ_ℓ component-wise.

Hence the neural network does indeed apply one (pair of) transformations per layer. However, no-one knows what neural networks are *actually* doing most of the time, so all of this is still largely speculative.

Nonetheless, here is an interesting observation that backs up Lin et al.'s claims. (Skip this part if you have never taken a course in Linear Algebra.) Suppose we declare that the neurons aren't allowed to do anything (i.e., they just implement the identity function). In that case, the neural network decomposes into the transformations

$$f = A_\ell \circ \cdots \circ A_2$$

Concatenating many linear functions yields itself a linear function, so this neural network does nothing except apply a single matrix to the input vector, namely the

matrix $A_\ell \cdots A_2$.

Let's suppose the input elements form a matrix N, and the goal of the network is to multiply that matrix with another matrix M. Then, this can be trivially realized with just an input and an output layer (no hidden layers), by setting $A_\ell = A_2 = M$. This corresponds to the standard way of doing matrix multiplication, which requires n^3 operations. But there are also (much more sophisticated) algorithms with funny time complexities such as $O(n^{2.3728639})$ – point being, they're faster for very large matrices.

Thus, since neural networks are universal approximators, there exists a neural network that implements matrix multiplication with similar time complexity. This is impossible to replicate on a shallow network. (Related: proofs in linear algebra often argue by decomposing a matrix into smaller ones.)

The paper calls these results "no-flattening theorems" and goes on to reference a bunch of other examples. So it is possible to rigorously prove that at least some specific neural networks cannot be flattened without losing efficiency. And it's also worth noting that there exists a version of the universal approximation theorem which manages to restrict a network's width in favor of more hidden layers.

That's all I have on the why question. Has it been a satisfying answer? I would vote a firm no – it seems a rather long way from being satisfactory. Nonetheless, I at least feel like I now have some nonzero insight into why neural networks are powerful, which is more than I had before reading the paper.

III

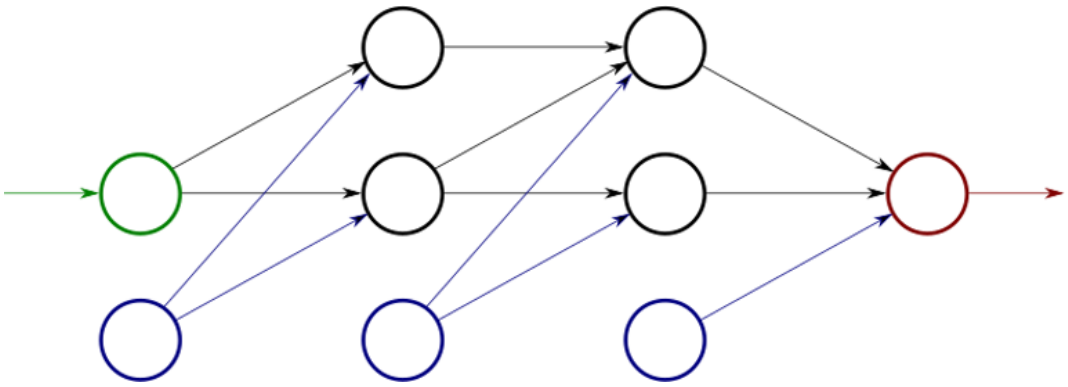
This final chapter is where we look at *how* to train a neural network. From this point onward, the post will get a lot more mathy – in particular, it assumes you know derivatives and the chain rule. If you don't, feel free to skip right to the end.

First, we need to discuss *which part* of the network we wish to learn. In principle, there are many possible approaches, but in practice, one usually fixes everything except the weights. That is, one fixes

- the underlying graph (i.e., all neurons and edges)
- the functions σ_k (which aren't very complicated, they might even all be the same)

and then looks for the set of weights w_k with which the network performs best. The above is also called the ***architecture*** of the network. Any given architecture can implement a wide variety of different functions (by giving it different weights).

As mentioned, we assume that we're in the setting of *supervised learning*, where we have access to a sequence $S = ((x_1, y_1), \ldots, (x_m, y_m))$ of training examples. Each x_i is an input to the network for which y_i is the corresponding correct output.

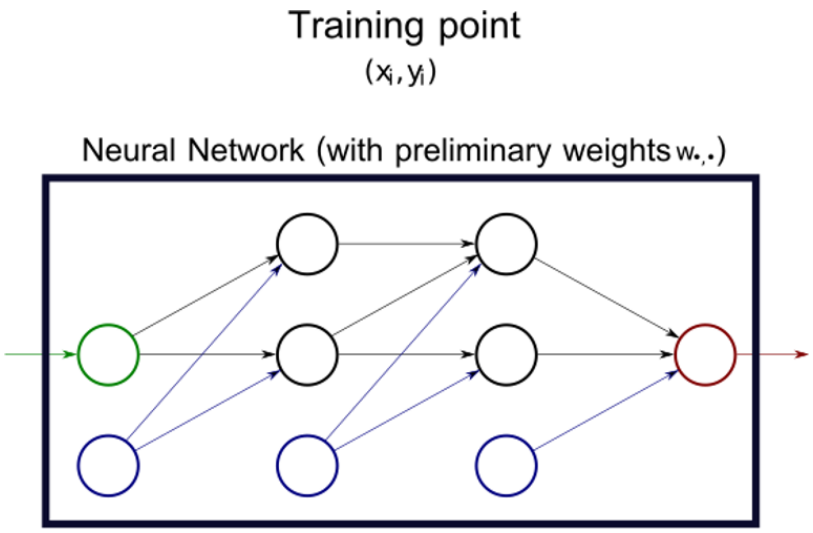


The idea is now to train the neural network to do well on the training data S and hope that this will cause it to do well in the real world.

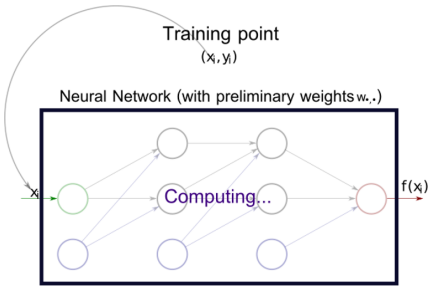
How do we do that? Given infinite computing power, there is a simple algorithm: we choose some finite encoding for real numbers, say 64-bit encoding, and perform an exhaustive search over all possible combinations of weight values; then we test the network with each one and stick with the combination that works best. Unfortunately, there are $2^{64 \cdot 12}$ many combinations of weight values, so this approach would require $2^{64 \cdot 12}$ rounds, where each round consists of testing the neural network with those weights on all of S. This is infeasible even for our small network, therefore we need a more efficient approach. The algorithm we will study is called ***backpropagation***, and it relies on something called ***Stochastic Gradient Descent***.

III.I

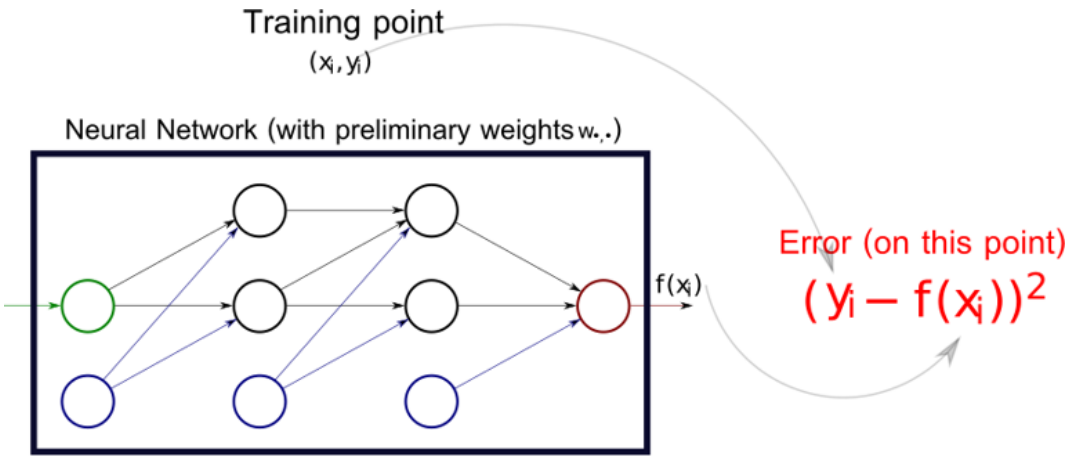
Here is how it works. Suppose we give the network some preliminary weights so that it is fully defined, and we focus on one particular training point, (x_i, y_i).



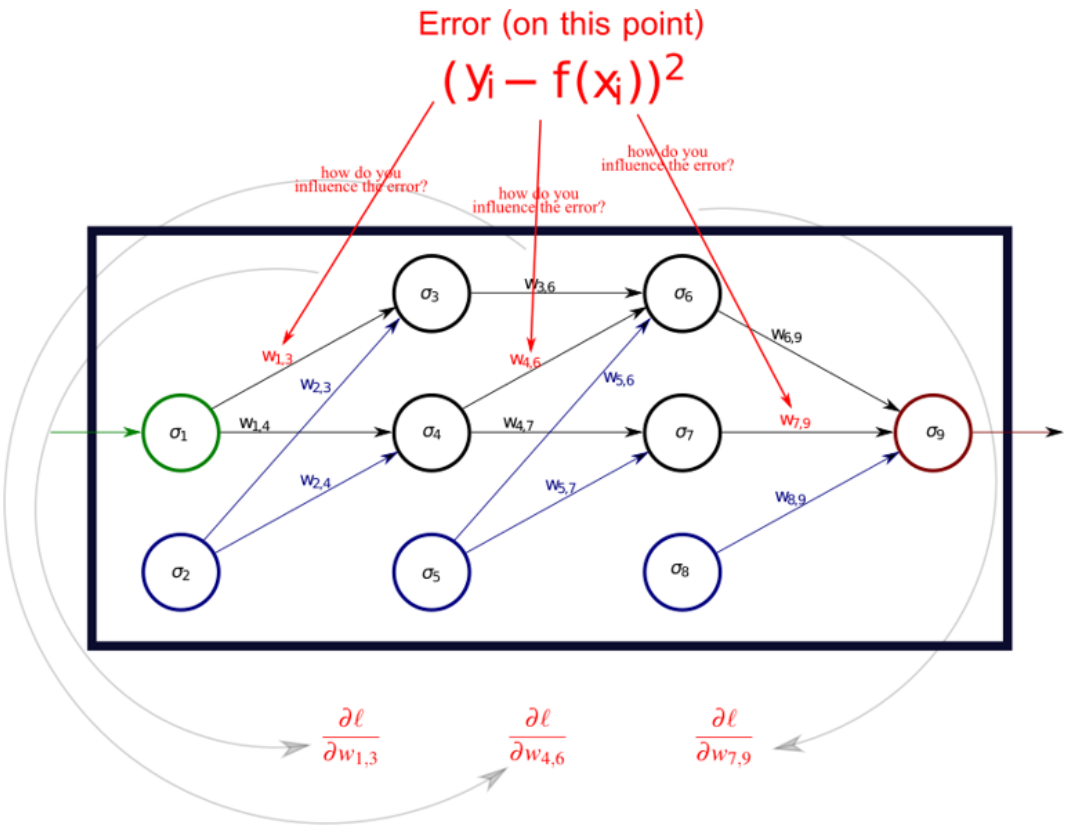
If we feed x_i into the network, we will get some number as the result. Let's call it $f(x_i)$.



This prediction will be good if $f(x_i)$ is close to y_i and bad otherwise, since y_i is the correct output for x_i. Thus, the difference between them is a measure for the quality of the prediction. For unimportant reasons, we square the difference, so we take $(y_i - f(x_i))^2$ as our measure for how our network did (on this one point). If $(y_i - f(x_i))^2$ is small, it did well; if it's large, it did poorly.



This difference depends on all of the weight values $w_{\bullet,\bullet}$, because all of them were used to compute $f(x_i)$. Suppose we arbitrarily pick out one such weight $w_{i,j}$, and pretend that all other weights are *fixed*. Then we can consider the difference $(y_i - f(x_i))^2$ as a function of that weight. We call that function $\ell : \mathbb{R} \to \mathbb{R}$ the *loss function.* To reiterate: the loss function takes a value x for the weight $w_{i,j}$, and outputs the value of $(y_i - f(x_i))^2$ given that we apply the network with value x for weight $w_{i,j}$. (Remember that all other weights are fixed.) And if we can somehow compute this function, we can also take the *derivative*. For the weight $w_{i,j}$, this will be the number $\frac{\partial \ell}{\partial w_{i,j}}$.



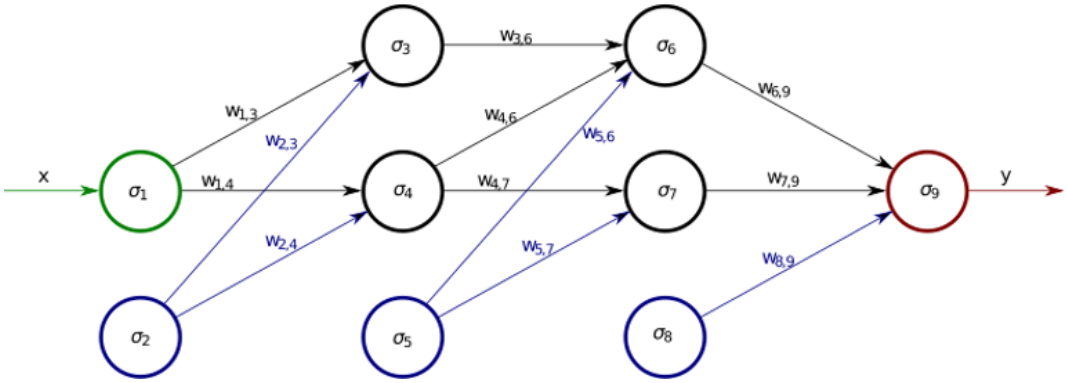
If $\frac{\partial \ell}{\partial w_{i,j}}$ is positive, we know that $(y_i - f(x_i))^2$ will increase if we increase the value for $w_{i,j}$ (and conversely, it will decrease if we decrease the value for $w_{i,j}$). If it is negative, we know that $(y_i - f(x_i))^2$ will decrease if we increase the value for $w_{i,j}$. In both cases, we know how to change $w_{i,j}$ so that the term $(y_i - f(x_i))^2$ will decrease – which is what we want. Now we do this for each weight separately, i.e., we change each weight such that $(y_i - f(x_i))^2$ decreases. The result of this step is a neural network in which all weights are a little bit better than before.

And now – starting with our new weights – we do all of this again with for the next training point. That will again update our weights, and we can repeat the process for a third point, and so on. After updating on all the training points we have, the final weights are the algorithm's output. *How much* we change each weight is determined by a parameter of the algorithm called the *step size*. It usually decreases over the course of the algorithm, i.e., we move the weights more in the beginning when we know they aren't yet any good.

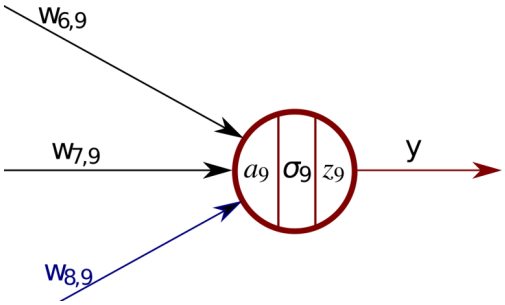
III.II

The remaining question is how to compute the derivative of the loss function with regard to each weight. (From here on, "derivative" will always mean "derivative of the loss function.")

Remember how the network was evaluated layer by layer? Evaluating the derivative with regard to each weight will also be done layer by layer, but this time in reverse. That's why it's called *backpropagation*. Let's begin by computing the weight of an edge leading into the final layer – let's take $w_{6,9}$.



At some point in chapter 1, we've zoomed into the sixth neuron to see that there are actually three distinct elements we need to differentiate: the input to the neuron, i.e., the sum of the weighted values from the incoming edges (a_6), the function of the neuron (σ_6) and the output after applying the function (z_6). Now we need to do the same with the ninth neuron.



Note that z_9 is the same as y. We're interested in the term $\frac{\partial \ell}{\partial w_{6,9}}$. Using the chain rule, we can write

$$\frac{\partial \ell}{\partial w_{6,9}} = \frac{\partial \ell}{\partial z_9} \cdot \frac{\partial z_9}{\partial a_9} \cdot \frac{\partial a_9}{\partial w_{6,9}}$$

The first term is $\frac{\partial \ell}{\partial z_9} = \frac{\partial (y - z_9)^2}{\partial z_9} = -2y + 2z_9$. (This notation is compact but somewhat sloppy – in truth, $\frac{\partial \ell}{\partial z_9}$ is a function which needs to be applied to a parameter.) The second term is the output of the ninth neuron with regard to its input; it depends on the function σ_9. And the third term is easy; we have

$$\frac{\partial a_9}{\partial w_{6,9}} = \frac{\partial [w_{6,9} z_6 + w_{7,9} z_7 + w_{8,9} z_8]}{\partial w_{6,9}} z_6$$

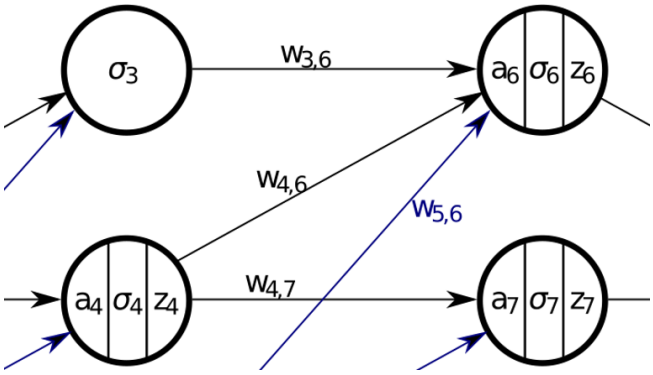
Note that these are all known parameters – we have obtained the $z_\bullet$ by evaluating the neural network on the input x_i.

At this point, we perform a non-obvious change of course. Even though the value $\frac{\partial \ell}{\partial w_{6,9}}$ is what we really care about, to process the *entire network*, it will be more useful to focus on the term $\frac{\partial \ell}{\partial a_9}$. That is, we will focus on the derivative with regard to the input value of the neuron of the final layer, rather than with regard to the weights of the incoming edges. This is easily done – we just drop the third term in the chain rule above, i.e., $\frac{\partial \ell}{\partial a_9} = \frac{\partial \ell}{\partial z_9} \cdot \frac{\partial z_9}{\partial a_9}$. Once we have those derivatives for the entire network (i.e., the derivatives with regard to the input values of every neuron), it will be easy to reconstruct the derivatives with regard to the weights – we just multiply with the third term again.

Thus, the way we will process the entire network is via the following three steps

- compute the derivative with regard to the input neurons of the final layer
- for each layer k (starting from the second last), compute the derivative with regard to the inputs of the neurons of layer k in terms of the derivatives with regard to the inputs of the neurons of layer $k + 1$
- for each weight $w_{i,j}$, compute $\frac{\partial \ell}{\partial w_{i,j}} = \frac{\partial \ell}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{i,j}} = \frac{\partial \ell}{\partial a_j} \cdot z_i$

We've already dealt with step #1, and step #3 is fully described by the equation above. All that's left is to demonstrate how we can do step #2. So suppose we have already processed layers three and four, and now we wish to process layer two. Then we already know the derivatives $\frac{\partial \ell}{\partial a_6}$ and $\frac{\partial \ell}{\partial a_7}$ and $\frac{\partial \ell}{\partial a_8}$ (check with the network above). We now demonstrate how, using this, we can compute the derivative with regard to an input neuron of the second layer. Let's choose a_4 an example; they all work the same way. Once again, we zoom into the relevant part of the network:



Now we have

$$\frac{\partial \ell}{\partial a_4} = \frac{\partial \ell}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_4}$$

The second term depends on σ_4. What about the first? Here is where the last complication comes in. The way z_4 influences the error cannot be reduced to any single derivative in the third layer, because the fourth neuron feeds into both the sixth and the seventh neurons. Thus, we have the apply the advanced chain rule to write

$$\frac{\partial \ell}{\partial z_4} = \frac{\partial \ell}{\partial a_6} \cdot \frac{\partial a_6}{\partial z_4} + \frac{\partial \ell}{\partial a_7} \cdot \frac{\partial a_7}{\partial z_4}$$

Now $\frac{\partial \ell}{\partial a_6}$ and $\frac{\partial \ell}{\partial a_7}$ are terms we have computed in previous rounds, and the other two terms are easy – we have $\frac{\partial a_6}{\partial z_4} = w_{4,6}$ and $\frac{\partial a_7}{\partial z_4} = w_{4,7}$.

III.III

If you have read the previous posts in this sequence, or just know about gradient descent, you might object that the learning process will get stuck in a local minimum because the loss-function is non-convex. In less jargony language: suppose we have gone through a few rounds of the training process with backpropagation. It could be that now our weights are locally optimal – any small change will make the network perform worse – but if we changed them far enough, we could still improve its performance. We don't want the step size to be too large; otherwise, we would keep jumping over the target. Thus, if the network is large, the algorithm will inevitably get stuck in a local minimum.

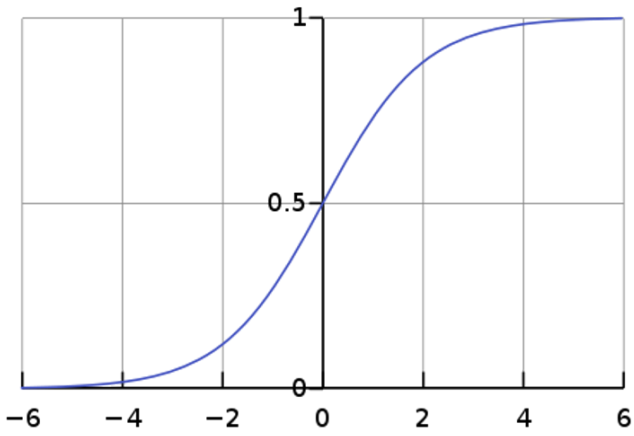
The practical solution is simply to run the entire backpropagation algorithm a bunch of times, say 1000 times, starting from different initial weights. The hope is that, if each run of the algorithm finds some local optimum, and then we take the best of those, the result could still be pretty good. While repeating the entire algorithm 1000 times may seem inefficient, recall that the brute force approach would require upwards of 2^n steps, where n is the number of neurons. For large networks, even running backpropagation a million times would be vastly more efficient than using a brute-force search.

It's worth noting that neural networks are popular because they perform well in practice, not because of theoretical results. Given how important they are, the existing theory is rather underwhelming.

Finally, I've skipped over the functions inside of the neurons all post long. That's because I think they're quite nonessential, and introducing them would complicate the key ideas. But for the sake of completeness: a choice still found in textbooks is the *logistic function* σ_{logistic}, given by the equation

$$\sigma_{\text{logistic}}(x) = \frac{e^x}{e^x + 1}$$

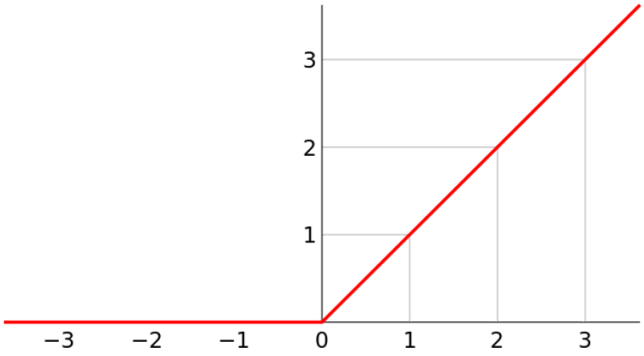
Its graph looks like this (picture from Wikipedia):



So it simply ranges between 0 and 1; it gets close to 0 for very small (negative) inputs, and close to 1 for very large inputs. This particular function has the lovely property that $\sigma_{\text{logistic}}(x) = \sigma_{\text{logistic}}(x) \cdot (1 - \sigma_{\text{logistic}}(x))$, i.e., its derivative is very easy to compute. However, I've been told that this function is not commonly used in practice anymore, and instead, a popular choice is something called the rectifier function f_{rect} defined by

$$f_{\text{rect}}(x) = \max(0, x)$$

Its graph looks like this,



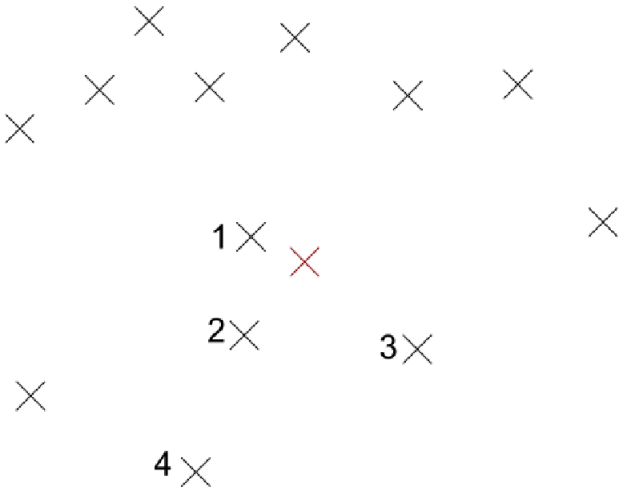
and its derivative is no harder to compute, as it's either 1 or 0.

UML XI: Nearest Neighbor Schemes

(This is the eleventh post in a sequence on Machine Learning based on this book. Click here for part I.)

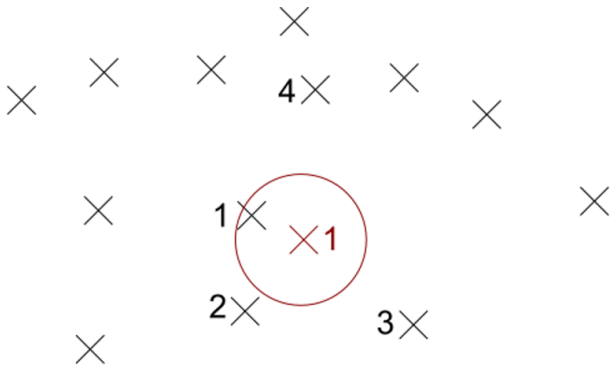
Last time, I tried to do something special because the topic was neural networks. Now we're back to the usual style, but with an unusually easy topic. To fit this theme, it will be a particularly image-heavy post.

The idea of ***nearest neighbor*** predictors is to predict the target value of a point based on the target value of the most similar points in the training data. For example, consider a regression problem with $X = R^2$ and $Y = R$, the following training data, and the new red point:



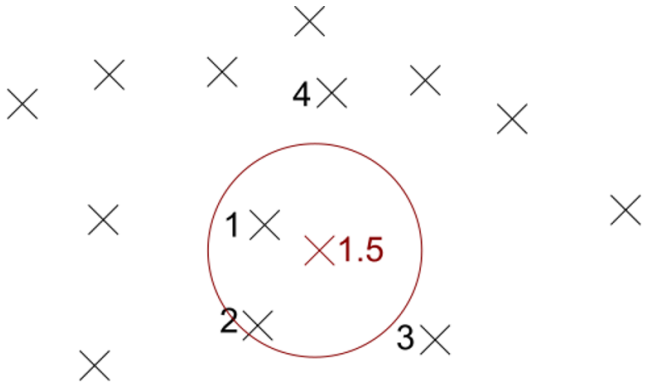
The (x, y) position of the points corresponds to their value in $X = R^2$. The number next to them corresponds to their target value in $Y = R$. (The four closest points just happen to have target values 1-4 by coincidence.) The other training points also have target values, which are not shown – but since they're real numbers, they are probably things like 35.234125763548685697545436399467958568584657456549654649645 34... .

This picture begs the question of how many neighbors to take into account. This is a parameter: in a k-nearest neighbor scheme, we consider the target values of the k nearest instances. In the instance above, if $k = 1$, we have the following situation:

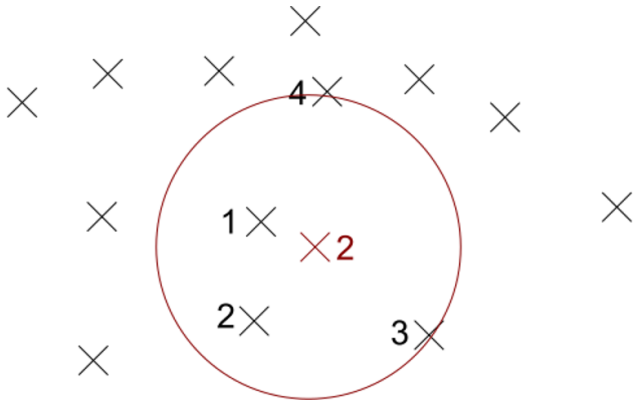


(Note that the circle just demonstrates which points are closest – we do *not* choose all points within a fixed distance.)

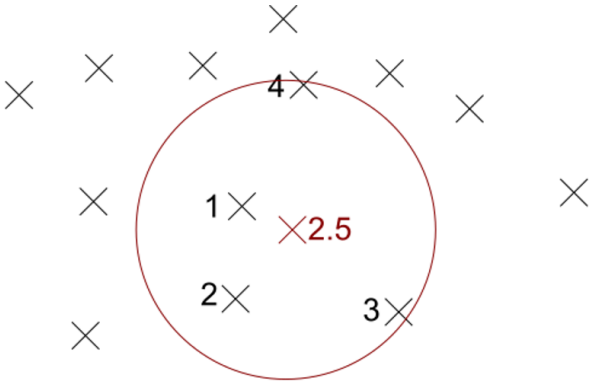
For k = 2, we have two inputs and need to decide what to do with them. For now, we simply declare that the label of our red point will be the mean of all k neighbors.



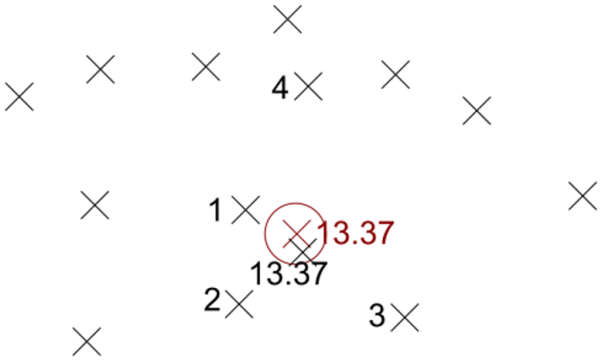
k = 3:



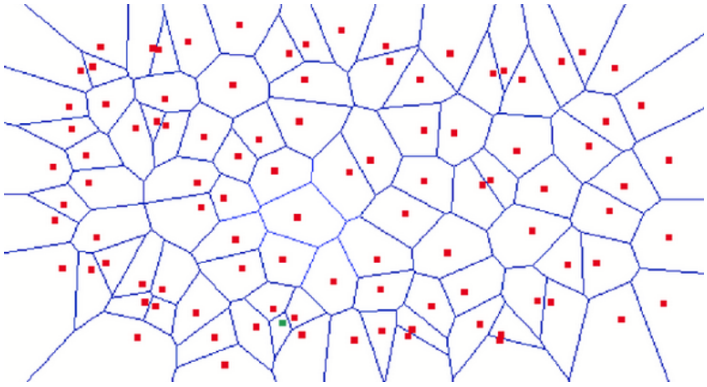
k = 4:



If the function we're trying to learn here is something like distance from a center and the red point is precisely at that center, then the scheme gets worse and worse the larger k becomes. On the other hand, suppose the instance looks like this instead:



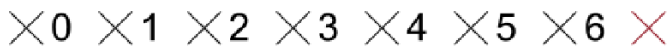
where the new point is an unfortunate outlier. In this case, the situation would be somewhat improved for larger k. In general, k = 1 means that every point will get to decide the target values of some new training points (unless there are other training points with identical positions). Here is an image that shows, for each point, the area in which that point will determine the target value of new points given that k = 1:



This is also called a *Voronoi diagram,* and it has some relevance outside of Machine Learning. For example, suppose the red dots are gas stations. Then, for any point, the [red instance determining the cell in which that point lies] represents the closest gas station.

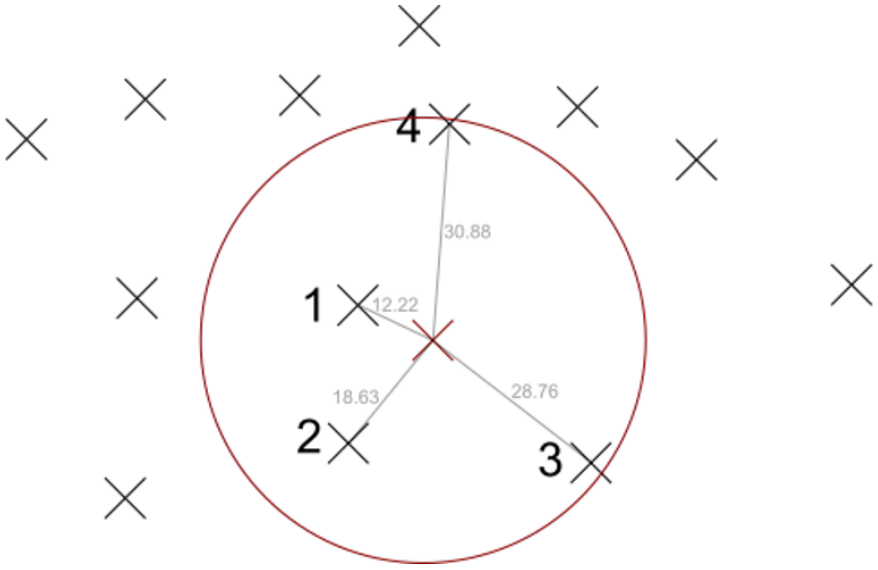
However, for machine learning, this might not be great – just suppose the green point is a crazy outlier. Should it get to decide the target values of new instances in its cell? Another way to describe this problem is that the [function that such a nearest-neighbor predictor will implement] is highly discontinuous.

In general, the more variance there is in the training data, the larger k should be, but making k larger will, in general, decrease accuracy – the well-known tradeoff. In that sense, one is applying a little bit of prior knowledge to the problem by choosing k. But not much – consider the following instance (this time with X = R):



A linear predictor would probably assign the red point a value that continues the trend – something like 7.000000000142. This is because linear models work on the assumption that trends are, in some sense, more likely to continue than to spontaneously revert. Not so with nearest neighbor schemes – if k = 1 it would get the label 6, and for larger k, the label would become smaller rather than larger. In that way, nearest neighbor schemes make weaker assumptions than most other predictors. This fact provides some insight into the question of when they are a good choice for a learning model.

One can also use a *weighted average* as the prediction, rather than the classical mean. Let's return to our instance, and let's use actual distances:



One way to do this is to weight each prediction proportional to the inverse distance of the respective neighbor. In that case, this would lead to the prediction

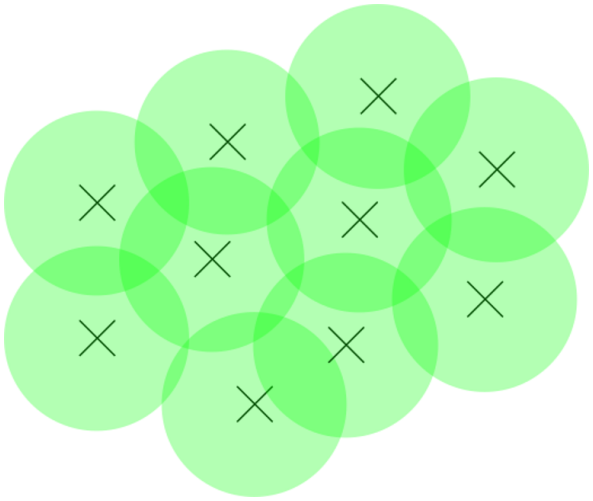
$$\frac{12.22^{-1} \cdot 1 + 18.63^{-1} \cdot 2 + 28.76^{-1} \cdot 3 + 30.88^{-1} \cdot 4}{12.22^{-1} + 18.63^{-1} + 28.76^{-1} + 30.88^{-1}} \approx 2.08736$$

where the denominator is there to normalize the term, i.e., make it as if all weights sum up to 1. (You can imagine dividing each weight by the denominator rather than the entire term; then, the weights literally sum up to 1.) In this case, the function we implement would be somewhat more smooth, although each point would still dominate in some small area. To make it properly smooth, one could take the inverse of [the distance plus 1] (thus making the weights range between 0 and 1 rather than 0 and ∞) and set k = m so that all training points are taken into consideration.

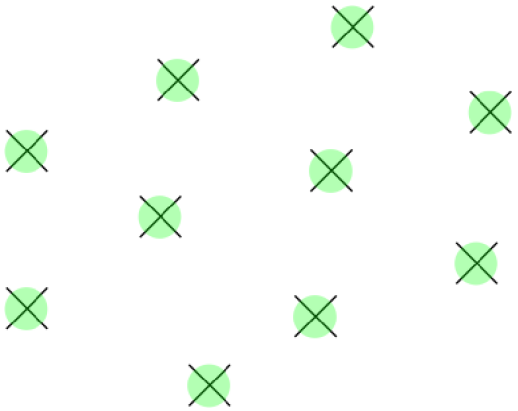
Note that I've been using regression as an example throughout, but decision trees can also be used for classification problems. In that case, each point gets the label which is most popular among its neighbors, as decided by a (weighted) majority vote.

Are there any guarantees we can prove for nearest neighbor schemes?

The question is a bit broad, but as far as sample complexity bounds/error bounds are concerned, the answer is "not without making some assumptions." Continuity of the target function is a necessary condition, but not enough by itself. Consider how much each point tells us about the function we're trying to learn – clearly, it depends on how fast the function changes. Then – suppose our function changes at a pace such that each point provides this amount of information (the green circle denotes the area in which the function has changed "sufficiently little," whatever that means for our case):

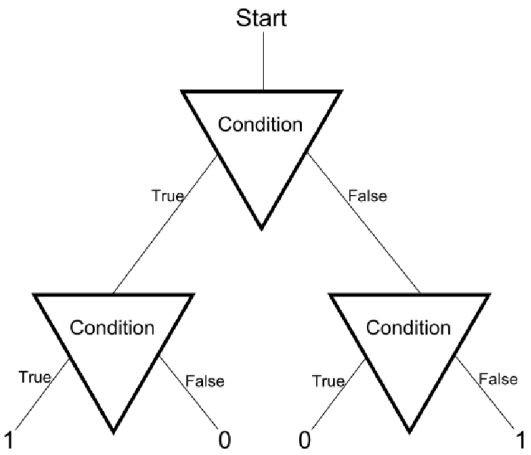


In this case, the training data will let us predict new points. But suppose it changes much more quickly:



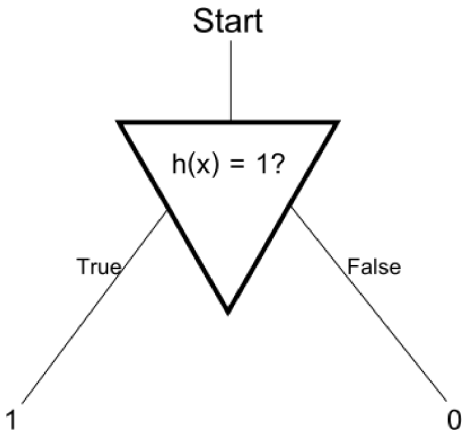
In this case, we have no chance. In general, for any amount of training data, we can imagine a target function that changes so quickly that we don't have a clue about how it looks on a majority of the area. Thus, in order to derive error bounds, one needs to assume a cap on this rate of change. If you recall the chapter on convexity, this property is precisely ρ-Lipschitzness.

There's a theorem to that effect, but the statement is complicated and the proof is boring. Let's talk about trees. A ***decision tree*** is a predictor that looks like this:

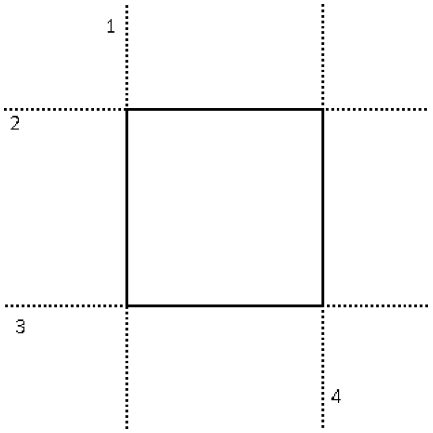


(In this case, for a binary classification problem.) If you have never seen a tree before, don't despair – trees are simple. The triangles are called internal nodes, the four endpoints are called leaves, and the lines are called edges. We begin at "Start," evaluate the first "Condition," then move down accordingly – pretty self-explanatory. Trees show up all over the place in Computer Science.

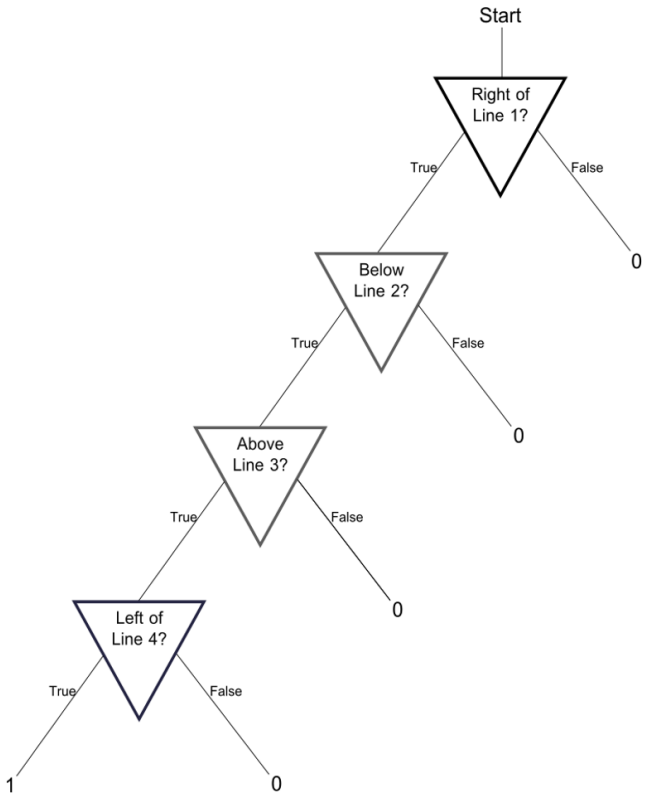
If we allow arbitrary conditions, this is somewhat silly – each predictor h can then be represented as a tree via



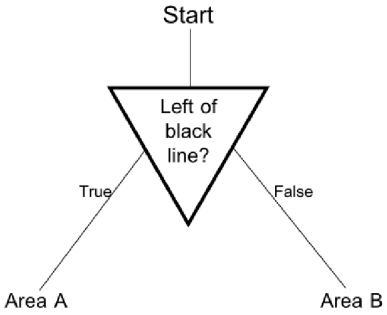
So the general rule is that the conditions be very simple. That being said, there are several ways one could illustrate that this class is quite expressive. For example, consider a rectangle with labeled sides like this:

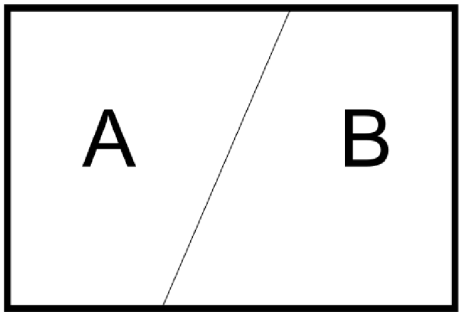


The following tree realizes a predictor which labels instances in the rectangle positively and anywhere outside negatively:



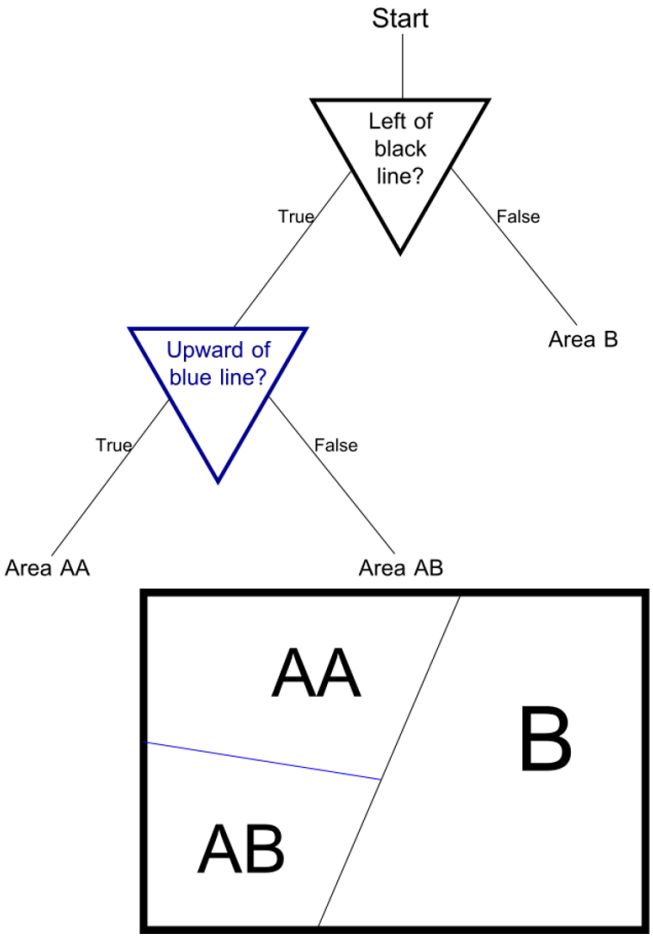
The same principle also illustrates how we can implement a logical AND, and OR goes analogously. You may work out how exactly this can be done – however, I don't think those are the best ways to demonstrate what trees are really about. Instead, consider what happens when the tree branches in two:





When the tree branches in two, the domain space is divided in two. And this is true for each branching point, even if the area corresponding to a node isn't the entire domain

space to begin with:



For the initial examples, I have just written down the labels (the 1 and 0 at the leaves) as part of the tree. In reality, they are derived from the training data: for any leaf, the only reasonable label is that which the majority of training points in the corresponding part of the domain space have. And this is why this post isn't called "nearest neighbor and decision trees" – decision trees *are* a nearest neighbor scheme. The difference lies in how the neighborhoods are constructed: in the classical approach, the neighborhood for each point p is based on the distance of every other point to p, while in a tree, the neighborhoods are the cells that correspond to the leaves.

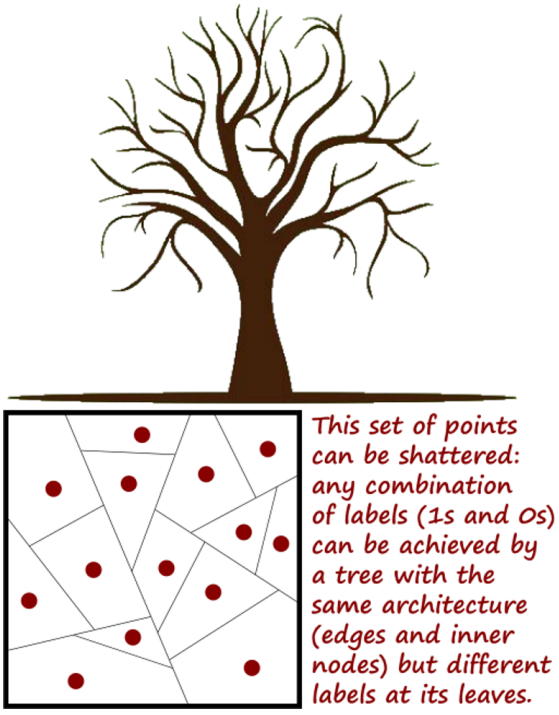
In particular, this makes the neighbor-relationship of a tree transitive – if p is a neighbor of x and x is a neighbor of q, then p is a neighbor of q – which is not true for the classical nearest-neighbor approach.

You might recall the concept of VC-dimension from chapter II, which applies for binary classification tasks. To recap: for a given hypothesis class H, the VC-dimension is the largest $n \in \mathbb{N}$ such that there exists a set of n domain points which is *shattered* by H.

What does it mean for a set $P \subseteq X$ to be shattered by H? It means that H contains a

predictor for any possible labeling combination of the points in P. If P has n elements, these are 2^n many combinations. (For example, if $P = \{x, y\}$, then H has to contain a predictor h_{00} with $h_0(x) = 0 = h_0(y)$, a predictor h_{11} with $h_{11}(x) = 1 = h_{11}(y)$ and also predictors h_{01} and h_{10}.) The VC dimension is a measure for the complexity of a class because if all labeling combinations are possible, then learning about the labels of some of the points doesn't tell us the labels of the others. There are both upper- and lower bounds on the sample complexity for classes with finite VC dimension (provided we allow arbitrary probability distributions D generating our label points).

This is relevant for decision trees because their VC dimension is trivial to compute...



... it simply equals the number of leaves. This follows from the fact that a tree divides the domain space into n subsets where n is the number of leaves, as we've just argued. It's equally easy to see that larger subsets cannot be shattered because the tree will assign all points within the same subset the same label.

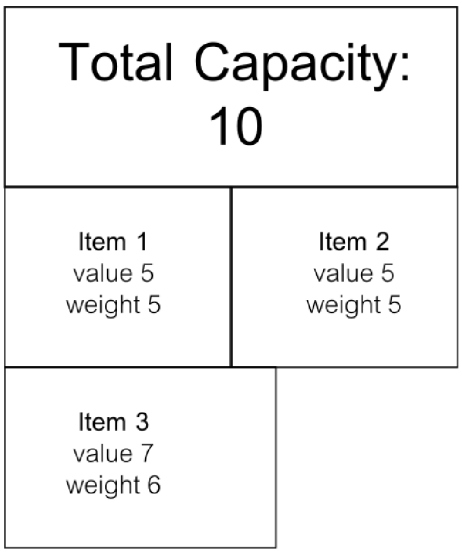
It follows that the class of arbitrary trees has infinite VC-dimension, whereas the class of trees with depth at most d has VC-dimension 2^{d-1} (the number of leaves doubles with every level we're allowed to go downward).

How do we obtain trees?

Since mathematicians are lazy, we don't like to go out and grow trees ourselves. Instead, we'd like to derive an algorithm that does the hard work for us.

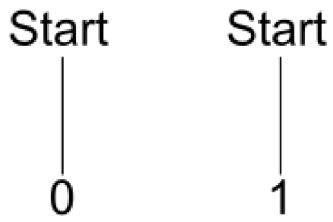
One possibility is a simple greedy algorithm. The term *greedy* is commonly used in computer science and refers to algorithms that make *locally optimal* choices. For example, consider the knapsack problem where one is given a number of possible objects that have a weight and a value, and the goal is to pack a subset of them that stays below a particular total weight and has maximum value. A greedy algorithm would start by computing the $\frac{\text{value}}{\text{weight}}$ scores for each item, and start packing the optimal ones.

The general dynamic with greedy algorithms is that there exist cases where they perform poorly, but they tend to perform well in practice. For the knapsack problem, consider the following instance:



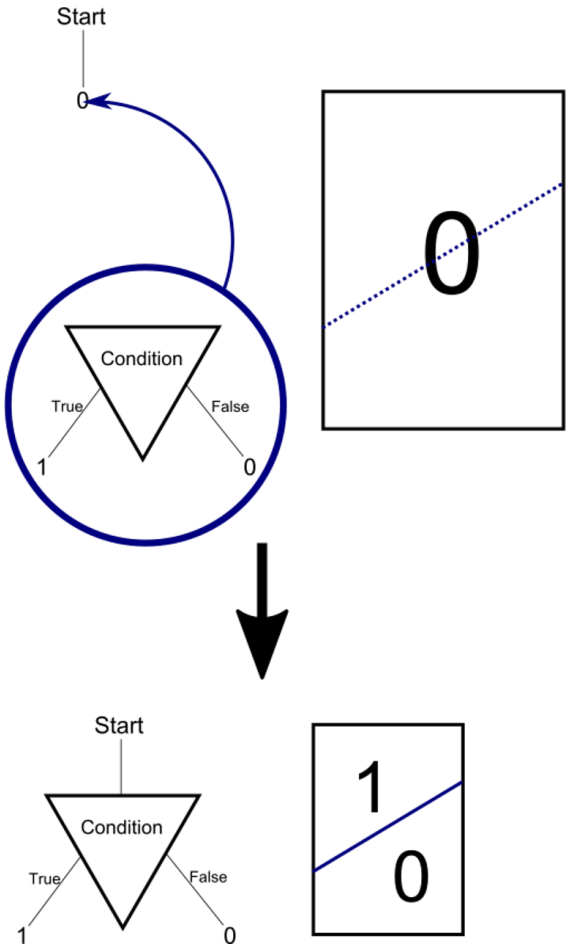
A greedy algorithm would start by packing item 3 because it has the best $\frac{\text{value}}{\text{weight}}$ score, at which point the remaining capacity isn't large enough for another item, and the game is over. Meanwhile, packing items 1 & 2 would have been the optimal solution to this problem.

Now we can do something similar for trees. We begin with one of these two trees,



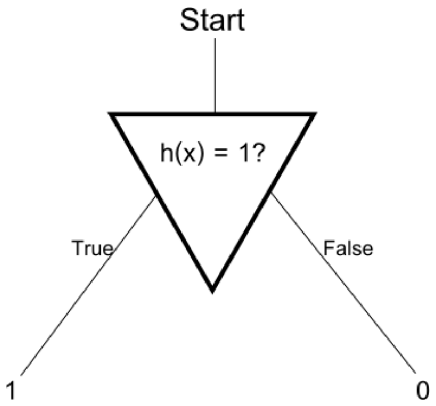
namely, we choose the one that performs better on the training sequence (i.e., if more than half of our training points have label 0, we go with the first tree). After this first step, all remaining steps are the same – we improve the tree by replacing a *leaf* with a [*condition from which two edges go out into two leaves with labels 0 and 1, respectively*]. This corresponds to choosing an area of the domain space and dividing it in two.

So let's say we've started with the first tree. We only have one leaf, so the first step will replace that leaf with [⋯]:



Now just imagine the same thing starting from an arbitrarily large tree: we take one of its many leaves and substitute the object in the blue circle for it (either exactly the same object or the same except with labels swapped). Somewhere in the domain space, some area that was previously all 0 or all 1 now gets divided into two areas, one with 1 and one with 0.

Since we're running a greedy algorithm, it chooses the split such that the total performance of the tree (on the training data) improves the most – without taking into consideration how this affects future improvements. Of course, we need to restrict ourselves to simple conditions; otherwise, the whole thing becomes pointless – recall this tree:



For example, each restriction could be of the form "bit #k of the input point is 1".

Many other approaches are possible. Having chosen such a class of restrictions, and also a way to measure how much performance is improved by each restriction, this defines a simple greedy algorithm.

Now, recall that the VC dimension of the class of trees is infinite. Thus, if we let our algorithm run for too long, it will keep growing and growing the tree until it is very large – at that point, it will have overfit the training data significantly, and its true error will probably be quite high. There are several approaches to remedy this problem:

- Terminate the algorithm earlier
- After the algorithm has run to completion, run an algorithm doing the opposite, i.e., cutting down the tree to make it more simple while losing as little performance on the training data as possible
- Grow more trees

To elaborate on the last point: since the problem with an overly large tree is that it learns noise from the training data – i.e., quirks that are only there by chance and don't represent the real world – one way to combat this is by having a bunch of trees and letting them define a predictor by majority vote. This will lead to the noise canceling out, for the same reason that Stochastic Gradient Descent works. There are, again, at least two ways to do this:

- Use the same tree-growing algorithm, but give it a randomly chosen subset of the training data each time
- Run a modified version of the algorithm several times, and ...
  - ... use the all training data each time; but
  - ... for each step, choose the optimal splitting point from a randomly chosen subset, rather than from all possible splits

The result is called a ***random forest***.

If we have a random forest, are we still implementing a nearest neighbor scheme? To confirm that the answer is yes, let's work a unified notation for all nearest neighbor schemes.

Suppose we have the training data $S = ((x_1, y_1), \ldots, (x_m, y_m))$, and consider a weighting function $w : X \times \{x_1, \ldots, x_m\} \to R$ that says "how much of a neighbor" each training point is to a new domain point. In k-nearest neighbor, we will have that $w(x, x_j) = 1$ iff x_j is one of the k nearest neighbors of x and 0 otherwise. For weighted k -nearest neighbor, if x_j is among the k nearest neighbors to x, then $w(x, x_j)$ will be some (in $(0, 1)$ or in R_+, depending on the weighting) determined by how close it is, and otherwise, it will be 0. For a tree, $w(x, x_j)$ will be 1 iff both x and x_j are in the same cell of the partition which the tree has induced on the domain space.

To write the following down in a clean way, let's pretend that we're in a regression problem, i.e., the y_k are values in R. You can be assured that it also applies to classification, it's just that it requires to mix in an additional function to realize the majority vote.

Given that we are in a regression problem, we have that

$$h(x) = \frac{1}{m} \sum_{i=1}^{m} w(x_i, x) \cdot y_i$$

where $h = A_{\text{classical k-nearest-neighbor}}(S) = A_{kNN}(S)$. Recall that A is a learning algorithm in our notation, and S is the training data, so A(S) is the predictor it outputs.

For a tree, the formula looks the same (but the weighting function is different). And for a forest made out of trees $1, \ldots, n$, we'll have n different weighting functions $w_1, \ldots, w_n$, where w_j is the weighting according to tree #j. Then for $h = A_{\text{random n-forest}}(S)$, we have

$$h(x) = \frac{1}{n} \sum_{j=1}^{n} \frac{1}{m} \sum_{i=1}^{m} w_j(x_i, x) \cdot y_i$$

which can be rewritten as

$$h(x) = \frac{1}{m} \sum_{i=1}^{m} \left[\frac{1}{n} \sum_{j=1}^{n} w_j(x_i, x) \right] \cdot y_i$$

which can, in turn, be rewritten as

$$h(x) = \sum_{i=1}^{m} w^*(x_i, x) \cdot y_i \qquad \text{where} \qquad w^*(x_i, x) := \sum_{j=1}^{n} w_j(x_i, x)$$

so it is just another nearest-neighbor scheme with weighting function w^*.

UML XII: Dimensionality Reduction

(This is the twelfth post in a sequence on Machine Learning based on this book. Click here for part I.)

This post will be more on the mathy side (all of it linear algebra). It contains orthonormal matrices, eigenvalues and eigenvectors, and singular value decomposition.

I

Almost everything I've written about in this series thus far has been about what is called *supervised learning,* which just means "learning based on a fixed amount of training data." This even includes most of the theoretical work from chapters I-III, such as the PAC learning model. On the other side, there is also unsupervised learning, which is learning without any training data.

Dimensionality reduction is about taking data that is represented by vectors in some high-dimensional space and converting them to vectors in a lower-dimensional space, in a way that preserves meaningful properties of the data. One use case is to overcome computational hurdles in supervised learning – many algorithms have runtimes that increase exponentially with the dimension. In that case, we would essentially be doing the opposite of what we've covered in the second half of post VIII, which was about increasing the dimensionality of a data set to make it more expressive. However, dimensionality reduction can also be applied to unsupervised learning – or even to tasks that are outside of machine learning altogether. I once took a class on Information Visualization, and they covered the technique we're going to look at in this post (although "covered" did not include anyone understanding what happens on a mathematical level). It is very difficult to visualize high-dimensional data; a reasonable approach to deal with this problem is to map it into 2d space, or perhaps 3d space, and then see whether there are any interesting patterns.

The aforementioned technique is called ***principal component analysis.*** I like it for two reasons: one, linear algebra plays a crucial role in actually *finding* the solution (rather than just proving some convergence bound); and two, even though the formalization of our problem will look quite natural, the solution turns out to have highly elegant structure. It's an example of advanced concepts naturally falling out of studying a practical problem.

II

Here is the outline of how we'll proceed:

- Define the setting
- Specify a formal optimization problem that captures the objective "project the data into a lower-dimensional space while losing as little information as possible"
- Prove that the solution has important structure
- Prove that it has even more important structure
- Repeat steps #3 and #4

At that point, finding an algorithm that solves the problem will be trivial.

III

Let $n, d \in \mathbb{N}_+$ with $d > n$ and let $(x_1, \ldots, x_m)$ be some data set, where $x_i \in \mathbb{R}^d$ for all $i \in [m]$. We don't assume the data has labels this time since we'll have no use for them. (As mentioned, dimensionality reduction isn't restricted to supervised learning.)

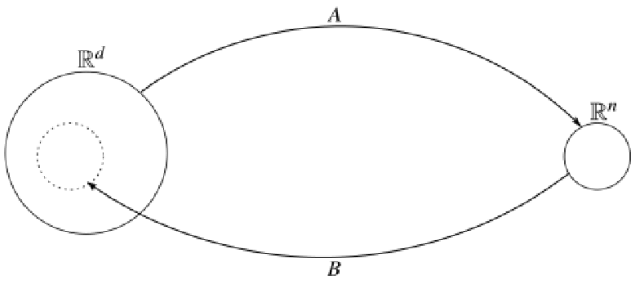
We now wish to find a linear map $\phi : \mathbb{R}^d \to \mathbb{R}^n$ that will project our data points into the smaller space $\mathbb{R}^n$ while somehow not losing much information and another linear map $\psi : \mathbb{R}^n \to \mathbb{R}^d$ that will approximately recover it.

The requirement that ϕ be linear is a pretty natural one. The sets $\mathbb{R}^d$ and $\mathbb{R}^n$ are bijective, so if arbitrary functions are allowed, it is possible to define ϕ in such a way that no information is lost. However, if you've ever seen a bijection between $\mathbb{R}$ and $\mathbb{R}^2$ written out, you'll know that this is a complete non-starter (such functions do not preserve any intuitive notion of structure).

Another way to motivate the requirement is by noting how much is gained by prescribing linearity. The space of all functions between two sets is unfathomably vast, and linearity is a strong and well-understood property. If we assume a linear map, we can make much faster progress.

IV

Since we are assuming linearity, we can represent our linear maps as matrices. Thus, we are looking for a pair (A, B), where A is an $n \times d$ matrix and B a $d \times n$ matrix.



The requirement that the mapping loses as little information as possible can be measured by how well we can recover the original data. Thus, we assume our data points are first mapped to $\mathbb{R}^n$ according to A, then mapped back to $\mathbb{R}^d$ according to B, and we measure how much the points have changed. (Of course, also we square

every distance because mathematicians *love* to square distances.) This leads to the objective

$$\text{argmin}_{A \in \text{Mat}_{n,d}, B \in \text{Mat}_{d,n}} \sum_{i=1}^{m} ||x_i - BAx_i||^2$$

We say that the pair (A, B) is a solution to our problem iff it is in the argmin above.

V

Before we go about establishing the first structural property of our solution, let's talk about matrices and matrix multiplication.

There are (at least) two different views one can take on matrices and matrix multiplication. The common one is what I hereby call the *computational view* because it emphasizes how matrix-vector and matrix-matrix multiplication is realized – namely, like this:

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ 1 & 2 & 3 & 4 \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot & \cdot & 4 & \cdot \\ \cdot & \cdot & 3 & \cdot \\ \cdot & \cdot & 2 & \cdot \\ \cdot & \cdot & 1 & \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 20 & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

$$c_{23} = \sum_{k=1}^{4} a_{2k} b_{k3} = 1 \cdot 4 + 2 \cdot 3 + 3 \cdot 2 + 4 \cdot 1 = 20$$

In this view, matrices and linear maps are sort of the same thing, at least in the sense that both can be applied to vectors.

Since reading Linear Algebra Done Right, I've become increasingly convinced that emphasizing the computational view is a bad idea. The alternative perspective is what I hereby call the *formal view* because it is closer to the formal definition. Under this view, a matrix is a rectangular grid of numbers:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix}$$

This grid (along with the bases of the domain space and target space) contains all the information about how a function works, but it is not itself a function. The crucial part here is that the first column of the matrix tells us *how to write the first basis vector of the domain space as a linear combination of the target space*, i.e., $\phi(b_1) = \sum_{i=1}^{n} a_{i,1} b'_i$ if

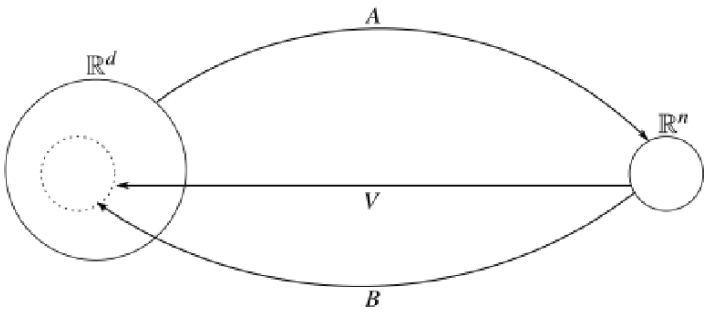
the two bases are $(b_1, \ldots, b_m)$ and $(b_1, \ldots, b_n)$. As far as this post is concerned, we only have two spaces, R^d and R^n, and they both have their respective standard basis. Thus, if we assume n = 2 and d = 3 for the sake of an example, then the d × n matrix

$$\begin{bmatrix} 1 & 4 \\ 2 & 2 \\ 5 & 7 \end{bmatrix}$$

tells us that the vector $(1, 0)$ is mapped onto $(1, 2, 5)$ and the vector $(0, 1)$ onto $(4, 2, 7)$. And that is all; this defines the linear map which corresponds to the matrix.

Another consequence of this view is that I exclusively draw matrices that have rectangular brackets rather than round brackets. But I'm not devoted to the formal view – we won't exclusively use it; instead, we will use whichever one is more useful for any given problem. And I won't *actually* go through the trouble of cleanly differentiating matrices and maps; we'll usually pretend that matrices can send vectors to places themselves.

With that out of the way, the first thing we will prove about our solution is that it takes the form $(\bullet, V)$, where V is a *pseudo-orthonormal* matrix. ("It takes the form" means "there is a pair in the argmin of our minimization problem that has this form.") A square matrix is *orthonormal* if its columns (or equivalently, its rows) are *orthonormal* to each other (if read as vectors). I.e., they all have inner product 0 with each other and have norm 1. However, there is also a variant of this property that can apply to a non-square matrix, and there is no standard terminology for such a matrix, so I call it pseudo-orthonormal. In our case, the matrix V will do this thing:



I.e., it will go from a small space to a larger space. Thus, it looks like this:

$$\begin{bmatrix} & & \\ v_{1,1} & \cdots & v_{1,n} \\ v_{2,1} & \cdots & v_{2,n} \\ & & \\ \vdots & & \vdots \\ v_{d,1} & \cdots & v_{d,n} \end{bmatrix}$$

Or alternatively, this:

$$\begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix}$$

where we write the entries as column vectors, i.e., $v_i = (v_{1,i} \cdots v_{d,i})^T$.

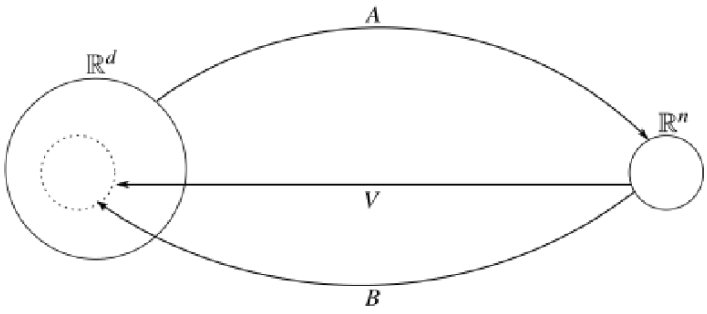
For a proper orthonormal matrix, both the column vectors and the row vectors are orthonormal. For a $d \times n$ pseudo-orthonormal matrix, just the v_i (i.e., the column vectors) are orthonormal. The row vectors can't be (unless many of them are zero vectors) because they're linearly dependent; there are d many, and they live in n-dimensional space.

For a square matrix, the requirement that it be orthogonal is equivalent to the equations $V^T V = I$ and $V V^T = I$, or simply $V^T = V^{-1}$. A $d \times n$ pseudo-orthonormal matrix isn't square and doesn't have an inverse, but the first of the two equations still holds (and is equivalent to the requirement that the matrix be pseudo-orthonormal). To see why, consider the matrix product in the computational view. Instead of flipping the first matrix, you can just imagine both side by side; but go vertical in both directions, i.e.:

$$\begin{bmatrix} v_{1,1} & \cdots & v_{1,n} \\ v_{2,1} & \cdots & v_{2,n} \\ \vdots & & \vdots \\ v_{d,1} & \cdots & v_{d,n} \end{bmatrix} \quad \begin{bmatrix} v_{1,1} & \cdots & v_{1,n} \\ v_{2,1} & \cdots & v_{2,n} \\ \vdots & & \vdots \\ v_{d,1} & \cdots & v_{d,n} \end{bmatrix}$$

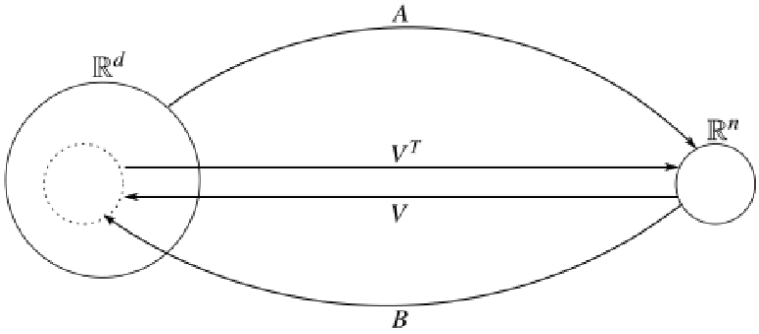
This shows that $(V^T V)_{k,\ell} = \langle v_k, v_\ell \rangle$, and this is 1 iff $k = \ell$ and 0 otherwise because the v_i are orthonormal. Thus, $V^T V = I_n$.

Now recall that we wish to construct a pseudo-orthonormal matrix V and prove that there exists a solution with V, i.e., a solution of the form $(\bullet, V)$.



To construct V note that $(1, 0, \ldots, 0)$ is mapped to the first column vector of any matrix (formal view), and analogously for every other basis vector. It follows that the image of a matrix is just the span of the column vectors. Thus, im(B) is an n-dimensional subspace of R^d, and we can choose an orthonormal basis $(v_1, \ldots, v_n)$ of that subspace. Then we define $V := [\, v_1 \quad \cdots \quad v_n \,]$ as in the example above (only now the v_i are the specific vectors from this orthonormal basis). Note that V is indeed a $d \times n$ matrix. Clearly, both V and B have the same image (formal view).

We will also have use for V^T (quite a lot, in fact). Let's add it into the picture:



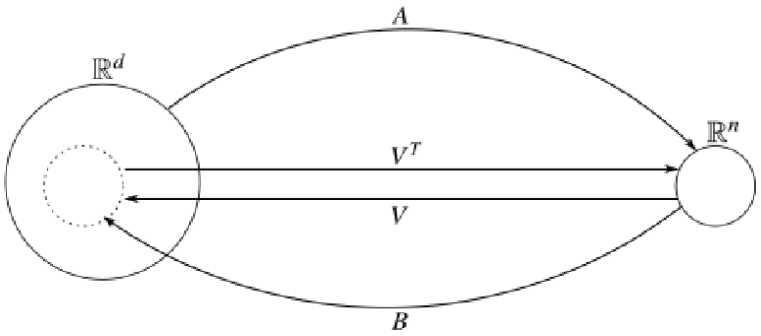
(Note that, even though the arrow for V^T starts from the small circle, it can take vectors from all of R^d.)

We've established that $V^T V$ is the identity matrix, but $V V^T$ isn't. This can also be seen from this picture: $V V^T$ is a map from R^d to R^d, but it maps all vectors into a subspace of R^d. Therefore, all vectors outside of the subspace have to be changed. But what about the vectors who are in the subspace? Could it be that it leaves those unchanged? The answer is affirmative; here's the proof:

Let $y \in \text{im}(V)$. Then $\exists z \in R^n$ such that $Vz = y$, and therefore,

$VV^T y = VV^T(Vz) = V(V^T V)z = Vz = y$.

This fact is going to be important in just a second. Recall that we were here:



and we wanted to prove that there's a solution of the form $(\bullet, V)$. Here's the proof:

Let (A, B) be a solution. Then BA is our optimal back-and-forth mapping. However, $BA = VV^T BA$, and, therefore, the pair (ABV^T, V) is also an optimal back-and-forth mapping and thus is a solution.

Note that $BA = VV^T BA$ holds only because the matrix BA sends all vectors into the subspace $\text{im}(B)$ which is the same as $\text{im}(V)$, and therefore VV^T doesn't do anything.

VI

Next, we prove that there is a solution of the form (V^T, V). We will do this by constructing a term that describes the distance we're trying to minimize and comparing its gradient to zero. Thus, we assume $x \in R^d$ and $y \in R^n$ and want to figure out how to choose y such that the term $||x - Vy||$ is minimal. Using that $||a - b||^2 = \langle a, a\rangle - 2\langle a, b\rangle + \langle b, b\rangle$ and that $\langle a, b\rangle = a^T b = b^T a$, we have:

$$||x - Vy||^2 = x^T x - 2y^T V^T x + y^T V^T V y = ||x||^2 - 2y^T V^T x + ||y||^2$$

The gradient of this term with respect to y is $-2V^T x + 2y$. Setting it to 0 yields $y = V^T x$. Thus, for each point $x \in R^d$, the vector y in R^n that will make the distance

$||x - Vy||$ smallest has the form $V^T x$. In particular, this is true for all of our training points x_i. Since we have shown that there exists a solution of the form $(\bullet, V)$, this implies that (V^T, V) is such a solution.

VII

Before we prove that our solution has even more structure, let's reflect on what it means that it has the form (V^T, V) with V orthonormal. Most importantly, it means that the vectors we obtain are all orthogonal to each other (formal view). We call those vectors the *principal components* of our data set, which give the technique its name. You can think of them as indicating the directions that are most meaningful to capture what is going on in our data set. The more principal components we look at (i.e., the larger of an n we choose), the more information we will preserve. Nonetheless, if a few directions already capture most of the variance in the data, then (V^T, V) might be meaningful even though n is small.

For more on this perspective, I recommend this video (on at least 1.5 speed).

VIII

Based on what we have established, we can write our optimization problem as

$$\text{argmin}_{V \in \text{Mat}_{d,n}, V \text{ orthonormal}} \sum_{i=1}^{m} ||x_i - VV^T x_i||^2$$

Looking at just one of our x_i, we can rewrite the distance term like so:

$$||x_i - VV^T x_i|| = ||x_i||^2 - 2x_i^T VV^T x_i + x_i^T VV^T VV^T x_i = ||x_i||^2 - x_i^T VV^T x_i$$

then, since $||x_i||^2$ does not depend on V, we can further rewrite our problem as

$$\text{argmax}_{V \in \text{Mat}_{d,n}, V \text{ orthonormal}} \sum_{i=1}^{m} x_i^T VV^T x_i$$

Now we perform a magic trick. The term $x_i^T VV^T x_i$ may just be considered a number, but it can also be considered a 1×1 matrix. In that case, it is equal to its trace. (The trace of a square matrix is the sum of its diagonal entries). The trace has the property that one can "rotate" the matrix product it is applied to, so

$$\text{trace}(M_1, \ldots, M_n) = \text{trace}(M_2, \ldots, M_n, M_1)$$

(This follows from the fact that trace(AB) = trace(BA), which is easy to prove.) It also has the property that, if $A^T B$ is square (but A and B are not), $\text{trace}(A^T B) = \text{trace}(AB^T)$; this will be important later. If we apply rotation to our case, we get

$$x_i^T V V^T x_i = \text{trace}(x_i^T V V^T x_i) = \text{trace}(V V^T x_i x_i^T) = \text{trace}(V^T x_i x_i^T V)$$

and, since the trace is linear,

$$\sum_{i=1}^{m} \text{trace}(V^T x_i x_i^T V) = \text{trace}(V^T (\sum_{i=1}^{m} x_i x_i^T) V)$$

The key component here is the matrix $X := (\sum_{i=1}^{m} x_i x_i^T)$. Note that X does not depend on V. However, we will finish deriving our technique by showing that the columns of V are precisely the n eigenvectors of X corresponding to its n largest eigenvalues.

IX

Many matrices have something called a *singular-value decomposition* – this is a decomposition $X = SDS^{-1}$ such that S consists of the eigenvectors of X (column vectors) and D is diagonal, i.e.:

$$S = \begin{bmatrix} w_1 & \cdots & w_d \end{bmatrix} \qquad D = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_d \end{bmatrix}$$

The matrix D can be considered to represent the same function as X but with regard to a different basis, namely the basis of the eigenvectors of X. Thus, the singular value decomposition exists iff there exists a basis that only consists of eigenvectors of X.

It is possible to check that the matrix S which realizes this decomposition is, in fact, the matrix with the eigenvectors of X as column vectors. Consider the first eigenpair

(w_1, λ_1) of X. We have $Xw_1 = \lambda_1 w_1$. On the other hand, S sends the vector $(1, 0, \ldots, 0)$ to w_1 (formal view), and therefore, S^{-1} sends w_1 to $(1, 0, \ldots, 0)$. Then D sends $(1, 0, \ldots, 0)$ to $(\lambda, 0, \ldots, 0)$ because it's a diagonal matrix, and S sends $(\lambda, 0, \ldots, 0)$ to $\lambda S(1, 0, \ldots, 0)$ which is λw_1 (formal view). In summary,

$$S D S^{-1} w_1 = S D(1, 0, \ldots, 0) = S(\lambda, 0, \ldots, 0) = \lambda w_1 = X w_1$$

This shows that SDS^{-1} does the right things on the vectors $w_1, \ldots, w_d$, which is already a proper proof that the decomposition works since $(w_1, \ldots, w_d)$ is, by assumption, a basis of R^d.

In our case, X is symmetric; this implies that there exists a decomposition such that S is orthonormal. (We won't show this.) Note that, unlike V, the matrix S is a square $d \times d$ matrix, hence $S^T = S^{-1}$ and the decomposition can be written as $X = SDS^T$.

Recall that the term we want to maximize is $\text{trace}(V^T X V)$. We can rewrite this as $V^T X V = V^T S D S^T V = [S^T V]^T D [S^T V]$. The matrix $[S^T V]$ is a $d \times n$ matrix, so not square, but it is pseudo-orthonormal since $[S^T V]^T [S^T V] = V^T S S^T V = V^T V = I$. Using the previously mentioned additional rules of the trace (which allow us to take the transpose of both $[S^T V]^T D$ and $[S^T] V$), we can write

$$\text{trace}(V^T X V) = \text{trace}([S^T V]^T D [S^T V]) = \text{trace}(D [S^T V][S^T V]^T)$$

Multiplying any matrix with a diagonal D from the left multiplies every entry in row i with $D_{i,i}$, and in particular, multiplies the diagonal element $([S^T V][S^T V]^T)_{i,i}$ with $D_{i,i}$. Therefore, the above equals $\sum_{i=1}^{d} D_{i,i}([S^T V][S^T V]^T)_{i,i}$. Now recall that $[S^T V]$ is pseudo-orthonormal and looks like this:

$$\begin{bmatrix} z_{1,1} & \cdots & z_{1,n} \\ z_{2,1} & \cdots & z_{2,n} \\ \vdots & & \vdots \\ z_{d_1} & \cdots & z_{d,n} \end{bmatrix}$$

The column vectors are properly orthonormal; they all have norm 1. The row vectors are not orthonormal, but if the matrix were extended to a proper $d \times d$ matrix, they would become orthonormal vectors. Thus, their norm is at most 1. However, the sum of all squared norms of row vectors equals the sum of all squared norms of column vectors (both are just the sum of every matrix entry squared), which is just n. It follows that each $([S^T V][S^T V]^T)_{i,i}$ is at most 1, and the sum, namely $\sum_{i=1}^{d}([S^T V][S^T V]^T)_{i,i}$, is at most n. Now consider the term whose maximum we wish to compute:

$$\sum_{i=1}^{d} D_{i,i}([S^T V][S^T V]^T)_{i,i}$$

Given the above constraints, the way to make it largest is to weigh the n largest $D_{i,i}$ with 1, and the rest with 0. It's not obvious that this can be achieved, but it is clear that it cannot get larger. Thus, $\sum_{i=1}^{d} D_{i,i}([S^T V][S^T V]^T)_{i,i} \leq \sum_{i=1}^{m} D^*_{i,i} = \sum_{i=1}^{m} \lambda^*_i$, where D^* is D reordered such that the largest n eigenvalues, which we call $\lambda^*_1, \ldots, \lambda^*_n$, come first.

Now recall that we have previously written our term as $\text{trace}(V^T(\sum_{i=1}^{m} x_i x_i^T)V)$ or equivalently $\text{trace}(V^T X V)$. If we choose $V = [w^*_1 \cdots w^*_n]$ where $w^*_1, \ldots, w^*_n$ are the eigenvectors of X corresponding to $\lambda^*_1, \ldots, \lambda^*_n$, we get $XV = X[w^*_1 \cdots w^*_n] = [\lambda^*_1 w^*_1 \cdots \lambda^*_n w^*_n]$ and $V^T X V = V^T[\lambda^*_1 w^*_1 \cdots \lambda^*_n w^*_n]$. The i-th entry on the diagonal of this matrix is $w^{*T}_i \lambda^*_i w^*_i = \lambda^*_i ||w^*_i||^2 = \lambda^*_i$. Since we have established

that this is an upper-bound, that makes this choice for V a solution to our original problem, namely $\operatorname{argmin}_{V \in \mathrm{Mat}_{d,n}, V \text{ orthonormal}} \sum_{i=1}^{m} ||x_i - V V^T . x_i||^2$

X

As promised, the algorithm is now trivial – it fits into three lines:

> Given $x_1, \ldots, x_m \in R^d$, compute the matrix $X := \sum_{i=1}^{m} x_i x_i^T$, then compute the eigenvalues $\lambda_1, \ldots, \lambda_d$ of X, with eigenvectors $w_1, \ldots, w_d$. Take the n eigenvectors with the largest eigenvalues and put them into a matrix V. Output (V^T, V).

UML XIII: Online Learning and Clustering

(This is the thirteenth post in a sequence on Machine Learning based on this book. Click here for part I.)

In the previous post, I've mentioned the distinction between supervised and unsupervised learning. Here is a more comprehensive picture:

- ***Supervised learning:*** learning a fixed predictor based on a fixed training sequence generated by a fixed distribution
- ***Unsupervised learning:*** learning without training data
- ***Online learning:*** updating a predictor over time based on data that arrives over time, generated by a process that might change and/or depend on past predictions

In particular, the performance of a predictor in supervised learning is measured after all learning has taken place, whereas, in online learning, the accuracy of our predictor matters at every step along the way.

As an example: while it is possible to model spam detection as a supervised learning problem, it is more *accurate* to model it as online learning, since (a), we may wish to update our spam filter continually, and (b), people composing spam emails may change their behavior based on existing spam filters. Online learning is also the model that comes closest to describing how real humans learn (especially children).

In this post, we'll look at Online Learning as a whole and at clustering, which is a particular problem of unsupervised learning.

Online Learning

For this post, we restrict ourselves to binary classification problems, i.e., $Y = \{0, 1\}$.

Recall that, in supervised learning, we assume there is a fixed probability distribution D over $X \times Y$ according to which the environment generates labeled points. Here, each $x \in X$ is a representation of the real-world thing we wish to label rather than the thing itself. For example, if the task is spam detection of emails, then x would be a vector of some of the email's features (length, address of the sender, number of times the word "money" appears, number of images, ...) rather than an encoding that includes the entire body of text. This means that two different emails may have the same feature representation, and, consequently, the same domain point $x \in X$ might sometimes have label 1 and sometimes label 0.

In some cases, we may wish to assume that labels are deterministic regardless, either because it's a realistic assumption for our particular problem, or for reasons of

simplicity. If we do, we can alternatively model the environment as having a probability distribution D over just X, as well as a *true labeling function* $h_{environment} : X \to Y$. The environment then generates labels by sampling $x \in X$, according to D, and outputting $(x, h_{environment}(x))$.

These last two paragraphs are pure repetition that I've included to precisely highlight the differences between supervised learning and online learning.

In the most general formulation of online learning, we do not assume that the labeled domain points are generated by a probability distribution D. This implies two differences to supervised learning:

- (1): the process by which labels are generated (be it $h_{environment}$ or the conditional probability distribution $D(y \mid x)$) is allowed to change over time; and
- (2): the process by which domain points are generated is not fixed, and might even depend on our predictions for previous labels

Furthermore, I've stated in the introduction that

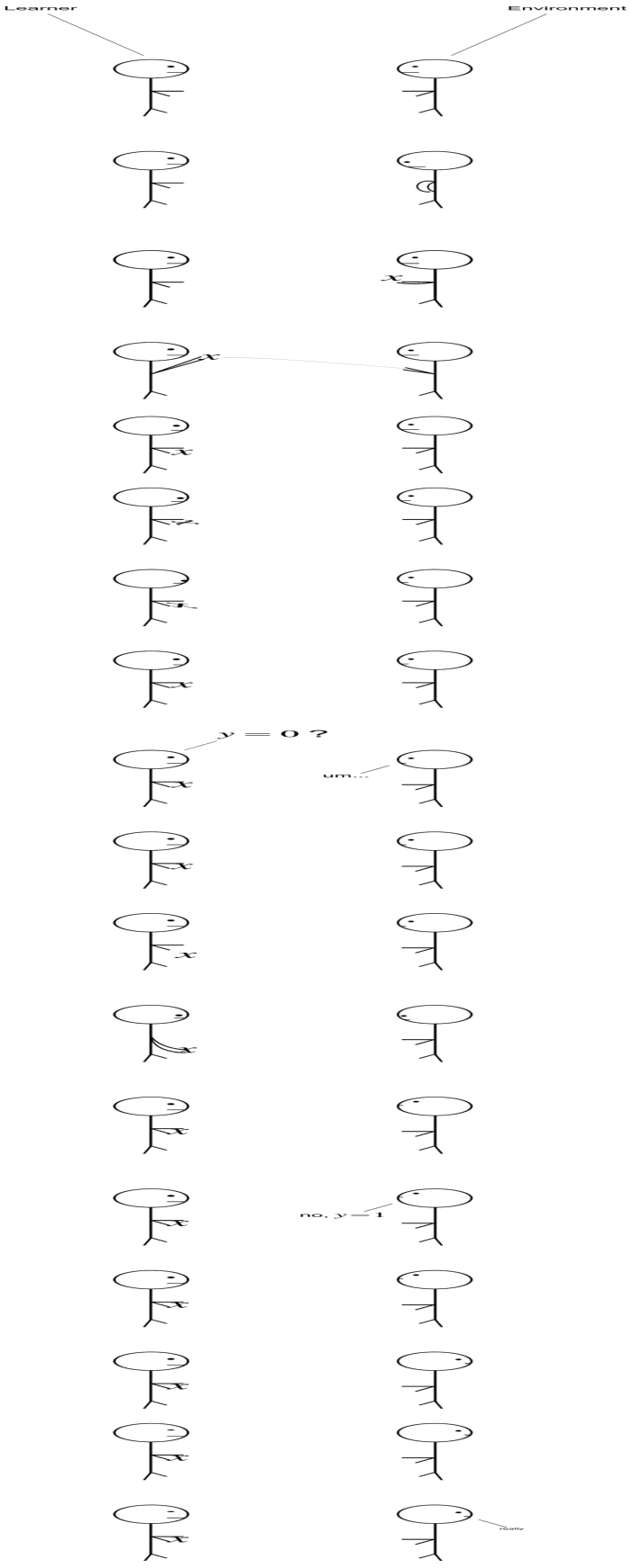
- (3) accuracy of a learning algorithm is measured based on its performance throughout the process

More precisely, we model the learning process to proceed in r*ounds.* In every round, first, the learner A is presented with some instance $x \in X$; then, it predicts a label (either 0 or 1); finally, it is given the true label for x. If the prediction was wrong, we say that A *made a mistake, a*nd the goal is to design algorithms that minimize the number of mistakes. Thus, we don't have a training phase followed by a prediction phase as in supervised learning, but rather a single phase of both training and prediction. We still model this in terms of a sequence $S = ((x_1, y_1), \ldots, (x_m, y_m))$, but we will call it a *data sequence* (term I made up) rather than a training sequence, and we measure how well an algorithm predicts each y_i based on the (x_j, y_j) with $j < i$.

It's not difficult to see that, in this general setting, establishing learning guarantees is impossible. For example, if there is no correlation between the current label and past labels, an algorithm cannot do better than to guess. Worse, if the environment always chooses the label which A didn't guess, A will inevitably make a mistake every round.

This means we have to make further assumptions to simplify the problem. One such assumption is that labels are *consistent with* (but not *generated by*) a predictor in our hypothesis class. That is, after all rounds are complete, there needs to be a predictor $h : X \to Y$ out of the class H such that all labels reported by the environment are consistent with h. However, unlike with supervised learning, we do *not* require that the

environment "chooses" this function ahead of time. Instead, think of the environment as a bullshitter – it may throw you the least useful domain point at every time, and then declare your prediction wrong (whatever it is). As long as, in the end, it can show you a predictor in H that *would have* behaved in precisely this way, this is legal behavior.


Learner
Environment
x
$y = 0$?
um...
no, $y = 1$
really

It is not obvious, at least to me, that this is a particularly useful way to model things – which real-life problem behaves in this way? – but it's what the book does, and I'm too short on time to do independent research, so we'll just roll with it.

To recap the setting in full:

> The learner A has access to the set X of domain points, the label set $Y = \{0, 1\}$, and a hypothesis class $H \subseteq \{f : X \to Y\}$. Given any predictor $h_e \in H$ and any sequence of points $x_1, \ldots, x_m \in X$, the data sequence $S = ((x_1, h_e(x_1)), \ldots, (x_m, h_e(x_m))$ determines a process of m rounds. At round j, the learner A will be given a domain point x_j and be asked to predict a label $y_j \in \{0, 1\}$. For every $j \in [m]$ such that $y_j \neq h_e(x_j)$, A has made a mistake.

And our goal is to design A such that it will make as few mistakes as possible. Formally, A is just an algorithm, so we have a lot of freedom in this construction.

Note that we will measure the performance of A in terms of the highest number of mistakes across all possible data sequences. Thus, even though the formalization above makes it sound like the x_j are fixed ahead of time, it is more accurate to think of them as being chosen by a malicious environment.

In the context of supervised learning, we've written A(S) to refer to the predictor A produces *after* having trained on the training sequence S. In online learning, this definition might still *make sense* (any reasonable algorithm will have its choices consistent with some predictor at every step); however, it's not useful, because the performance of the final predictor isn't what interests us. Instead, given a learner A and a data sequence S, we define $M_A(S)$ as *the number of mistakes* which A made throughout while predicting the labels of S, one by one. Note that $M_A(S) \in \{0, \ldots, |S|\}$.

Given a hypothesis class H, we then define $M_A(H) := \sup\{M_A(S) \mid S \in S\}$, where S is the set of all data sequences that the environment is allowed to come up with (i.e., all data sequences that are consistent with some predictor in H). Note that S is allowed to contain sequences of arbitrary length. Therefore, depending on what A does, it might be that the set $\{M_A(S) \mid S \in S\}$ is unbounded, in which case $M_A(H) = \infty$. With that, we can define:

A hypothesis class H is ***learnable*** iff ∃ an algorithm A such that $M_A(H) \neq \infty$.

This is a good time to think about how to construct A. Assume that the hypothesis class H is finite. How should A behave to make sure the environment can only screw it over for so long?

Here's one possible algorithm, which we call $A_{halving}$. This learner keeps track of a set $V^{(t)} \subseteq H$ of possible hypotheses at every time step t. When presented with a domain point $x_t \in X$, it divides the set in two, depending on which label they give x. I.e.:

$$V_-^{(t)} := \{h \in V^{(t)} : h(x_t) = 0\} \qquad V_+^{(t)} := \{h \in V^{(t)} : h(x_t) = 1\}$$

We have $V^{(t)} = V_-^{(t)} \sqcup V_+^{(t)}$ (disjoint union), which means that at least one of them contains half or more of the predictors in $V^{(t)}$. Our learner $A_{halving}$ chooses the label y_t according to that set. I.e., if $V_+^{(t)} \geq \frac{1}{2}V^{(t)}$ and half or more of the predictors in $V^{(t)}$ say that x has label 1, $A_{halving}$ outputs label 1.

If the environment declares the prediction wrong, $A_{halving}$ halves the remaining hypotheses by setting $V^{(t+1)} := V_-^{(t)}$. That way, it can make at most ld(|H|) mistakes total.

Informally speaking, think of the hypothesis class H as the bullshitter's ammunition. The only way we avoid making mistakes is to reduce this ammunition as fast as possible. $A_{halving}$ reduces it by at least 50% at each step.

Note that any A only gets to make a binary decision at each step. The decomposition $V^{(t)} = V_-^{(t)} \sqcup V_+^{(t)}$ is a natural one, i.e., not specific to $A_{halving}$. After the true label has been announced, the set of remaining candidates will always be one of the two – and each algorithm gets to decide which one.

L-Dimension

Unlike what one might naively suspect, the halving algorithm above is *not* the optimal way to minimize the number of mistakes. This is because it chooses the next subset (i.e., $V_-^{(t)}$ or $V_+^{(t)}$) based on the number of hypotheses in them. However, number-of-hypotheses is a flawed measure for complexity, as you may recall from chapters I-III. In the context of supervised learning, a much better measure is the VC-dimension.

(If you're not familiar with the VC-dimension is, see post II. The one-sentence definition is that the VC-dimension of a hypothesis class H is the largest integer ksuch that there exists a set of k domain points in X that is shattered by H – where a set $P \subseteq X$ is shattered by H iff, for every possible combination of labels of points in P, there exists a predictor $h \in H$ assigning them these labels.)

Our goal now is to work out the analogous concept for this particular brand of online learning. This will be called the ***L-dimension***, also named after a person (in this case, "Littlestone"). The L-dimension of a hypothesis class (written L-dim(H)) will be the largest integer k such that, for every learner A, there exists a data sequence S such that $M_A(S) = k$. This means that $M_A(H) \geq$ L-dim(H) for every algorithm A.

The L-dimension is lower-bounded by the VC-dimension. Informally, given a set P of k many points that is shattered by H, the environment can simply present A with all those points in any order and declare that all of A's predictions were wrong. Since every combination of labels of points in P is realized by some predictor in H, this will be legit regardless of what A does. (Formally, one would have to construct a [data sequence based on the shattered set] on which A fails.)

The reverse is not true – a set of points that is not shattered could still be presented in an inconvenient order such that every learner fails.

Let's take a close look at how both concepts are different. We can characterize the VC-dimension like so:

The VC-dimension of H is at least k $\iff$

$\exists$ a set of k domain points that is shattered by H

By vacuously quantifying over possible orders and labels, we can state the same in a more complicated way thus:

The VC-dimension of H is at least k $\iff$

$\exists$ a set of k domain points such that $\forall$ order on the set

$\forall$ combination of labels

no points' label follows from the previous ones

Similarly (although less cleanly since the formalism is trickier), we can characterize the L-dimension like so:

The L-dimension of H is at least k $\iff$

$\exists f : (X \times Y)^* \to X$ such that for k steps

$\forall$ combination of labels

no points' label follows from the previous ones

where f is the environment's function that chooses the next domain point each round.

This highlights how exactly both concepts differ. The VC-dimension is about a set of points for which knowing the labels of *any* subset of them does not imply the label of the remaining points. The L-dimension is about the ability to pull out new points on the spot (depending on the labels of the previous ones) such that the labels of previous points don't imply the labels of future points.

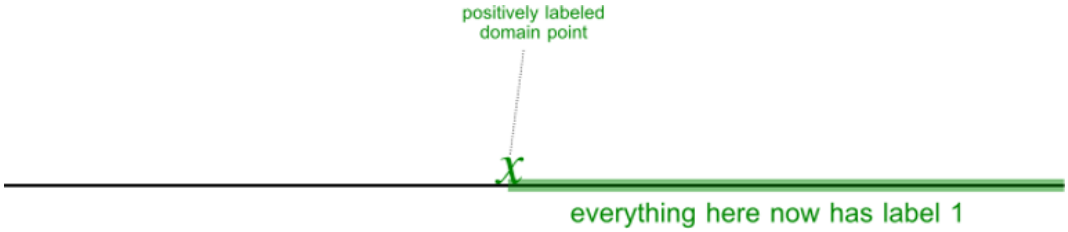
Sometimes, both concepts coincide. Consider the hypothesis class of all predictors that assign label 1 to three domain points and label 0 to all others, i.e.,

$$H_{3\text{-positive}} := \{f : R \to Y \mid |\{x \in R : f(x) = 1\}| = 3\}$$

If A starts by repeatedly guessing 0, it can make at most three mistakes – it's not possible to present domain points in an order such that A doesn't obtain the relevant information. This is because A's knowledge is *evenly distributed* across all domain points it hasn't seen yet; first, it doesn't know the label of any, then it suddenly learns the label of them all. In such cases, nothing is "gained" (or "lost, from A's perspective) from the ability to choose points dynamically.

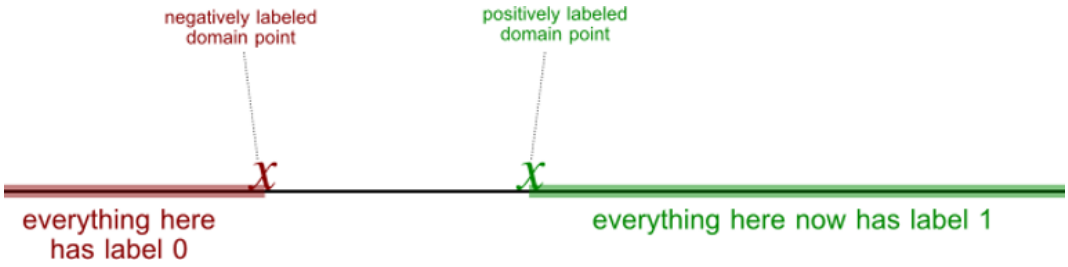
On the other hand, take the class of threshold predictors $H_{\text{threshold}} := \{\theta_x \mid x \in R\}$ where $\theta_x(y) := 1 \iff y \geq x$, and consider what happens if A learns that the label of the

domain point is 1. In that case, it knows the threshold is to the left of x, which means that all points to the right of x also have label 1...



... but it doesn't know the labels of the points to the left of x. In this case, A's knowledge about domain points is *unevenly distributed*. In such a case, the ability to choose new points dynamically matters a lot – if we were trying to construct a shattered set, we would already have hit a wall.

Now, suppose A guesses 1 on a point to the left next, which will then have label 0. Now the picture looks like so:



The area that A can classify safely has increased, but there is still the middle area that is up in the air. This allows the environment to present A with another point it doesn't know yet – say the point right in the middle. Then, if A guesses label 1, the environment will declare that the label is 0. In that case, A knows that all points to the left of x are labeled 0, and the area it doesn't know has been cut in half. Conversely, if A guesses label 0, the environment will declare that the label is 1. In that case, A knows that all points to the right of x are labeled 1, and the area it doesn't know has been cut in half.

In both cases, the [area A cannot safely classify] is cut in half. Since the area is a segment of the real line, it can be cut in half arbitrarily often. This means that, even though VC-dim($H_{threshold}$) = 1, we have L-dim($H_{threshold}$) = ∞.

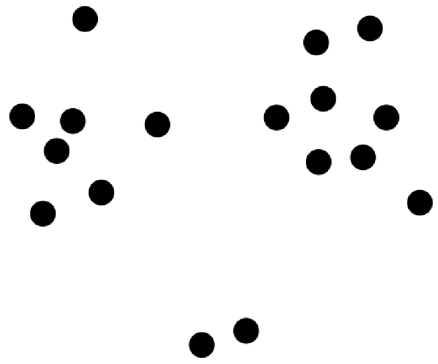
With this bit of theory established, we can design another algorithm $A_{\text{little-L}}$. Whereas A_{halving}, confronted with the choice between $V_+^{(t)}$ and $V_-^{(t)}$, makes its decision based on which set has fewer predictors in it, $A_{\text{little-L}}$ makes its decision based on which class has lower L-dimension.

If $\text{L-dim}(V_-^{(t)}) = \text{L-dim}(V_+^{(t)}) = k$, then $\text{L-dim}(V^{(t)})$ is at least $k + 1$. This is so because otherwise, the environment can let A make a mistake at round t, followed by at least k more in subsequent rounds (recall what the L-dimension represents). Therefore, whenever $A_{\text{little-L}}$ plays, the L-dimension of $V^{(t)}$ decreases by at least 1 every step. This implies that $M_A(H) \leq \text{L-dim}(H)$. Since $M_A(H) \geq \text{L-dim}(H)$ also holds, this implies that $A_{\text{little-L}}$ is the optimal algorithm in this model.

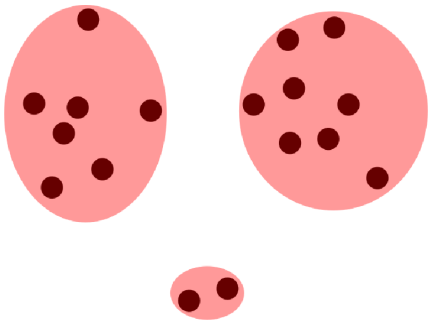
Note that, if H has 2^k elements, then $M_{A_{\text{halving}}}(H) \leq k$. Furthermore, $\text{L-dim}(H) \leq k$ and thus $M_{A_{\text{little-L}}} \leq k$. If the L-dimension is exactly k (this happens if H is simply the set of all possible predictors on a domain set with k elements), then both learners have identical mistake bounds. However, if $\text{L-dim}(H) = d < k$, then $A_{\text{little-L}}$ is guaranteed to make at most d mistakes, while A_{halving} might make up to k.

Clustering

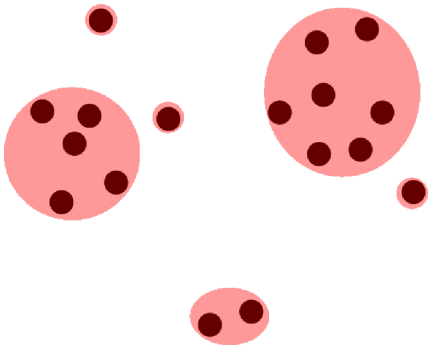
Clustering is about dividing a set of points into disjoint subsets, called ***clusters***, which are meant to group similar elements together. For example, consider the following instance:



Here is a possible clustering:



Here's another:



According to the book, there is no "ground truth" when it comes to clustering, in part because there could be multiple meaningful ways to cluster any given data set. I would dispute that claim – given a finite set of points, there trivially *has* to be an optimum clustering for any precise criterion. However, there is usually no way to evaluate how well this criterion has been met. Suppose you are given some data set, and run a clustering algorithm just to understand it a little better, without even knowing what you are looking for. There will be some clustering that provides you with an optimal amount of insight (otherwise, there wouldn't be any point in having nontrivial algorithms), but it is impossible to tell whether any given clustering is optimal. Furthermore, upon seeing one clustering, you will learn some things about the data, which *changes* the metric of which clustering is most informative next.

Changing metrics aside, learning clustering from training data seems possible in principle. One could input several training sets and their respective optimal clusterings according to human judgment. However, this would require a significantly more complex formalism than what we utilized for supervised learning. For example, the quality of the clustering could no longer be evaluated locally – the same point might "belong" into a different cluster if the training set is extended.

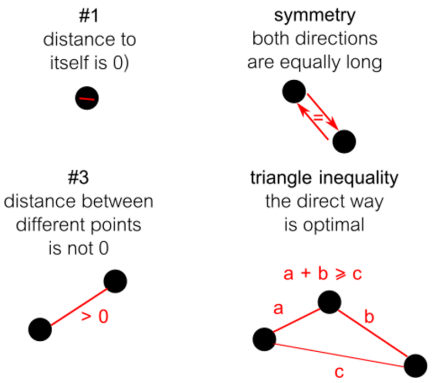
The practical consequence is that one doesn't have a "learner" in the same sense (at least, the book doesn't discuss any such approaches). Instead, all one can do is to define various algorithms that seem to make intuitive sense. This makes our job simpler; all the clustering algorithms we'll look at are quite easy to understand. Consequently, and because there are pretty good resources out there, I'll mostly skip on describing the algorithms in detail and link to external resources instead.

Formalizing the setting

We assume our input is a *finite metric space*. That is, a pair (X, d) where X is any finite set and $d : X \times X \to \mathbb{R}$ a *metric* on the set (it measures distances between points).

Saying that d is a metric is equivalent to saying that it has the following four properties:

- $d(x, x) = 0 \quad \forall x \in X$ (#1)
- $d(x, y) = d(y, x) \quad \forall x, y \in X$ (symmetry)
- $d(x, y) > 0 \quad \forall x, y \in X$ s.t. $x \neq y$ (#3)
- $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality)



Something we do *not* require is the ability to draw a line between points, compute the midpoint of that line, or anything like that. Clustering algorithms work just based on distances. Given (X, d), the *output* of a clustering algorithm is a set $C_1, \ldots, C_k$ such that $X = C_1 \sqcup \cdots \sqcup C_k$. Some clustering algorithms also take k as an input parameter, i.e., they're being told how many clusters they ought to put out.

Impossibility Results

Here's something interesting, before we look at/link to explanations of the actual algorithms. You might have heard the claim that "there is no optimal voting system." This is based on something called *Arrow's impossibiltiy theorem*, and I'm going to explain exactly what it means (this will be relevant for clustering in just a bit). Consider a situation where there are a bunch of candidates and a bunch of parties doing ranked-choice voting. Each party's vote is an ordered list of how much they like all candidates. Now some voting system evaluates these votes and outputs an ordered list of candidates as the result.

Consider the following properties such a system might have:

- **Non-dictatorship:** there is no one party such that the system always outputs exactly the order this party submitted

- **Independence of irrelevant alternatives:** any party changing their order of C and D cannot change the system's order of A and B. Only changes that are about either A or B can do that.
- **Pareto efficiency:** if all votes prefer A over B, so must the system's output

Everyone to whom all three properties seem highly desirable has a problem because Arrow's Theorem states that no system can meet all three properties *unless* there are only two candidates. (If there are only two candidates, a majority vote with a deterministic way to break ties satisfies all three properties). The proof consists of taking a system that has properties #2 and #3 and showing that it is dictatorial (i.e., there is a party such that the system always outputs that party's submission). If one considers #1 and #3 to be essential (and #2 somewhat less so), Arrow's theorem can be summarized as "in every reasonable voting system with more than two candidates, strategic voting must be a thing." These kinds of results are sometimes called *impossibility theorems*: we list a bunch of properties that all seem to make sense and then prove that they're incompatible.

For clustering algorithms, we have something similar, except that it's much less impressive because the properties aren't as obviously desirable. Nonetheless, it's worth bringing up. The properties are

- **Scale-invariance:** multiplying all positions with a constant factor doesn't change the clustering
- **Consistency**: moving a point closer towards all points in a cluster cannot cause it to be dropped from that cluster, and moving it farther away from all points in a cluster cannot cause it to become part of that cluster
- **Richness:** any clustering is possible (for some distance function)

If the distance function measures euclidean distances in 2-dimensional space, then the last property says that, for any possible clustering, you can arrange your points such that the algorithm will return that clustering.

The impossibility theorem states that, while any two of these properties are achievable, no algorithm can achieve all three.

To me, consistency feels like an obvious requirement, but the other two less so. If one shares this view, the impossibility theorem can be summarized as "any reasonable clustering algorithm either depends on scale or is restricted to only a subset of possible clusters."

In this case, the proof is simple. Suppose $X = \{x_1, \ldots, x_n\}$ where $n > 1$. We assume a clustering algorithm A that meets all three properties above and derive a contradiction.

One possible clustering is the trivial clustering, $C_{\text{trivial}} := \{\{x_1\}, \ldots, \{x_n\}\}$. Due to the **richness** property, there must be some metric d^* such that $A(X, d^*) = C_{\text{trivial}}$.

Furthermore, let $C_?$ be an arbitrary clustering other than $C_{trivial}$. Due to richness, there has to, again, be some distance function $d_?$ such that $A(X, d_?) = C_?$.

Now we show that $A(X, d^*) = A(X, d_?)$, which will be our contradiction. We do this by transforming d^* into $d_?$ in such a way that (due to the three properties of A) the cluster cannot change.

First, we change d^* to d_{small} by reducing all distances by the same factor such that the largest distance in d_{small} becomes the smallest distance in d^*. (Then, any distance in d_{small} is at most as large as any distance in $d_?$.) Formally, we set $d_{small}(x, y) := \alpha d^*(x, y) \ \forall x, y \in X$ where

$$\alpha := \frac{\min\{d_?(x, y) \mid x, y \in X\}}{\max\{d^*(x, y) \mid x, y \in X \text{ s.t. } x \neq y\}}$$

due to **scale invariance**, we have $A(X, d^*) = A(X, d_{small})$. Well, and now we increase every distance from d_{small} until it is equal to that of $d_?$. Due to **consistency**, there cannot be any cluster in $A(X, d_?)$ that consists of more points than before because every point either remained at the same distance to that cluster or moved farther away. Thus, since $A(X, d_{small})$ had each point in its own cluster, so does $A(X, d_?)$, which implies that $A(X, d_{small}) = A(X, d_?)$.

Finally, let's take a (brief) look at two families of clustering algorithms.

I

Suppose we have some way of measuring the distance between a point and a cluster. Then, we can do the following:

- Begin with the trivial clustering, i.e., $C^{(0)} := C_{trivial}$
- Merge the cluster/point pair with minimal distance
    - (Note that $|C^{(t+1)}| = |C^{(t)}| - 1$, i.e., we have one fewer cluster.)
- Repeat the previous step until all points are in a single cluster
- Output the entire history; at every time step, we have one possible clustering

Alternatively, we can stop based on some criterion (max distance or max number of clusters).

This is called ***agglomerative clustering***. See here for a decent video explanation.

Note that every definition for the distance between a point p and a cluster C yields its own variant of the algorithm. Possible choices are

- $\min\{d(p,q) \mid q \in C\}$
- $\max\{d(p,q) \mid q \in C\}$
- $\frac{1}{|C|}\sum_{q\in C} d(p,q)$

II

A different approach is called the k-***means algorithm***, which works as follows:

- Choose k points $c_1, \ldots, c_k$ at random called the *centers*
- Let the corresponding clusters be $C_1, \ldots, C_k$
- Assign each point to the cluster whose center it is closest to. I.e., assign $p \in X$ to C_j where $j \in \operatorname{argmin}_{i\in\{1,\ldots,k\}}[d(p, c_i)]$
- Move each center to the center of mass of its cluster. I.e., set $c_j := \frac{1}{|C_j|}\sum_{p\in C_j} c_j$. (If we are not in a vector space and cannot do this, instead choose $c_j \in \operatorname{argmin}_{x\in X}\sum_{p\in C_j} d(x, c_j)$.)
- Repeat until the assignment doesn't move any point to a different cluster

The result depends heavily on the initial randomizing step. To remedy this, one can run the algorithm many times, and then choose the best cluster, where "best" can be measured in a couple of ways, such as by comparing the sums $\sum_{j=1}^{k}(\sum_{p\in C_j} d(c_j, p)^2)$ and choosing the clustering with the smallest sum.

See here for a great video explanation (I recommend 2× speed).

UML final

(This is the fourteenth and final post in a sequence on Machine Learning based on this book. Click here for part I.)

The first part of this post will be devoted to multiclass prediction, which is the last topic I'll cover in the usual style. (It's on the technical side; requires familiarity with a bunch of concepts covered in earlier posts.) Then follows a less detailed treatment of the remaining material and, finally, some musings on the textbook.

Multiclass Prediction

For this chapter, we're back in the realm of *supervised learning.*

Previously in this sequence, we've looked at *binary classification* (where $|Y| = 2$) and *regression* (where $Y = R$). In ***multiclass prediction*** or ***multiclass categorization***, we have $|Y| \in N$ but $|Y| > 2$. For simplicity, we take the next simplest case, i.e., $|Y| = 3$. (Going up from there is straight-forward.) More specifically, consider the problem of recognizing dogs and cats in pictures, which we can model by setting $Y = \{\text{dog}, \text{cat}, \text{noanimal}\}$. For the purposes of this chapter, we ignore the difficult question of how to represent the instance space X.

We take the following approach:

- train a function to compute a score for each label that indicates how well the instance fits that label
- classify each new domain point as the best-fitting label

In our case, this means we are interested in learning a function of the form $f : X \times Y \to R$. Then, if we have

$$f(x, \text{cat}) = 2 \qquad f(x, \text{dog}) = \sqrt{5} \qquad f(x, \text{no\_animal}) = -1$$

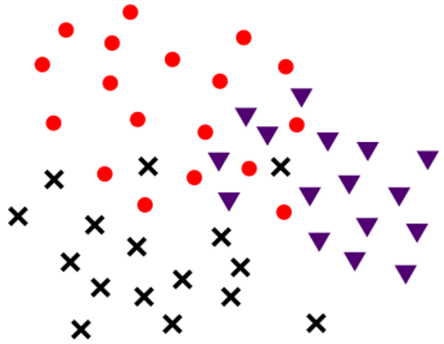
this translates into something like "x kinda looks like a dog, but a little bit more like a cat, and it does seem to be some animal." Each such function determines a predictor $h_f : X \to Y$ via the rule $h_f(x) = \text{argmax}_{y \in Y} f(x, y)$.

We will restrict ourselves to linear predictors. I.e., we assume that the feature space X is represented as something like R^d and our predictors are based on hyperplanes.

With the setting established, the question is how to learn f.

Approach 1: Reduction to Binary Classification

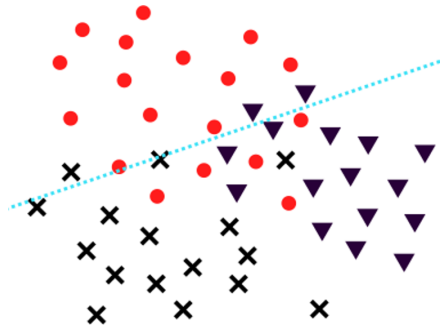
Suppose (unrealistically) that $X = R^2$ and our training data looks like this:



where

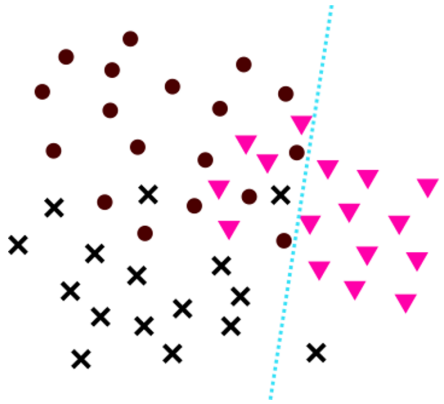
We could now train a separate scoring function for each label. Starting with cats, we would like a function $f_{cat} : X \to R$ that assigns each point a score based on how much it looks like a cat.

For f_{cat}, the label cat corresponds to the label 1, whereas dog and no_animal both correspond to the label 0. In this setting, learning f_{cat} corresponds to finding a passable *hyperplane*. (Apply linear programming for Soft Support Vector Machines.)



Note that, while hyperplanes are used to classify points, they do, in fact, produce a numerical score based on how far on the respective side each point is.

Next, we train a f_{dog} function. Perhaps its hyperplane looks like so:

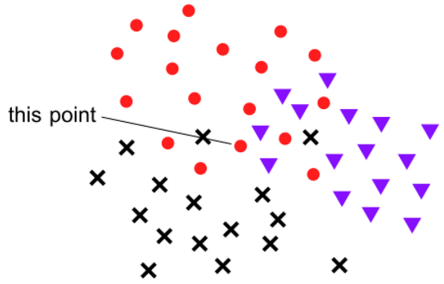


And, finally, a f_{no_animal} function in a similar fashion. Then, we define

$$f(x, cat) := f_{cat}(x) \ \ \forall x \in X$$

And likewise with dog and no_animal. As before, we let $h_f(x) \in \text{argmax}_{y \in Y} f(x, y)$.

Note that a point might be classified correctly, even if it is on the wrong side of the respective hyperplane. For example, consider this point:



f_{cat} thought this point was not a cat, but it thought it less intensely than f_{dog} thought that it was not a dog. Thus, depending on f_{no_animal}'s verdict, the argmax that h computes might still put out a cat label. Of course, the reverse is also true – a point might be classified as a cat by f_{cat} but also as a dog by f_{dog}, and if the dog classification is more confident, the argmax will pick that as the label.

The reduction approach is also possible with predictors that merely put out a binary yes/no. However, numerical scores are preferable, precisely because of cases like the above. If we had used classifiers only, then $h_{cat}(x) = h_{dog}(x) = h_{no\_animal}(x) = 0$ for x = this point and we could do no better than to guess.

While the reduction approach can work, it won't have the best performance in practice, so we look for alternatives.

Approach 2: Linear Programming

Using the fact that we are in the linear setting, we can rephrase our problem as follows: let $\text{cat} = 0,\ \text{dog} = 1,\ \text{no\_animal} = 2$, then we wish to find three *vectors* w_0, w_1, w_2 such that, for all training points (x, i), we have $\langle w_i, x\rangle > \langle w_j, x\rangle$ for all $j \neq i$. In words: for each training point, the inner product with the vector measuring the similarity to the correct label needs to be larger than the inner product with the other vectors (in this case, the 2 others). This can be phrased as a *linear program* in much the same way as for Support Vector Machines (post VIII).

However, it only works if such vectors exist. In practice, this is only going to be the case if the dimension is extremely large.

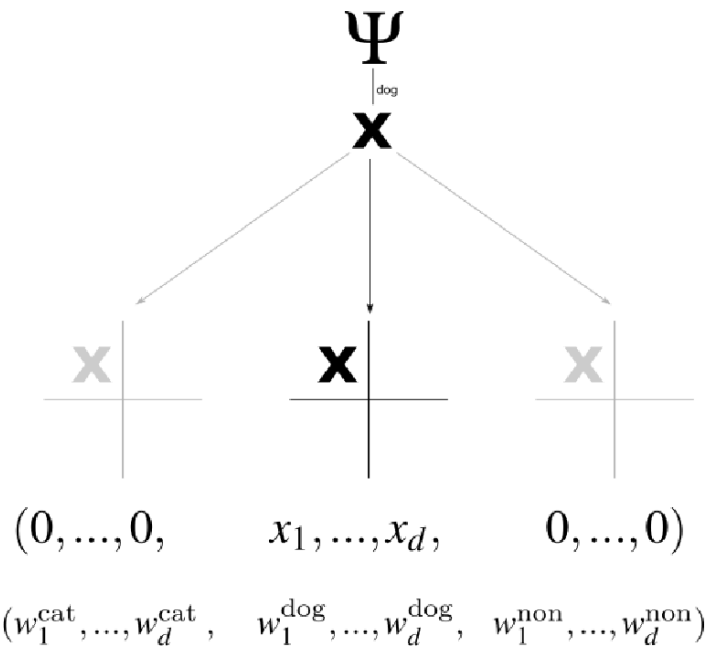
Approach 3: Surrogate Loss and Stochastic Gradient Descent

A general trick I first mentioned in post V is to apply *Stochastic Gradient Descent* (post VI) with respect to a surrogate loss function (i.e., a loss function that is convex and upper-bounds the first loss). In more detail (but still only sketched), this means we

- represent the class of hypotheses H as a familiar set (easy for hyperplanes)
- for each labeled data point (x_i, y_i) in our training sequence S, construct the *point-based loss function* $\ell_{(x_i,y_i)} : H \to R$
- construct convex point-based loss functions $\overset{*}{\ell}_{(x_i,y_i)} : H \to R$ such that $\ell_{(x_i,y_i)} \leq \overset{*}{\ell}_{(x_i,y_i)}$
- compute the gradients $\nabla\overset{*}{\ell}_{(x_i,y_i)}(h)$ and update our hypothesis step by step, i.e., set $h^{(t+1)} := h^{(t)} - \eta\nabla\ell_{(x_t,y_t)}(h^{(t)})$.

To do this, we have to change the framing somewhat. Rather than measuring the loss of our function $f : X \times Y \to R$ (or of the predictor $h_f : X \to Y$), we would like to analyze the loss of a vector $w \in R^n$ for some $n \in N$. However, we do not want to change our high-level approach: we still wish to evaluate all pairs (x, cat) and (x, dog) and $(x, \text{no\_animal})$ separately.

This can be done by the following construction. Let d be the number of coordinates that we use for each label, i.e., $X = R^d$. We can "expand" this space to R^{3d} and consider it to hold three copies of each data point. Formally, we define the function $\Psi : X \times Y \to R^{3d}$ by $\Psi(x, \text{cat}) = x \circ 0 \circ 0$ and $\Psi(x, \text{dog}) = 0 \circ x \circ 0$ and $\Psi(x, \text{no\_animal}) = 0 \circ 0 \circ x$, where 0 is the vector consisting of d many zeros. Then we can represent all parts of our function f as a single vector $w \in R^{3d}$. If we take the inner product $\langle\Psi(x, \text{dog}), w\rangle$, it will only apply the part of w that deals with the dog label since the other parts of $\Psi(x, \text{dog})$ are all 0s.



Thus, what was previously $f(x, \text{cat})$ is now $\langle \Psi(x, \text{cat}), w \rangle$. The nontrivial information that was previously encoded in f is now encoded in w. Our predictor h_w (which is fully determined by w) follows the rule $h_w(x) \in \text{argmax}_{y \in Y} \langle \Psi(x, y), w \rangle$. Thus, we still assign to each point the label for which it received the highest score. Note that w makes use of all 3d coordinates; there's no repetition.

Now we need to define loss functions. Since our goal is to apply Stochastic Gradient Descent, we focus on point-based loss functions $\ell_{(x,y)}$ that measure the performance of w on the point (x, y) only. (We write x fat now since we specified that it's a vector.) One possible choice is the generalized 0-1 loss, $\overset{\text{gen-0-1}}{\ell}_{(x,y)}(w) = 1_{h_w(x) \neq y}$ that is 1 iff w assigns the wrong label and 0 otherwise. However, this choice may not be optimal: if an image contains a cat, we probably prefer the predictor claiming it contains a dog to it claiming it doesn't contain any animal. Therefore, we might wish to define a function $L : Y \times Y \to R$ that measures *how* different two labels are. Then, we set $\overset{L}{\ell}_{(x,y)}(w) := L(h_w(x), y)$. The function L should by *symmetric* and *non-negative*, and it should be the case that $L(y, y) = 0 \ \forall y \in Y$. For our example, a reasonable choice could be

L (cat , cat) = 0	L (cat , dog) = 0.2	L (cat , no_animal) = 1
L (dog , cat) = 0.2	L (dog , dog) = 0	L (dog , no_animal) = 1
L (no_animal , cat) = 1	L (no_animal , dog) = 1	L (no_animal , no_animal) = 0

The loss function $\ell^{L}_{(x,y)}$ as defined above is non-convex: if h_w and $h_{w'}$ output different labels on x, they will have different losses, which means that somewhere on the straight line between w and w', the loss $\ell^{L}_{(x,y)}$ makes a sudden jump (and sudden jumps make a function non-convex).

Thus, our goal now is to construct a convex loss function ℓ^* such that, for each (x, y) in the training sequence, $\ell^{*}_{(x,y)}$ upper-bounds $\ell^{L}_{(x,y)}$.

To this end, consider the term

$$\langle w, \Psi(x, h_w(x))\rangle - \langle w, \Psi(x, y)\rangle$$

for any point (x, y). Recall that the predictor h_w chooses its label precisely in such a way that the above inner product is maximized. It follows that the term above is non-negative. (It's 0 for $y = h_w(x)$). Therefore, adding this term to our loss will upper-bond the original loss. We obtain

$$L(h_w(x), y) + (\langle w, \Psi(x, h_w(x))\rangle - \langle w, \Psi(x, y)\rangle)$$

This term penalizes w for (1) the difference between the predicted and the correct label; and (2) the difference between w's score for the predicted and the correct label. This would be a questionable improvement – we now punish w twice for essentially the same mistake. But the term above is only an intermediate step. Now, instead of using h_w in the term above, we can take the maximum over all possible labels $y' \in Y$; clearly, this will not make the term any smaller. The resulting term will define our loss function. I.e.:

$$\ell^{*}_{(x,y)}(w) := \max_{y' \in Y} (L(y', y) + (\langle w, \Psi(x, y')\rangle - \langle w, \Psi(x, y)\rangle))$$

Note that, unlike before, the difference between the two inner products may be negative.

Let's think about what these two equations mean (referring two the two centered equations, let's call them E1 and E2). E1 (the former) punishes w in terms of the correct label y and the label which w guessed: it looks both at how far its prediction is off and how wrong the label is. E2 does the same for $y' = h_w(x)$, so it will always put out at least as high of an error. However, E2 also looks at all the other labels. To illustrate the effect of this, suppose that the correct label y is dog, but h_w guessed cat. The comparison of the correct label dog to the guessed label cat (corresponding to y' = cat) will yield an error of 0.2 (for the difference between both labels) plus the difference between w's score for both (which is inevitably positive since it

guessed cat). Let's say $\langle w, \Psi(x, \text{cat})\rangle = 0.8$ and $\langle w, \Psi(x, \text{dog})\rangle = 0.5$, then the difference is another 0.3 and the total error comes out 0.5.

Now consider the term for $y' = \text{no\_amimal}$. Suppose $\langle w, \Psi(x, \text{no\_animal})\rangle = 0.3$; in that case, w did think dog was a more likely label than no_animal, but only by 0.2. Therefore, $\langle w, y'\rangle - \langle x, y\rangle = -0.2$. This is a negative term, which means it will be subtracted from the error. However, the loss function "expects" a difference of 1, because that's the difference between the labels dog and no_animal. Thus, the full term $L(y', y) + (\langle w, y'\rangle - \langle x, y\rangle)$ for $y' = \text{no\_animal}$ comes out at $1 - 0.2 = 0.8$, which is higher than for cat.

(Meanwhile, the trivial case of $y' = \text{dog}$ – i.e., comparing the correct label to itself – always yields an error of 0.)

Alas, the loss function ends up scolding w the most not for the (slightly) wrong label it *actually* put out, but for the (very) wrong label it *almost* put out. This might be reasonable behavior, although it certainly requires that the numerical scores are meaningful. In any case, note that the primary point of formulating the point-based loss functions $\overset{*}{\ell}_{(x,y)}$ was to have it be convex, so even if the resulting function were less meaningful, this might be an acceptable price to pay. (To see that it is, in fact, convex, simply note that it is the maximum of a set of linear functions.)

With this loss function constructed, applying Stochastic Gradient Descent is easy. We start with some random weight vector $w^{(0)}$. At step t, we take $w^{(t)}$ and the labeled training point (x_t, y_t). Consider the term

$$\overset{*}{\ell}_{(x_t,y_t)}(w^{(t)}) := \max_{y' \in Y} (L(y', y_t) + (\langle w^{(t)}, \Psi(x_t, y')\rangle - \langle w^{(t)}, \Psi(x_t, y_t)\rangle))$$

The gradient of the maximum of a bunch of functions is the maximum of their gradients. Thus, if y^* is the label for which the above is maximal, then

$$\nabla \overset{*}{\ell}_{(x_t,y_t)}(w^{(t)}) = \Psi(x_t, y^*) - \Psi(x_t, y_t)$$

Therefore, our update rule is $w^{(t+1)} := w^{(t)} - \eta[\Psi(x_t, y^*) - \Psi(x_t, y_t)]$.

Remaining Material

Now we get to the stuff that I read at least partially but decided to cover in less detail.

Naive Bayes Classifier

Naive Bayes is a simple type of classifier that can be used for specific categorization tasks (supervised learning), both binary classification and multi-classification but not regression. The "naive" does not imply that Bayes is naive; instead, it is naive *and* uses Bayes' rule.

We usually frame our goal in terms of wanting to learn a function $h : X \to Y$. However, we can alternatively ask to learn the conditional probability distribution $D((x, y) \mid x)$. If we know this probability – i.e., the probability of observing a (point, label) pair once we see the point – it is easy to predict new points.

For the rest of this section, we'll write "probability" as Pr rather than D, and we'll write $Pr(y \mid x)$ rather than $Pr((x, y) \mid x)$.

Now. For any pair $(x, y) \in X \times Y$, we can express $Pr(y \mid x)$ in terms of $Pr(x \mid y)$ and $P(x)$ and $P(y)$ using Bayes' rule. This implies that knowing all the $Pr(x \mid y)$ is sufficient to estimate any $Pr(y \mid x)$ (the priors on x and y are generally not an issue). The difficulty is that X is usually infinite and we only have access to some finite training sequence S.

For this reason, Naive Bayes will only be applicable to certain types of problems. One of them, which will be our running example, is *document classification* in the *bag of words model*. Here, $X = \{0, 1\}^d$ where d is the number of words in our dictionary, and each document is represented as a vector $x = (x_1, \ldots, x_d) \in X$. where bit x_i indicates whether or not the i-th word of our dictionary appears in [the document represented by x]. Thus, we throw away order and repetitions but keep everything else. Furthermore, Y is a set of possible topics, and the goal is to learn a predictor that sees a new document and assigns it a topic.

The crucial property of this example is that each *coordinate* of our vectors is very simple. On the other hand, the vectors themselves are not simple: we have $d \approx 170.000$ (number of words in the dictionary), which means we have about 2^{170000} parameters to learn (all $P(y \mid x)$). The solution is the "naive" part: we assume that all feature coordinates are independent so that $Pr((x_1, \ldots, x_d) \mid y) = \prod_{i=1}^{d} Pr(x_i \mid y)$. This assumption is, of course, incorrect: the probabilities for x_i = "football" and x_j = "goal" are certainly not independent. But an imperfect model can still make reasonable predictions.

Thus, naive Bayes works by estimating the parameters $Pr(x_i \mid y)$ for any $i \in [d]$ and $y \in Y$. If Y consists of 100 possible topics, then we wish to estimate $100 \cdot d \approx 17.000.000$ parameters, which may be feasible.

Given a labeled document collection (the training sequence S), estimating these parameters is now both simple and fast, which is the major strength of Naive Bayes. In fact, it is the fastest possible predictor in the sense that each training point only has to be checked once. Note that

$\Pr(x_i \mid y)$ is the probability of encountering word x_i in a document about topic y. For example, it could be the probability of encountering the word "Deipnophobia" (fear of dinner parties) in a document about sports. To estimate this probability, we read through all documents about sport in our collection and keep track of how many of them contain this term (spoilers: it's zero). On that basis, we estimate that

$$\Pr(\text{"Deipnophobia"} \mid \text{sport}) = \frac{|\{\text{documents about sport with this word}\}| + 1}{|\{\text{documents about sport}\}| + 1} = \frac{0+1}{132+1}$$

Why the +1? Well, as this example illustrates, the natural estimate will often be 0. However, a 0 as our estimate will mean that any document which contains this word has no chance to be classified as a sports document (see equations below), which seems overkill. Hence we "smooth" the estimate by the +1.

Small tangent: I think it's worth thinking about *why* this problem occurs. Why isn't the computed value our best guess? It's not because there's anything wrong with Bayesian updating – actually the opposite. While the output rule is Bayesian, the parameter estimation, as described above, is not. A Bayesian parameter estimation would have a *prior* on the value of $\Pr(\text{"Deipnophobia"} \mid \text{sport})$, and this prior would be *updated* upon computing the above fraction. In particular, it would never go to 0. But, of course, the simple frequentist guess makes the model vastly more practical.

Now suppose we have all these parameters. Given an unknown document x, we would like to output a document y in the set $\operatorname{argmax}_{y \in Y} \Pr(y \mid x)$. For each y, we can write

$$\Pr(y \mid x) = \Pr(x \mid y) \cdot \frac{\Pr(y)}{\Pr(x)} \propto \Pr(y) \Pr(x \mid y)$$

where we got rid of the term Pr(x) because it won't change the argmax; it doesn't matter how likely it was to encounter our new document, and it will be the same no matter which topic we consider.

Now kicks in the naivety: we will choose a topic in the set

$$\operatorname{argmax}_{y \in Y} \Pr(y) \prod_{\substack{i \in [d] \\ x_i = 1}} \Pr(x_i \mid y)$$

This is all stuff we can compute – we can estimate the prior Pr(y) based on our document collection, and the remaining probabilities are precisely our parameters.

In practice, one uses a slightly more sophisticated way to estimate parameters (taking into account repetitions). One also smoothes slightly differently (smoothing is the +1 step) and maps everything into log space so that the values don't get absurdly small, and we can add probabilities rather than multiplying them. Nonetheless, the important ideas are intact.

Feature Selection

Every learning problem can be divided into two steps, namely

- (1) Feature Selection
- (2) Learning

This is true even in the case of unsupervised learning or online learning: before applying clustering algorithms onto a data set, one needs to choose a representation.

The respective chapter in the book does not describe how to select initial features, only how, given a (potentially large) set T of features, to choose a subset $U \subseteq T$. The approaches are fairly straight-forward:

- Measure the effectiveness of each feature by training a predictor based on that feature only; pick the k best ones (*local scoring)*
- Start with an empty set of features, i.e., $U^{(0)} = \emptyset$. In step t, for each $f \in T - U^{(t)}$, test the performance of a predictor trained on $U^{(t)} \cup \{f\}$. Choose f^* to be the feature maximizing the performance, and set $U^{(t+1)} = U^{(t)} \cup \{f^*\}$ (*greedy selection*)
- Start with the full set of features, i.e., $U^{(0)} = T$. In each step, drop one feature such that the loss in performance is minimal

It's worth pointing out that greedy selection can be highly ineffective since features f_1 and f_2 could be useless on their own but work well together. For an absurdly contrived example, suppose that $y = f_1 + f_2 \bmod p$, i.e., the target value is the sum of both features modulo some prime number p. In that case, if f_1 and f_2 are uniformly distributed, and knowing either one of them is zero information about the target value. Nonetheless, both taken together determine the target value exactly.

(This example is derived from cryptography: splitting a number y into additive parts modulo some fixed prime number is called *secret sharing.* The idea is that all shares are required to reconstruct the secret, but any subset of shares is useless. For example, if $y = f_1 + f_2 + f_3 \bmod p$ and all f_i are uniformly distributed, then y and $f_1 + f_3$ are independently distributed (both uniform), and ditto with $f_1 + f_2$ and $f_2 + f_3$. This remains true for an arbitrary number of shares. In general, Machine Learning and cryptography have an interesting relationship: the former is about learning information, the second about *hiding* information, i.e., making learning impossible. That's why cryptography can be a source of negative examples for Machine Learning algorithms.)

In this particular case, even the greedy algorithm wouldn't end up with features f_1 and f_2 since it has no motivation to pick up either of them to begin with – but the third approach would keep them both. On the other hand, if f_1 is useful on its own but f_2 only in combination with f_1, then the local scoring approach would at best choose f_1, while the greedy algorithm might end up with both. And, of course, it is also possible to construct examples where the third approach fails. Suppose we have 10 features that all heavily depend on each other. Together, they add more than one feature typically does but not ten times as much, so they're not worth keeping. Local scoring and greedy selection won't bother, but the third approach will be such with them: getting rid of one would worsen performance by too much.

Regularization and Stability

This is a difficult chapter that I've read partially but skipped entirely in this sequence.

The idea of ***regularization*** is as follows: suppose we can represent our hypotheses as vectors w_i, i.e., $H = R^d$. Instead of minimizing the empirical loss ℓ_S, we minimize the *regularized empirical loss* $\overset{R}{\ell}_S$ defined as

$$\overset{R}{\ell}_S(w) := \ell_S(w) + R(w)$$

where $R : H \to R$ is a regularization function. A common choice is $R(w) = \lambda||w||^2$ for some parameter $\lambda \in R_+$. In that case, we wish to minimize a term of the form $\ell_S(w) + \lambda||w||^2$. This approach is called ***Regularized Loss Minimization***. There are at least two reasons for doing this. One is that the presence of the regularization function makes the problem easier. The other is that, if the norm of a hypothesis is a measure for its complexity, choosing a predictor with a small norm might be desirable. In fact, you might recall the *Structural Risk Minimization* paradigm, where we defined a sequence $H_1, H_2, \ldots$ of hypothesis classes such that $H = \bigcup_{i=1}^{\infty} H_i$. If we let $H_i := \{w \in H \mid ||w||^2 \leq i\}$, then both Structural Risk Minimization and Regularized Loss Minimization privilege predictors with smaller norm. The difference is merely that the tradeoff is continuous (rather than step-wise) in the case of Regularized Loss Minimization.

The problem

$$\min_{w \in H} [\overset{2}{\ell}_S(w) + \lambda||w||^2]$$

where ℓ^2 is the squared loss can be solved with linear algebra – it comes down to inverting a matrix, where the $\lambda||w||^2$ term ensures that it is always invertible.

Generative Models

Recall the general model for supervised learning: we have the domain set X, the target set Y, and the probability distribution D over $X \times Y$ that generates the labels. Our goal is to learn a predictor $h : X \to Y$.

It took me reaching this chapter to realize that this goal is not entirely obvious. While the function $h : X \to Y$ is generally going to be the *most interesting* thing, it is also conceivable to be interested in the full distribution D. In particular, even if the best predictor h is known (that's the predictor h^* given by $h^*(x) \in \text{argmax}_{y \in Y} D((x, y) \mid x)$), one still doesn't know how

likely each domain point is to appear. Differently put, the probability distribution over X is information separate from the predictor; it's neither necessary nor sufficient for finding one.

The idea of ***generative models*** is to attempt to estimate the distribution directly. A way to motivate this is by Tegmark et. al.'s observation that the process that generates labeled points often has a vastly simpler description than the points themselves (chapter 2.3 of post X). This view suggests that estimating D could actually be *easier* than learning the predictor $h : X \to Y$.

(Although not literally since learning D implies learning the best predictor h.)

The book has Naive Bayes as part of this chapter, even though it only involves estimating the $P(y \mid x)$, not the probability distribution over X.

Advanced Theory

The book is structured in four parts. The first is about the fundamentals, which I covered in posts I-III. The second and third are about learning models, which I largely covered in posts IV-XIV. If there is a meaningful distinction between parts two and three, it's lost on me – they don't seem to be increasing in difficulty, and I found both easier than part one. However, the fourth part, which is titled *advanced theory*, is noticeably harder – too difficult for me to cover it at one post per week.

Here's a list of its chapters:

- ***Rademacher Complexities,*** which is an advanced theory studying the rate at which training sequences become representative (closely related to the concept of uniform convergence from post III).
- ***Covering numbers,*** a related concept, about bounding the complexity of sets
- Proofs for the sample complexity bounds in the quantitative version of the *fundamental theorem of statistical learning* (I've mentioned those a couple of times in this sequence)
- Some theory on the learnability of multiclass prediction problems, including another off-shoot of the VC-dimension.
- ***Compression Bounds,*** which is about establishing yet another criterion for the learnability of classes: if "a learning algorithm can express the output hypothesis using a small subset of the training set, then the error of the hypothesis on the rest of the examples estimates its true error."
- ***PAC-Bayes,*** which is another learning approach

Closing Thoughts on the Book

I've mentioned before that, while Understanding Machine Learning is the best resource on the subject I've seen so far, it feels weaker than various other textbooks I've tried from Miri's list. Part one is pretty good – it actually starts from the beginning, it's actually rigorous, and it's reasonably well explained. Parts two and are mixed. Some chapters are good. Others are quite bad. Occasionally, I felt the familiar sense of annoyance-at-how-anyone-could-possibly-think-this-was-the-best-way-explain-this-concept that I get all the time reading academic papers very rarely with Miri-recommended textbooks. It feels a bit like the authors identified a problem (no good textbooks that offer a comprehensive *theoretical* treatment of the field), worked on a solution, but lost inspiration somewhere on the way and ended up producing several chapters that feel rather uninspired. The one on neural networks is a good example.

(Before bashing the book some more, I should note that the critically acclaimed Elements of Statistical Learning made me angry immediately when I tried to read it – point being, I appear to be unusually hard to please.)

The exercises are lacking throughout. I think the goal of exercises should be to help me internalize the concepts of the respective chapter, and they didn't really do that. It seems that the motivation for them was often "let's offload the proof of this lemma into the exercises to save space" or "here's another interesting thing we didn't get to cover, let's make it an exercise." (Note that there's nothing wrong with offloading the proof of a lemma to exercises *per se*, they *can* be perfectly fine, but they're not *automatically* fine.) Conversely, in Topology, the first exercises of each sub-chapter usually felt like "let's make sure you understood the definitions" and the later ones like "let's make sure you get some real practice with these concepts," which seems much better. Admittedly, it's probably more challenging to design exercises for Machine Learning than it is for a purely mathematical topic. Nonetheless, I think they could be improved dramatically.

Next up (but not anytime soon) are λ-calculus and Category theory.