Machine Learning

or: an Unofficial Guide to the Georgia Institute
of Technology's **CS7641**: *Machine Learning*

George Kudrayvtsev
george.k@gatech.edu

Last Updated: March 3, 2020

Creation of this guide was powered entirely by caffeine in its many forms. ☕ If you found it useful and are generously looking to fuel my stimulant addiction, feel free to shoot me a donation on Venmo @george_k_btw or PayPal kudrayvtsev@sbcglobal.net with whatever this guide was worth to you.

Happy studying! 😇

I'm just a student, so I can't really make any guarantees about the correctness of my content. If you encounter typos; incorrect, misleading, or poorly-worded information; or simply want to contribute a better explanation or extend a section, please raise an issue on my notes' GitHub repository. ☺

PART I

Supervised Learning

Our first minicourse will dive into **supervised learning**, which is a school of machine learning that relies on human input (or "supervision") to train a model. Examples of supervised learning include anything that has to do with labelling, and it occurs far more often than unsupervised learning. It's often reduced down to **function approximation** (think numpy's polyfit, for example): given enough predetermined pairs of (input, output)s, the trained model can eventually predict a never-before-seen input with reasonable accuracy.

An elementary example of supervised learning would be a model that "learns" that the dataset on the right represents $f(x) = x^2$. Of course, there's no guarantee that this data really does represent $f(x) = x^2$. It certainly could just "look a lot like it." Thus, in supervised learning we will need to a basal assumption about the world: that we have some well-behaved, consistent function behind the data we're seeing.

x	$f(x)$
2	4
9	81
4	16
7	49
. . .	

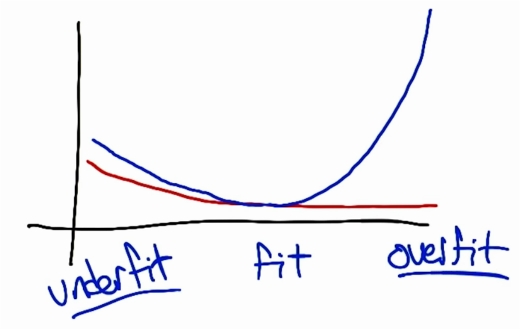
Contents

Techniques

Supervised learning is typically broken up into two main schools of algorithms. **Classification** involves mapping between complex inputs (like image of faces) and labels (like `True` or `False`, though they don't necessarily need to be binary) which we call "classes". This is in contrast with **regression**, in which we map our complex inputs to an arbitrary, often-continuous, often-numeric value (rather than a discrete value that comes from a small set of labels). Classification leans more towards data with discrete values, whereas regression is more-universal, being applicable to any numeric values.

Though data is everything in machine learning, it isn't perfect. Errors can come from a variety of places:

- hardware (sensors, precision)
- human element (mistakes)
- malicious intent (willful misrepresentation)
- unmodeled influences



These hidden errors will factor into the resulting model if they're present in the training data. Similarly, they'll cause inaccuracies in evaluation if they're present in the testing data. We need to be careful about how accurately we "fit" the training data, ideally keeping it general enough to flourish on real-world data. A method for reducing this risk of **overfitting** is called **cross-validation**: we can use some of the training data as a "fake" testing set. The "Goldilocks zone" of training is between **underfitting** and overfitting, where the error across both training data and cross-validation data are relatively similar.

Classification

> *Science is the systematic classification of experience.*
>
> — George Henry Lewes, *Physical Basis of Mind*

Breaking down a classification problem requires a number of important elements:

- **instances**, representing the input data from which the overall model will "learn;"
- the **concept**, which is the abstract concept that the data represents (hopefully representable by a well-formed function);
- a **target concept**, which is the "answer" we want: the ability to classify based on our concept;
- the **hypotheses** are all of the possible functions (ideally, we can restrict ourselves from literally *all* functions) we're willing to entertain that may describe our concept;
- some input **samples** pulled from our instances and paired (by someone who "knows") with the *correct* output;
- some **candidate** which is a potential target concept; and
- a **testing set** from our instances that our candidate concept has not yet seen in order to evaluate how close it is to the ideal target concept.

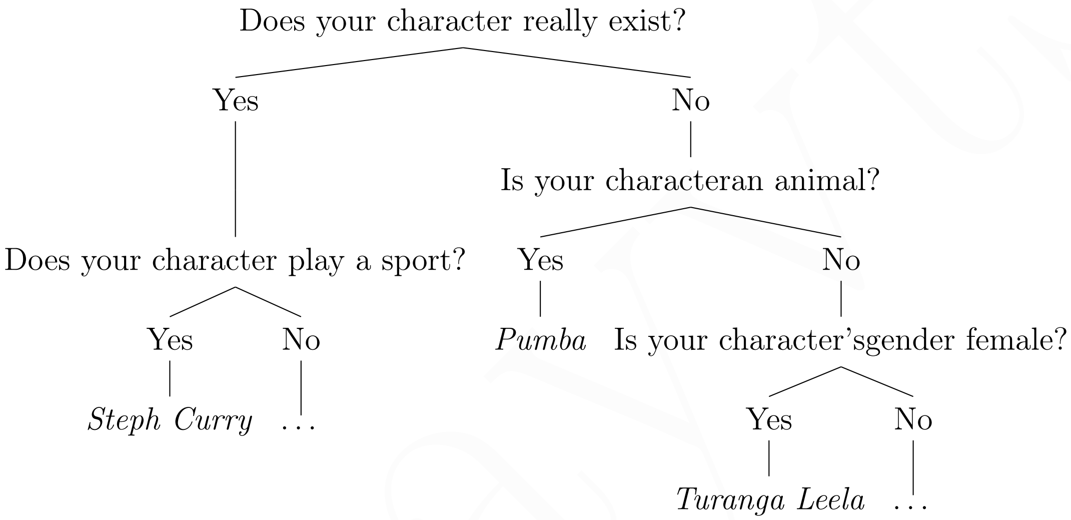
1.1 Decision Trees

Quinlan '86

Decision trees are a form of classification learning. They are exactly what they sound like: trees of decisions, in which branches are particular questions (in which each path down a branch represents a different answer to said question) based on the input, and leaves are final decisions to be made. It maps various choices to diverging paths that end with some decision.

To create a decision tree for a concept, we need to identify pertinent **features** that would describe it well. For example, if we wanted to decide whether or not to eat at a restaurant, we could use the weather, particular cuisine, average cost, atmosphere, or even occupancy level as features.

For a great example of "intelligence" being driven by a decision tree in popular culture, consider the famous "character guessing" AI Akinator. For each yes-or-no question it asks, there are branches the answers that lead down a tree of further questions until it can make a confident guess. One could imagine the following (incredibly oversimplified) tree in Akinator's "brain:"


Does your character really exist?
Yes
No
Is your characteran animal?
Does your character play a sport?
Yes
No
Yes
No
Pumba
Is your character'sgender female?
Steph Curry
. . .
Yes
No
Turanga Leela
. . .

It's important to note that decision trees are a **representation** of our features. Only after we've formed a representation can we start talking about the **algorithm** that will use the tree to make a decision.

The order in which we apply each feature to our decision tree should be correlated with its ability to reduce our space. Just like Akinator divides the space of characters in the world into fiction and non-fiction right off the bat, we should aim to start our decision tree with questions whose answers can sweep away swaths of finer decision-making. For our restaurant example, if we want to spend $\leq \$10$ no matter what, that would eliminate a massive amount of restaurants immediately from the first question.

1.1.1 Getting Answers

The notion of a "best" question is obviously subjective, but we can make an attempt to define it with a little more mathematical rigor. Taking some inspiration from binary search, we could define a question as being good if it divides our data roughly in half. Regardless of our final decision (heh) regarding the definition of "best," the algorithm is roughly the same:

1. Pick the "best" attribute.

2. Ask the question.
3. Follow the answer path.
4. If we haven't hit a leaf, go to Step 1.

Of course the flaw is that we want to *learn* our decision tree based on the data. The above algorithm is for *using* the tree to make decisions. How do we create the tree in the first place? Do we need to search over the (massive) space of all possible decision trees and use some criteria to filter out the best ones?

Given n boolean attributes, there are 2^n possible ways to arrange the attributes, and 2^{2^n} possible answers (since there are 2^n different decisions for each of those arrangements)... we probably want to be a little smarter than that.

1.1.2 Asking Questions: The ID3 Algorithm

ID3 Algorithm, Udacity

If we approach the feature-ranking process greedily, a simple top-down approach emerges:

- $A \leftarrow$ best attribute
- Assign A as the decision attribute (the "question" we're asking) for the particular node n we're working with (initially, this would be the tree's root node).
- For each $v \in A$, create a branch from n.
- Lump the training examples that correspond to the particular attribute value, v, to their respective branch.
- If the examples are perfectly classified with this arrangement (that is, we have one training example per leaf), we can stop.
- Otherwise, repeat this process on each of these branches.

The "information gain" from a particular attribute A can be a good metric for qualifying attributes. Namely, we measure how much the attribute can reduce the overall entropy:

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in A} \frac{|S_v|}{|S|} \cdot \text{Entropy}(S_v) \tag{1.1}$$

where the entropy is calculated based on the probability of seeing values in A:

$$\text{Entropy}(A) = -\sum_{v \in A} \Pr[v] \cdot \log_2 \Pr[v] \tag{1.2}$$

These concepts come from Information Theory which we'll dive into more when we discuss randomized optimization algorithms in chapter 5; for now, we should just think of this as a measure of how much information an attribute gives us about a system. Attributes that give a lot of information are more valuable, and should thus be higher on the decision tree. Then, the "best attribute" is the one that gives us the maximum information gain: $\max\limits_{A \in \mathcal{A}} \text{Gain}(S, A)$.

Inductive Bias

There are two kinds of bias we need to worry about when designing any classifier:

- **restriction bias**, which automatically occurs when we decide our hypothesis set, $\mathcal{H}$. In this case, our bias comes from the fact that we're only considering functions that can be represented with a decision tree.
- **preference bias**, which tells us what sort of hypotheses *from* our hypothesis set, $h \in \mathcal{H}$, we prefer.

The latter of these is at the heart of inductive bias. Which decision trees—out of all of the possible decision trees in the universe that can represent our target concept—will the ID3 algorithm prefer?

Splits Since it's greedily choosing the attributes with the most information gain from the top down, we can confidently say that it will **prefer trees with good splits** at the top.

Correctness Critically, the ID3 algorithm repeats until the labels are correctly classified. And though it may be obvious, it's still important to note that it will hence **prefer correct decision trees** to incorrect ones.

Depth This arises naturally out of the top-heavy split preference, but again, it's still worth noting that ID3 will **prefer trees that are shallower** or "shorter."

1.1.3 Considerations

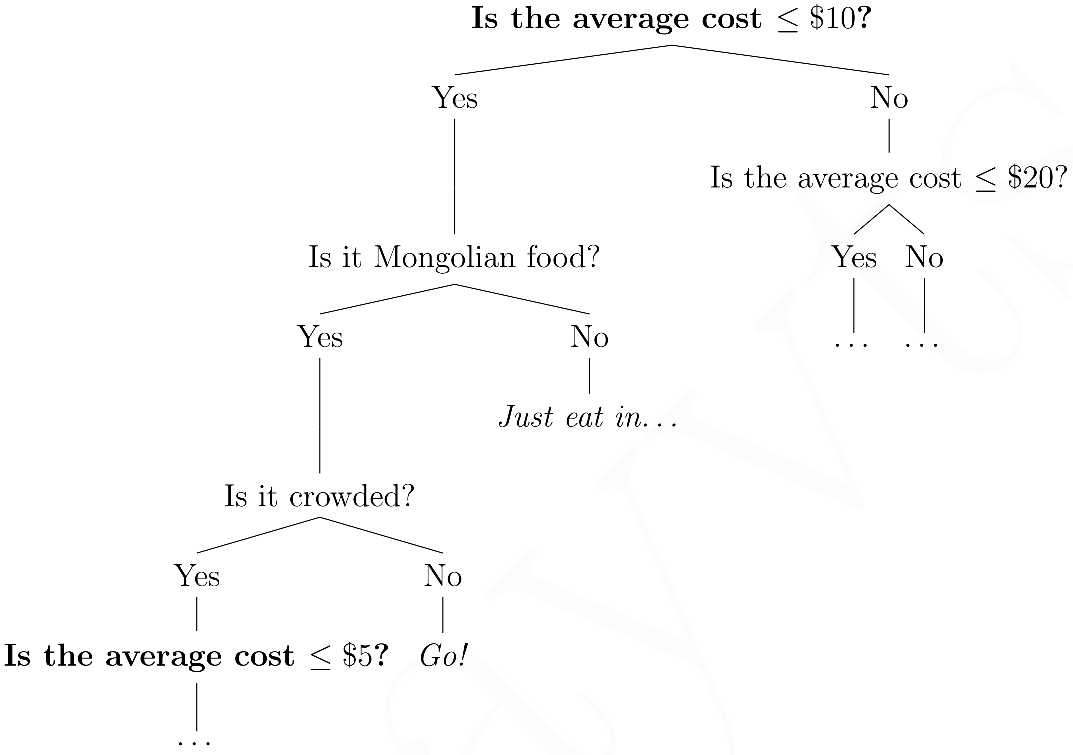
Asking (Continuous) Questions

The careful reader may have noticed the explicit mention of branching on attributes based on *every possible value* of an attribute: $v \in A$. This is infeasible for many features, especially **continuous** ones. For our earlier restaurant example, we may have discretized our "cost" feature into one, two, or three dollar signs (à la Yelp), but what if we wanted to keep them as a raw average dish dollar value instead?

Well, if we're sticking to the "only ask Boolean questions" model, then binning is a viable approach. Instead of making decisions based on a precise cost, we instead make decisions based on a place being "cheap," which we might subjectively define as cost $\in [0, 10)$, for example.

Repeating Attributes

Does it make sense to ask about an attribute more than once down its branch? That is, if we ask about cost somewhere down a path, can (or should) we ask again, later?



With our "proof by example," it's pretty apparent that the answer is "yes, it's acceptable to ask about the same attribute twice." *However*, it really depends on the attribute. For example, we wouldn't want to ask about the weather twice, since the weather will be constant throughout the duration of the decision-making process. With our bucketed continuous values (cost, age, etc.), though, it does make sense to potentially refine our buckets as we go further down a branch.

Stopping Point

The ID3 algorithm tells us to stop creating our decision tree when all of our training examples are classified correctly. That's. . . a lot of leaf nodes. . . It may actually be pretty problematic to refine the decision tree to such a point: when we leave our training set, there may be examples that don't fall into an exact leaf. There may also be examples that have identical features but actually have a different outcome; when we're talking about restaurant choices, opinions may differ:

	Weather	**Cost**	**Cuisine**	**Go?**
Alice:	Cloudy	$	Mexican	✓
Bob:	Cloudy	$	Mexican	×

If both of these rows were in our training set, we'd actually get an infinite loop in the naïve ID3 algorithm: it's impossible to classify every example correctly. It makes sense to adopt a termination approach that is a little more general and robust. We want to avoid overfitting our training examples!

If we bubble up the decisions down the branch of a tree back up to its parent node, then **prune** the branch entirely, we can avoid overfitting. Of course, we'd need to make sure that the generalized decision does not increase our training error by too much. For example, if a cost-based branch had all of its children branches based on weather, and all but one of those resulted in the go-ahead to eat, we could generalize and say that we should always eat for the cost branch without incurring a very large penalty for the one specific "don't eat" case.

Adapting to Regression

Decision trees as we've defined them here don't transfer directly to regression problems. We no longer have a useful notion of information gain, so our approach at attribute sorting falls through. Instead, we can rely on purely statistical methods (like variance and correlation) to determine how important an attribute is. For leaves, too, we can do averages, local linear fit, or a host of other approaches that mathematically generalize with no regard for the "meaning" of the data.

1.2 Ensemble Learning

An **ensemble** is a fancy word for a collective that works together. In this section, we're going to discuss combining learners into groups who will each contribute to the hypothesis individually and essentially corroborate towards the answer.

Ensemble learning is powerful when there are features that may slightly be indicative of a result on their own, but definitely inconclusive, whereas a combination of some of the rules is far more conclusive. In essence, when the whole is greater than the sum of its parts. For example, it's hard to consider an email containing the word "money" as spam, but containing a sketchy URL, the word "money," *and* having mispelled words might be far more indicative.

The general approach to ensemble learning algorithms is to learn rules over smaller subsets of the training data, then combine all of the rules into a collective, smarter decision-maker. A particular rule might apply well to a subset (such as a bunch of spam emails all containing the word "Viagra"), but might not be as prevalent in the whole; hence, each weak learner picks up simple rules that, when combined with the other learners, can make more-complex inferences about the overall dataset.

This approach begs a few critical questions: we need to determine **how to pick subsets** and **how to combine learners**.

1.2.1 Bagging

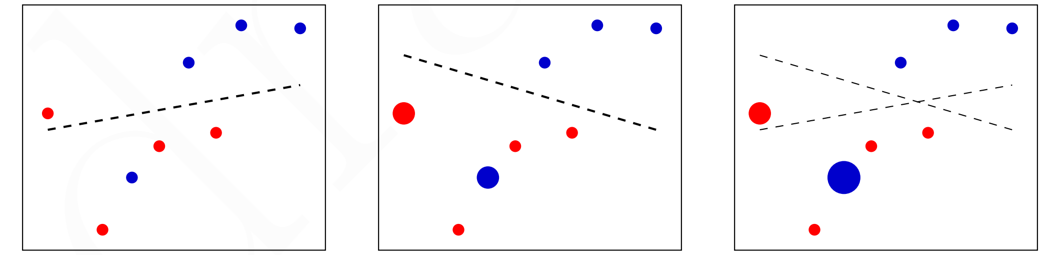
Turns out, simply making choosing data uniformally randomly to form our subset (with replacement) works pretty well. Similarly-simply, combining the results with an average also works well. This technique is called **boostrap aggregation**, or more-commonly **bagging**.

The reason why taking the average of a set of weak learners trained on subsets of the data can outperform a single learner trained on the entire dataset is because of overfitting, our mortal fear in machine learning. Overfitting a subset will not overfit the overall dataset, and the average will "smooth out" the specifics of each individual learner.

1.2.2 Boosting

We must be able to pick subsets of the data a little more cleverly than randomly, right? The basic idea behind **boosting** is to prefer data that we're *not* good at analyzing. We essentially craft learners that are specifically catered towards data that previous learners struggled with in order to form a cohesive picture of the entire dataset.

Initially, all training examples are weighed equally. Then, in each "boosting round," we find the weak learner that achieves the lowest error. Following that, we raise the weights of the training examples that it *mis*classified. In essence, we say, "learn these better next time." Finally, we combine the weak learners from each step into our final learner with a simple weighted average: weight is directly proportional to accuracy.



(a) Our set of training examples and the first boundary guess.

(b) Reweighing the error examples and trying another boundary.

(c) Final classifier is a combination of the weak learners.

Figure 1.1: Iteratively applying weak learners to differentiate between the red and blue classes while boosting mistakes in each round.

Let's define this notion of a learner's "error" a little more rigorously. Previously, when we pulled from the training set with uniform randomness, this was easy to define. The number of mismatches M from our model out of the N-element subset meant an error of $^{M}/_{N}$. However, since we're now weighing certain training examples differently (incorrect $\implies$ likelier to get sampled), our error is likewise different. Shouldn't we punish an incorrect result on a data point that we are intentionally trying to learn more-so than an incorrect result that a dozen other learners got correct?

Example 1.1: Understanding Check: Training Error

Suppose our training subset is just 4 values, and our weak learner $H(x)$ got two of them correct:

x_1	x_2	x_3	x_4
×	✓	×	✓

What's our training error? Trivially $^1/_2$, you might say. Sure, but what if the probability of each x_i being chosen for this subset was different? Suppose

	x_1	x_2	x_3	x_4
	×	✓	×	✓
$\mathbb{D}$:	$^1/_2$	$^1/_{20}$	$^2/_5$	$^1/_{20}$

Now what's our error? Well getting x_1 wrong is a *way* bigger deal now, isn't it? It barely matters that we got x_2 and x_4 correct... So we need to weigh each incorrect answer accordingly:

$$\varepsilon = \underbrace{\frac{1}{2} + \frac{2}{5}}_{\text{incorrects}} = 1 - \underbrace{\frac{1}{20} - \frac{1}{20}}_{\text{corrects}} = \boxed{\frac{9}{10}}$$

To drive the point in the above example home, we'll now formally define our error as the probability of a learner H not getting a data point $\mathbf{x}_i$ correct *over the distribution* of $\mathbf{x}_i$s. That is,

$$\varepsilon_i = \Pr_{\mathbb{D}}\left[H(\mathbf{x}_i) \neq y_i\right]$$

Our notion of a **weak learner**—a term we've been using so far to refer to a learner that does well on a subset of the training data—can now likewise be formally defined: it's a learner that performs better than chance for *any* distribution of data (where ε is a number very close to zero):

$$\forall \mathbb{D} : \Pr_{\mathbb{D}}\left[\cdot\right] \leq {}^1/_2 - \varepsilon$$

Note the implication here: if there is *any* distribution for which a set of hypotheses can't do better than random chance, there's no way to create a weak learner from those hypotheses. That makes this a pretty strong condition, actually, since you need a lot of good hypotheses to cover the various distributions.

Boosting at a high level can be broken down into a simple loop: on iteration t,

- construct a distribution $\mathbb{D}_t$, and
- find a weak classifier $H_t(x)$ that minimizes the error over it.

Then after the loop, combine the weak classifiers into a stronger one.

Specific boosting algorithms vary in how they perform these steps, but a well-known one is the adaptive boosting (or **AdaBoost**) algorithm outlined in algorithm 1.1. Let's dig into its guts.

AdaBoost

From a very high level, this algorithm follows a very human approach for learning:

The better we're doing overall, the more we should focus on individual mistakes.

Freund & Schapire, '99

We return to the world of classification, where our training set maps from a feature vector to a "correct" or "incorrect" label, $y_i \in \{-1, 1\}$:

$$\mathcal{X} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$$

Determining Distribution We start with a uniform distribution, so $\mathbb{D}_1(i) = {}^1\!/_n$. Then, the probability of a sample in the *next* distribution is weighed by its "correctness":

$$\mathbb{D}_{t+1}(i) = \frac{\mathbb{D}_t(i)}{z_t} \cdot \exp(-\alpha_t y_i H_t(\mathbf{x}_i)) \qquad \text{where } \alpha_t = \frac{1}{2} \ln \frac{1 - \varepsilon_t}{\varepsilon_t}$$

Uh, *scrrrrrr. . .* let's break this down piece by piece.

The leading fraction is the previous probability scaled by a normalization factor z_t that is necessary to keep $\mathbb{D}_{t+1}$ a proper probability distribution, so we can basically ignore it.[1]

Notice the clever trick here given our values for y_i: when the classification is *correct*, $y_i = H_t(\mathbf{x}_i)$ and their product is 1; when it's *incorrect*, their product is always -1.

Because $\alpha > 0$ (shown later in the detailed math aside), the $-\alpha$ will always flip the sign of the "correctness" result. Thus, we're doing $e^{-\alpha}$ when the learner agrees and e^{α} otherwise. A negative exponential is a fraction, so we should expect the whole term to make $\mathbb{D}_t(i)$ decrease when we get it right and increase when we get it wrong:

$$\mathbb{D}_{t+1}(i) \implies \begin{cases} \uparrow & \text{if } H(\cdot) \text{ is incorrect} \\ \downarrow & \text{otherwise} \end{cases}$$

On each iteration, our probability distribution adjusts to make $\mathbb{D}$ favor incorrect answers so that our classifier H can learn them better on the next round; it weighs incorrect results more and more as the overall model performance increases.

[1] We don't dive into this in the main text for brevity, but here z_t would be the sum of the *pre*-normalized weights. In an algorithm (like in algorithm 1.1), you might first calculate a $\mathbb{D}'_{t+1}$ that didn't divide any terms by z_t, then calculate $z = \sum_{d \in \mathbb{D}} d$ and do $\mathbb{D}_{t+1}(i) = {}^{\mathbb{D}'_{t+1}(i)}\!/_z$ at the end.

Finding the Weak Classifier Notice that we kind-of glossed over determining $H_t(x)$ for any given round of boosting. Weak learners encompass a large class of learners; a simple decision tree could be a weak learner. All it has to do is guarantee performance that is slightly better than random chance.

Deng, '07

Final Hypothesis As you can see in algorithm 1.1, the final classifier is just a weighted average of the individual weak learners, where the weight of a learner is its respective α. And remember, α_t is in terms of ε_t, so it measures how well the t^{th} round went overall; thus, a good round is weighed more than a bad round.

The beauty of ensemble learning is that you can combine many simple weak classifiers that individually hardly do better than chance together into a final classifier that performs *really* well.

Quick Maffs: Boosting, Beyond Intuition

Let's take a deeper look at the definition of $\mathbb{D}_{t+1}(i)$ to understand how the probability of the i^{th} sample changes. Recall that in our equation, the exponential simplifies to $e^{\mp\alpha}$ depending on whether $H(\cdot)$ guesses y_i correctly or incorrectly, respectively.

$$\mathbb{D}_{t+1}(i) = \frac{\mathbb{D}_t(i)}{z_t} \cdot \underbrace{\exp(-\alpha_t y_i H_t(\mathbf{x}_i))}_{e^{\mp\alpha}} \qquad \text{where } \alpha_t = \frac{1}{2}\ln\frac{1-\varepsilon_t}{\varepsilon_t}$$

Let's look at what e^{α} simplifies to:

$$\begin{aligned} e^{\alpha} &= \exp\left(\frac{1}{2}\cdot\ln\frac{1-\varepsilon}{\varepsilon}\right) \\ &= \exp\left[\ln\left(\left(\frac{1-\varepsilon}{\varepsilon}\right)^{\frac{1}{2}}\right)\right] && \text{power rule of logarithms} \\ &= \sqrt{\frac{1-\varepsilon}{\varepsilon}} && \text{recall that } \ln x = \log_e x \text{ and } a^{\log_a n} = n \end{aligned}$$

(it should be obvious now why we said $\alpha \geq 0$)

We can follow the same reasoning for $e^{-\alpha}$ and get a flipped result:

$$e^{-\alpha} = \exp\left(-\frac{1}{2}\cdot\ln\frac{1-\varepsilon}{\varepsilon}\right) = \left(\frac{1-\varepsilon}{\varepsilon}\right)^{-1/2} = \left(\frac{\varepsilon}{1-\varepsilon}\right)^{1/2} = \sqrt{\frac{\varepsilon}{1-\varepsilon}}$$

So we have two general outcomes depending on whether or not the weak learner classified $\mathbf{x}_i$ correctly (dropping the $\sqrt{\cdot}$ for simplicity of analysis):

$$f(\varepsilon) = \exp^2(-\alpha_t y_i H_t(\mathbf{x}_i)) = \begin{cases} \dfrac{1-\varepsilon}{\varepsilon} & \text{if } H(\cdot) \text{ was } \textit{wrong} \\ \dfrac{\varepsilon}{1-\varepsilon} & \text{if } H(\cdot) \text{ was } \textit{right} \end{cases}$$

Remember that ε_t is the total error of all incorrect answers that H_t gave; it's a sum of probabilities, so $0 < \varepsilon < 1$. But note that H is a weak learner, so it *must* do better than random chance, so in fact $0 < \varepsilon < \frac{1}{2}$. The functions are plotted in Figure 1.2; from them, we can draw some straightforward conclusions:

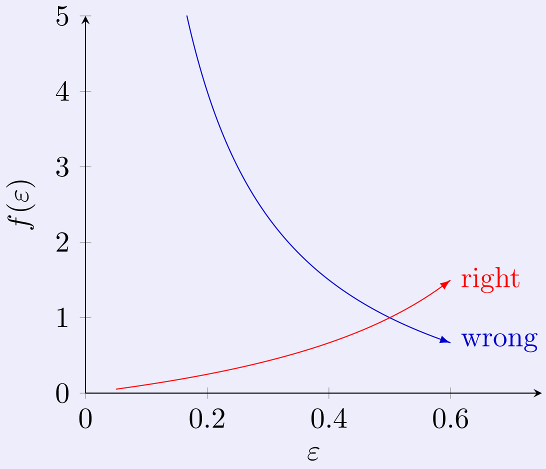


Figure 1.2: The two ways the exponential can go when boosting, depending on whether or not the classifier gets sample i right ($\frac{1-\varepsilon}{\varepsilon}$, in red) or wrong ($\frac{\varepsilon}{1-\varepsilon}$, in blue).

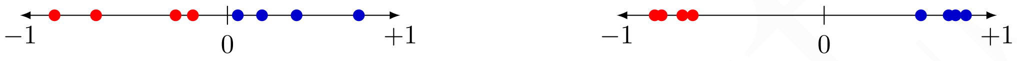
- When our learner is **incorrect**, the weight of sample i increases exponentially as we get more and more confident ($\varepsilon \to 0$) in our model.
- When our learner is **correct**, the weight of sample i will decrease (relatively) proportionally with our overall confidence.

To summarize these two points in different words: a learner will always weigh incorrect results more than correct results, but will focus on incorrect results more and more as the learner's overall performance increases.

Considerations: Overfitting

Boosting is a robust method that tries very hard to avoid overfitting. Testing performance often mimics training performance pretty closely. Why is this the case? Though we've been discussing **error** at length up to this point, and how minimizing error has been our overarching goal, it's worth discussing the idea of **confidence**, as well. If, for example, you had a 5-nearest neighbor in which three neighbors strongly voted one way and two neighbors voted another way, you might have low error but also low confidence, relative to a scenario where all five voted the same way.

Because boosting is insecure and suffers from social anxiety, it tries really hard to be confident and this lets it avoid overfitting. Once a boosted ensemble reaches a state at which it has low error and can separate positive and negative examples well, adding more and more weak learners will actually continue to spread the gap (similar to support vector machines which we're about to discuss with respect to margins) at the boundary.



However, nobody's perfect. Boosting still tends to overfit in an oddly-specific case Dr. Isbell highlights: when its underlying weak learner uses a complex artificial neural network (one with many nodes and layers). More generally, if the underlying learners all overfit and can't stop overfitting (like a complex neural net tends to do), boosting can't do anything to prevent that. Boosting also suffers under **pink noise**—uniform noise—and tends to overfit.[2]

1.3 Support Vector Machines

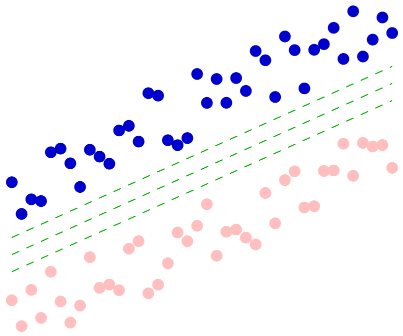
Let's return to the notion of a dataset being linearly-separable. From the standpoint of human cognition, finding a line that cleanly divides two colors is pretty easy. In general when we're trying to classify something into one category or another, there's no reason for us to consider anything but the specific details that truly separate the two categories. We hardly pay attention to the bulk of the data, focusing on what defines the *boundary* between them.

This is the motivation behind support vector machines.

I covered SVMs briefly in my notes on computer vision. For a gentler introduction to the topic, refer there. This section will have a lot more assumed knowledge so that I can avoid repeating myself.

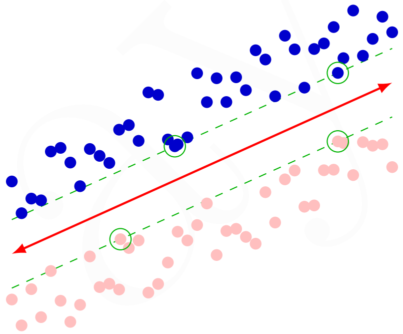
Now, which of the green dashed lines below "best" separates the two colors?

[2] "White noise" is Gaussian noise.



They're all correct, but why does the middle one "feel" best? Aesthetics? Maybe. But more likely, it's because the middle line does the best job at separating the data without making us commit too much to it. It leaves the biggest margin of potential error if some hidden dots got revealed.

A **support vector machine** operates on this exact notion: it tries find the boundary that will maximize the margin from the nearest data points. The optimal margin lines will always have some special points that intersect the dashed lines:



These points are called the **support vectors**. Just like a human, a support vector machine reduces computational complexity by focusing on examples near the boundaries rather than the entire data set. So how can we use these vectors to maximize the margin?

1.3.1 There are lines and there are lines...

Let's briefly note that a line in 2D is typically represented in the form $y = mx + b$. However, we want to generalize to n dimensions.

The standard form of a line in 2D is defined as: $ax + by + c = 0$, where $a, b, c \in \mathbb{Z}$ and $a > 0$. From this, we can imagine a "compact" representation of the line that only uses its constants, so if we want the original equation back, we can dot this vector with $\begin{bmatrix} x & y & 1 \end{bmatrix}$:

$$\begin{bmatrix} a & b & c \end{bmatrix} \cdot \begin{bmatrix} x & y & 1 \end{bmatrix} = 0$$

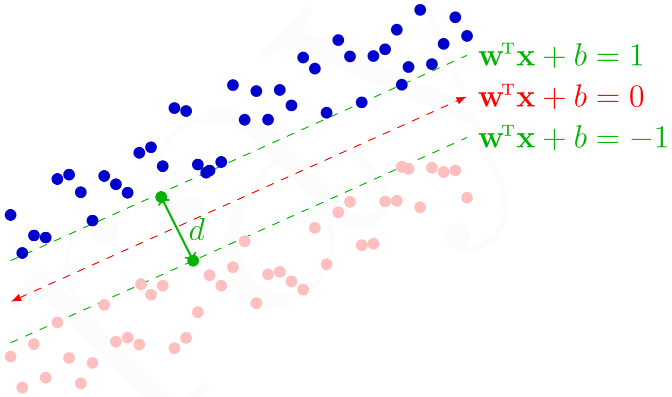
Thus if we let $\mathbf{s} = \begin{bmatrix} a & b & c \end{bmatrix}$ and $\mathbf{w} = \begin{bmatrix} x & y & 1 \end{bmatrix}$, then our line can be expressed simply as: $\mathbf{w}^\mathsf{T}\mathbf{s} = 0$. This lets us representing a line in vector form and use any number of dimensions. Our $\mathbf{w}$ defines the parameters of the (hyper)plane; notice that here, $\mathbf{w} \perp$ the xy-plane.

If we want to stay reminiscent of $y = mx + b$, we can drop the last term of $\mathbf{w}$ and use the raw constant: $\mathbf{w}^\mathsf{T}\mathbf{s} + c = 0$.

1.3.2 Support Vectors

Similar to what we did with boosting, we'll say that when $y \geq 1$, the input value $\mathbf{x}$ was part of the class, whereas if $y \leq -1$ it wasn't (take care to differentiate the fact that y is the label now rather than something related to the y axis). Then a line in our "label space" is of the form $y = \mathbf{w}^\mathsf{T}\mathbf{x} + b$.

What's the output, then, of the line that divides the two classes? Well it's exactly between all of the 1s and the -1s, so it must be the line $\mathbf{w}^\mathsf{T}\mathbf{x} + b = 0$. Similarly, the two decision boundaries are exactly ± 1. Note that we don't know $\mathbf{w}$ or b yet, but know we want to maximize d:


$\mathbf{w}^\mathsf{T}\mathbf{x} + b = 1$
$\mathbf{w}^\mathsf{T}\mathbf{x} + b = 0$
$\mathbf{w}^\mathsf{T}\mathbf{x} + b = -1$
d

Well if we call the two green support vectors above $\mathbf{x}_1$ and $\mathbf{x}_2$, what's the distance between them? Well,

$$\begin{aligned} \mathbf{w}^\mathsf{T}\mathbf{x}_1 + b &= 1 \\ -(\mathbf{w}^\mathsf{T}\mathbf{x}_2 + b &= -1) \\ \hline \mathbf{w}^\mathsf{T}(\mathbf{x}_1 - \mathbf{x}_2) &= 2 \\ \hat{\mathbf{w}}^\mathsf{T}(\mathbf{x}_1 - \mathbf{x}_2) &= \frac{2}{\|\mathbf{w}\|} \end{aligned}$$

Thus we want to maximize $M = \frac{2}{\|\mathbf{w}\|}$ (M for **margin**), while also classifying all of our data points correctly. Mathematically-speaking, maximization is harder than minimization (thank u calculus), so we're actually better off optimizing for $\min \frac{1}{2}\|\mathbf{w}\|^2$.

So if we define $y_i \in \{-1, 1\}$ for every training sample as we did when Boosting, then we arrive at a standard **quadratic optimization problem**:

$$\boxed{\begin{array}{rl} \text{Minimize:} & \dfrac{1}{2}\left\|\mathbf{w}\right\|^2 \\ \text{Subject to:} & y_i(\mathbf{w}^\mathrm{T}\mathbf{x}_i + b) \geq 1 \end{array}}$$

which is a well-understood, always-solveable optimization problem whose solution is just a linear combination of the support vectors:

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$

where the α_is are "learned weights" that are only non-zero at the support vectors. Any support vector i is defined by $y_i = \mathbf{w}^\mathrm{T}\mathbf{x}_i + b$, so:

$$y_i = \underbrace{\sum_i \alpha_i y_i \mathbf{x}_i^\mathrm{T}}_{\mathbf{w}^\mathrm{T}} \mathbf{x} + b = \pm 1$$

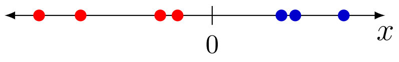
We can use this to build our classification function: $f(\mathbf{x}) = \operatorname{sign}\left(\sum_i \alpha_i y_i \boxed{\mathbf{x}_i^\mathrm{T}\mathbf{x}} + b\right)$

Note the highlighted box: the entirety of the classification depends *only* on this dot product between some "new point" $\mathbf{x}$ and our support vectors $\mathbf{x}_i$s. The dot product is a metric of similarity: here, it's the projection of each $\mathbf{x}_i$ onto the new $\mathbf{x}$, but it doesn't have to be...

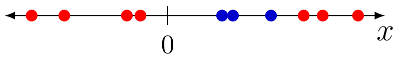
1.3.3 Extending SVMs: The Kernel Trick

We've been working with nice, neat 2D plots and drawing a line to separate the data. This begs the question: **what if the data isn't linearly-separable?** The answer to this question is the source of power of SVMs: we'll use it to find separation boundaries between our data points in higher dimensions than our features provide.

Obviously, we can find the optimal separator between the following group of points:



But what about these?



No such luck this time. But what if we mapped them to a higher-dimensional space? For example, if we map these to $y = x^2$, a wild linear separator appears!

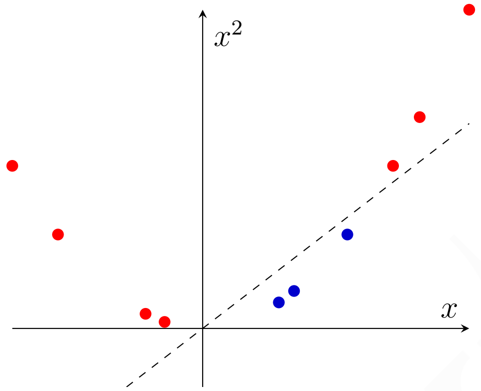


Figure 1.3: Finding a linear separator by mapping to a higher-dimensional space.

This seems promising… how can we find such a mapping (like the arbitrary $x \mapsto x^2$ above) for other feature spaces? Notice that we added a dimension simply by manipulating the *representation* of our features.

Let's generalize this idea. We can call our mapping function Φ; it maps the $\mathbf{x}$s in our feature space to another higher-dimensional space $\varphi(\mathbf{x})$, so $\Phi : \mathbf{x} \mapsto \varphi(\mathbf{x})$. And recall the "similarity metric" in our classification function; let's isolate it and define $K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\mathsf{T}\mathbf{b}$. Then, we have

$$f(\mathbf{x}) = \operatorname{sign}\left(\sum_i \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b\right)$$

The **kernel trick** here is simple: it states that if there exists *some* $\Phi(\cdot)$ that can represent $K(\cdot)$ as a dot product, we can actually use K in our linear classifier. We don't actually need to find, define, or care about Φ, it just needs to exists. And in practice, it *does* exist for almost any function we can think of (though I won't offer an explanation on how). That means we can apply almost any K and it'll work out. For example, if our 2D dataset was separable by a circular boundary, we could use $K(\mathbf{a}, \mathbf{b}) = \left(\mathbf{a}^\mathsf{T}\mathbf{b}\right)^2$ in our classifier and it would actually find the boundary, despite it not being linearly-separable in *two* dimensions.

Soak that in for moment… we can almost-arbitrarily define a K that represents our data in a different way and it'll still find a boundary that just so happens to be linear in a higher-dimensional space.

Example 1.2: A Simple Polynomial Kernel Function

Let's work through a proof that a particular kernel function is a dot product in some higher-dimensional space. Remember, we don't actually care about what that space *is* when it comes to applying the kernel; that's the beauty of the kernel trick. We're working through this to demonstrate how you would show that some kernel function does have a higher-dimensional mapping.

We have 2D vectors, so $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$.

Let define the following kernel function: $K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j)^2$

This is a simple **polynomial kernel**; it's called that because we are creating a polynomial from the dot product. To prove that this is a valid kernel function, we need to show that $K(\mathbf{x}_i, \mathbf{x}_j) = \varphi(\mathbf{x}_j)^{\mathrm{T}}\varphi(\mathbf{x}_j)$ for *some* φ.

$$\begin{aligned}
K(\mathbf{x}_i, \mathbf{x}_j) &= (1 + \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j)^2 \\
&= \left(1 + \begin{bmatrix} x_{i1} & x_{i2} \end{bmatrix} \begin{bmatrix} x_{j1} \\ x_{j2} \end{bmatrix}\right)\left(1 + \begin{bmatrix} x_{i1} & x_{i2} \end{bmatrix} \begin{bmatrix} x_{j1} \\ x_{j2} \end{bmatrix}\right) && \text{expand} \\
&= 1 + x_{i1}^2 x_{ij}^2 + 2x_{i1}x_{j1}x_{i2}x_{j2} + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{j1} + 2x_{i2}x_{j2} && \text{multiply it all out} \\
&= \begin{bmatrix} 1 & x_{i1}^2 & \sqrt{2}x_{i1}x_{i2} & x_{i2}^2 & \sqrt{2}x_{i1} & \sqrt{2}x_{i2} \end{bmatrix} \begin{bmatrix} 1 \\ x_{j1}^2 \\ \sqrt{2}x_{j1}x_{j2} \\ x_{j2}^2 \\ \sqrt{2}x_{j1} \\ \sqrt{2}x_{j2} \end{bmatrix} && \text{rewrite it as a vector product}
\end{aligned}$$

At this point, we can see something magical and crucially important: each of the vectors only relies on terms from *either* $\mathbf{x}_i$ or $\mathbf{x}_j$! That means it's a... wait for it... dot product! We can define φ as a mapping into this new 6-dimensional space:

$$\varphi(\mathbf{x}) = \begin{bmatrix} 1 & x_1^2 & \sqrt{2}x_1x_{n2} & x_2^2 & \sqrt{2}x_1 & \sqrt{2}x_2 \end{bmatrix}^{\mathrm{T}}$$

Which means now we can express K in terms of dot products in φ, exactly as we wanted:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \varphi(\mathbf{x_i})^{\mathrm{T}}\varphi(\mathbf{x_j}) \qquad \square$$

The choice of $K(\cdot)$ is much like the choice of $d(\cdot)$ in k-nearest neighbor: it encodes *domain knowledge* about the data in question that can help us classify it better. Some common kernels include polynomial kernels (like the one in the example) and the **radial basis kernel** which is essentially a Gaussian:

$$\begin{aligned}
K(\mathbf{a}, \mathbf{b}) &= \dots \\
&= (\mathbf{a}^{\mathrm{T}}\mathbf{b} + c)^p && \text{(polynomial)} \\
&= \exp\left(-\frac{\|\mathbf{a} - \mathbf{b}\|^2}{2\sigma^2}\right) && \text{(radial basis)}
\end{aligned}$$

1.3.4 Summary

In general, we've come to the conclusion that finding the linear separator of a dataset with the maximum margin is a good way to generalize and avoid overfitting. SVMs do this with a quadratic optimization problem to find the support vectors. Finally, we discussed how the kernel trick lets us find these linear separators in an arbitrary n-dimensional space by projecting the data to said space.

ALGORITHM 1.1: A simplified AdaBoost algorithm. Note that $y_i \in \{-1, 1\}$, so we're working with classification here, but it can just as easily be adapted to regression. *(this algorithm comes from my notes on computer vision)*

```
Input: $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$ and $\mathcal{Y} = \{y_1, y_2, \ldots, y_m\}$, a training set mapping
       both positive and negative training examples to their corresponding
       labels: $\mathbf{x}_i \mapsto y_i$.
Input: $H(\cdot)$: a weak classifier type.
Result: A boosted classifier, $H^*$.
$\hat{\mathbf{w}} = \left[{}^1\!/\!_m \quad {}^1\!/\!_m \quad \ldots\right]$          // a uniform weight distribution
$t \approx 0$          // some small threshold value close to 0
foreach training stage $j \in [1..n]$ do
    $\hat{\mathbf{w}} = {}^{\mathbf{w}}\!/\!_{\|\mathbf{w}\|}$
    $h_j = H(\mathcal{X}, \mathcal{Y}, \hat{\mathbf{w}})$    // train a weak learner for the current weights
    /* The error is the sum of the incorrect training predictions.   */
    $\varepsilon_j = \sum_{i=0}^{M} w_i \qquad \forall w_i \in \hat{\mathbf{w}}$ where $h_j(\mathbf{x}_i) \neq y_i$
    $\alpha_j = \frac{1}{2} \ln\left(\frac{1-\varepsilon_j}{\varepsilon_j}\right)$
    /* Update the weights only if the error is large enough.   */
    if $\varepsilon > t$ then
        $w_i = w_i \cdot \exp\left(-y_i \alpha_j h_j(\mathbf{x}_i)\right) \qquad \forall w_i \in \mathbf{w}$
    else
        break
end
/* The final boosted classifier is the sum of each $h_j$ weighed by its
   corresponding $\alpha_j$. Prediction on an new $\mathbf{x}$ is then simply:   */
$H^*(\mathbf{x}) := \text{sign}\left[\sum_{j=0}^{M} \alpha_j h_j(\mathbf{x})\right]$
return $H^*$
```

Regression

> *We stand the risk of regression, because you refused to take risks. So life demands risks.*
>
> — Sunday Adelaja

When we free ourselves from the limitation of small discrete quantities of labels, we are open to approximate a much larger range of functions. Machine learning techniques that use regression can approximate real-valued and continuous functions.

In supervised learning, we are trying to perform *inductive* reasoning, in which we try to figure out the abstract bigger picture from tiny snapshots of the world in which we don't know most of the rules (that is, approximate a function from input-output pairs). This is in stark contrast with *deductive* reasoning, through which individual facts are combined through strict, rigorous logic to come to bigger conclusions (think "if *this*, then *that*").

2.1 Linear Regression

You've likely heard of **linear regression** if you're reading these notes. Linear regression is the mathematical process of acquiring the "line-of-best fit" that we used to plot in grade school. Through the power of linear algebra, the line of best fit can be rigorously defined by solving a linear system.

One way to find this line is to find the sum of least squared-error between the points and the chosen line; more specifically, a visual demonstration can show

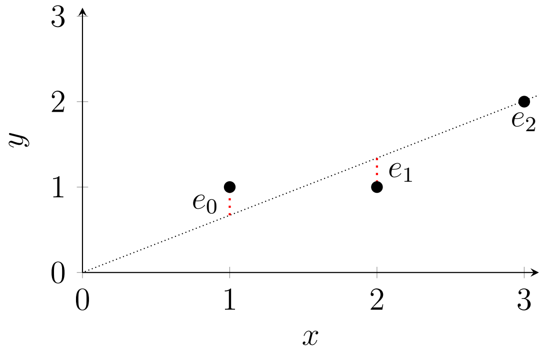


Figure 2.1: A set of points with "no solution": no line passes through all of them. The set of errors is plotted in red: (e_0, e_1, e_2).

us this is the same as minimizing the **projection** error of the points on the line.

Suppose we have a set of points that don't exactly fit a line: $\{(1,1),(2,1),(3,2)\}$, plotted in Figure 2.1. We want to find the best possible line $y = mx + b$ that minimizes the total error. This corresponds to solving the following system of equations, forming $\mathbf{y} = \mathbf{A}\mathbf{x}$:

$$\begin{cases} 1 = b + m \cdot 1 \\ 1 = b + m \cdot 2 \\ 2 = b + m \cdot 3 \end{cases} \implies \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

The lack of an exact solution to this system (algebraically) means that the vector of y-values isn't in the **column space** of $\mathbf{A}$, or: $\mathbf{y} \notin C(\mathbf{A})$. The vector can't be represented by a linear combination of column vectors in $\mathbf{A}$.

We can imagine the column space as a plane in xyz-space, and $\mathbf{y}$ existing outside of it; then, the vector that'd be *within* the column space is the projection of $\mathbf{y}$ into the column space plane: $\mathbf{p} = \operatorname*{proj}_{C(\mathbf{A})} \mathbf{y}$. This is the closest possible vector in the column space to $\mathbf{y}$, which is exactly the distance we were trying to minimize! Thus,

$$\mathbf{e} = \mathbf{y} - \mathbf{p}$$

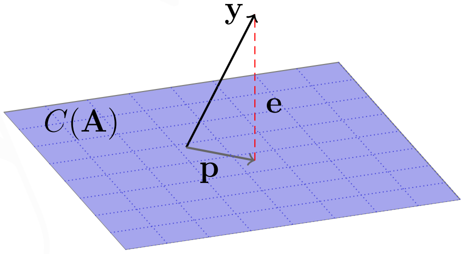


Figure 2.2: The vector $\mathbf{y}$ relative to the column space of $\mathbf{A}$, and its projection $\mathbf{p}$ onto the column space.

The projection isn't super convenient to calculate or determine, though. Through algebraic manipulation, calculus, and other magic, we learn that the way to find the least squares approximation of the solution is:

$$\begin{aligned} \mathbf{A}^\mathsf{T}\mathbf{A}\mathbf{x}^* &= \mathbf{A}^\mathsf{T}\mathbf{y} \\ \mathbf{x}^* &= \underbrace{\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)^{-1}}_{\textbf{pseudoinverse}} \mathbf{A}^\mathsf{T}\mathbf{y} \end{aligned}$$

which is exactly what the linear regression algorithm calculates.

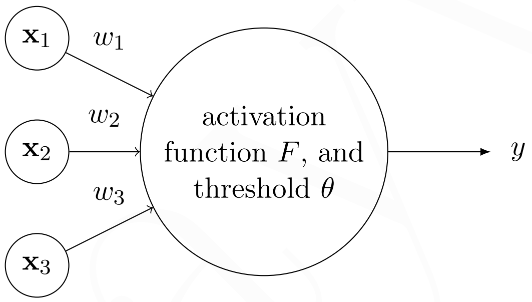
More Resources. This section basically summarizes and synthesizes this Khan Academy video, this lecture from the *Computer Vision* course (which goes through the *full* derivation), this section of *Introduction to Linear Algebra*, and this explanation from NYU. These links are provided in order of clarity.

2.2 Neural Networks

Let's dive right into the deep end and learn about how a **neural network** works.

Neurons in the brain can be described relatively succinctly from a high level: a single neuron is connected to a bunch of other neurons. It accumulates energy and once the amount is bigger than the "**activation threshold**," the neuron fires, sending energy to the neurons its connected to (potentially causing a cascade of firings). Artificial neural networks take inspiration from biology and are somewhat analogous to neuron interactions in the brain, but there's really not much benefit to looking at the analogy further.

The basic model of an "artificial neuron" is a function powered by a series of inputs $\mathbf{x}_i$, and weights w_i, that somehow run through F and produce an output y:



Typically, the activation function "fires" based on a **firing threshold** θ.

2.2.1 Perceptron

The simplest, most fundamental activation function procudes a binary output (so $y \in \{0, 1\}$) based on the weighted sum of the inputs:

$$F(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n, w_1, w_2, \ldots, w_n) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i \mathbf{x}_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

This is called a **perceptron**, and it's the foundational building block of neural networks going back to the 1950s.

Example 2.1: Basic Perceptron

Let's quickly validate our understanding. Given the following input state:

$$\begin{aligned} x_1 &= 1, w_1 = \frac{1}{2} \\ x_2 &= 0, w_2 = \frac{3}{5} \\ x_3 &= -1.5, w_3 = 1 \end{aligned}$$

and the firing threshold $\theta = 0$, what's the output?

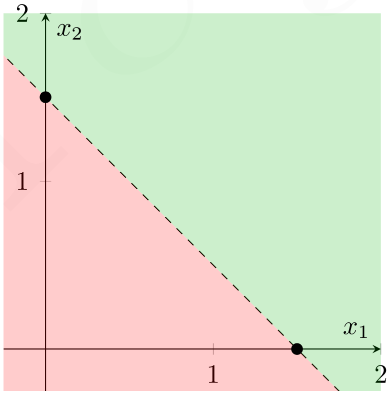
Well, pretty simply, we get

$$y = \left[1\left(\frac{1}{2}\right) + 0\left(\frac{3}{5}\right) + (-1.5)\cdot 1 = \frac{1}{2} - \frac{3}{2} = -1\right] < 0 = 0$$

The perceptron doesn't fire!

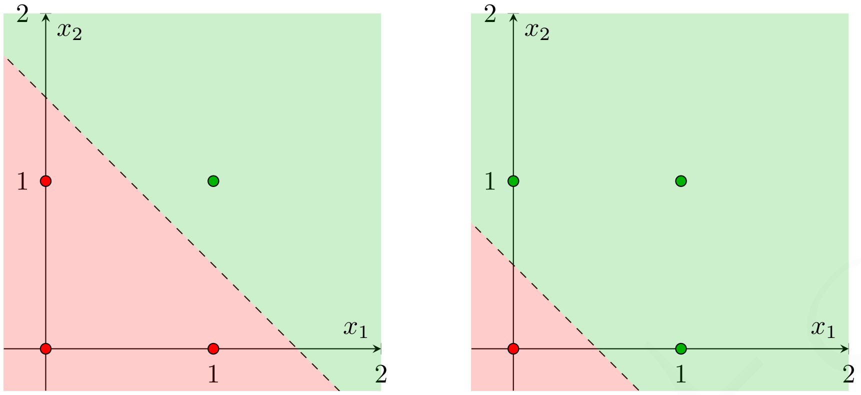
What kind of functions can we represent with a perceptron? Suppose we have two inputs, x_1 and x_2, along with equal weights $w_1 = w_2 = \frac{1}{2}, \theta = \frac{3}{4}$. For what values of x_i will we get an activation?

Well, we know that if $x_2 = 0$, then we'll get an activation for anything that makes $x_1 w_1 \geq \theta$, so $x_1 \geq {}^{\theta}/_{w_1} \geq 1.5$. The same rationale applies for x_2, and since we know that the inequality is linear, we can just connect the dots:



Thus, a perceptron is a linear function (each $w_i \mathbf{x}_i$ term is linear), and so it can be used to compute the **half-plane** boundary between its inputs.

This very example actually computes an interesting function: if the inputs are binary (that is, $x_1, x_2 \in \{0, 1\}$), then **this actually computes binary AND operation**. The only possible outputs are marked accordingly; only when $x_1 = x_2 = 1$ does $y = 1$! We can actually model OR the same way with different weights (like $w_1 = w_2 = \frac{3}{2}$).



Note that we "derived" OR by adjusting $w_{1,2}$ until it worked, though we could've also adjusted θ. This might trigger a small "aha!" moment if the idea of induction from stuck with you: if we have some known input/output pairs (like the truth tables for the binary operators), then we can use a computer to rapidly guess-and-check the weights of a perceptron (or perhaps an entire neural network. . . ?) until they accurately describe the training pairs as expected.

Combining Perceptrons

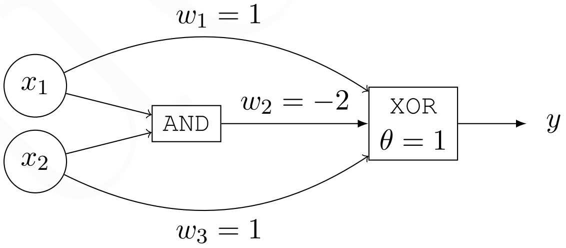
To build up an intuition for how perceptrons can be combined, let's binary XOR. It can't be described by a single perceptron because it's not a linear function; however, it *can* be described by several!

Intuitively, XOR is like OR, except when the inputs succeed under AND. . . so we might imagine that XOR $\approx$ OR$-$AND.

x	y	$x \oplus y$
1	1	0
1	0	1
0	1	1
0	0	0

Table 2.1: The truth table for XOR.

So if x_1 and x_2 are both "on," we should take away the result of the AND perceptron so that we fall under the activation threshold. However, if only one of them is "on," we can proceed as normal. Note that the "deactivation weight" needs to be equal to the sum of the other weights in order to effectively cancel them out, as shown Figure 2.3.



x_1	x_2	AND	y
1	1	1	$1 \cdot 1 + -2 \cdot 1 + 1 \cdot 1 = 0$
1	0	0	$1 \cdot 1 + -2 \cdot 0 + 1 \cdot 0 = 1$
0	1	0	$1 \cdot 0 + -2 \cdot 0 + 1 \cdot 1 = 1$
0	0	0	$1 \cdot 0 + -2 \cdot 0 + 1 \cdot 0 = 0$

Figure 2.3: A neural network modeling XOR and its summations for all possible bit inputs. The final column in the table is the summation expression for perceptron activation, $w_1x_1 + w_2F_{\text{AND}}(x_1, x_2) + w_3x_2$.

Learning Perceptrons

Let's delve further into the notion of "training" a perceptron network that we alluded to earlier: twiddling the w_is and θs to fit some known inputs and outputs. There are two approaches to this: the **perceptron rule**, which operates on the post-activated y-values, and **gradient descent**, which operates on the raw summation.

Perceptron Rule Suppose $\hat{y}$ is our perceptron's current output, while y is its desired output. In order to "move towards" the goal y, we adjust our w_is accordingly based on the "error" between y and $\hat{y}$. That is,

$$\begin{aligned} \text{define } \hat{y} &= \left(\textstyle\sum_i w_i x_i \geq 0\right) \\ \text{then use } \Delta w_i &= \eta\left(y - \hat{y}\right) x_i \\ \text{to adjust } w_i &\leftarrow w_i + \Delta w_i \end{aligned}$$

where $\eta > 0$ is a **learning rate** which influences the size of the Δw_i adjustment made every iteration.

Notice the absense of the activation threshold, θ. To simplify the math, we can actually treat it as part of the summation: a "fake" input with a fixed weight of -1, since:

$$\begin{aligned} \textstyle\sum_i^n w_i x_i = \theta &\implies \textstyle\sum_i^n w_i x_i - \theta = 0 \\ &\implies \textstyle\sum_i^{n+1} w_i x_i = 0 \qquad \text{where } w_{n+1} = -1 \text{ and } x_{n+1} = \theta \end{aligned}$$

This extra input value is now called the **bias** and its weight can be tweaked just like the other inputs.

Notice that when our output is correct, $y - \hat{y} = 0$ so there is no effect on the weights. If $\hat{y}$ is too big, then our $\Delta w_i < 0$, making that $w_i x_i$ term smaller, and vice-versa if $\hat{y}$ is too small. The learning rate controls our adjustments so that we take small steps in the right direction rather than overshooting, since we don't know how close we are to fixing $\hat{y}$.

Theorem 2.1. *If a dataset is* ***linearly-separable*** *(that is, able to be separated by a line), then a perceptron can find it with a finite number of iterations by using the perceptron rule.*

Of course, it's impossible to know whether a sufficiently-large "finite" number of a iterations has occurred, so we can't use this fact to determine whether or not a dataset is linearly-separable, but it's still good to know that it will terminate when it can.

Gradient Descent We still need something in our toolkit for datasets that aren't linearly-separable. This time, we'll operate on the unthresholded summation since it gives us far more information about how close (or far) we are from triggering an activation. We'll use a as shorthand for the summation: $a = \sum_i w_i x_i$.

Then we can define an error metric based on the difference between a and each expected output: we sum the error for each of our input/output pairs in the dataset D. That is,

$$E(\mathbf{w}) = \frac{1}{2} \sum_{(x,y) \in D} (y - a)^2$$

Let's use our good old friend calculus to solve this via **gradient descent**. A function is at its minimum when its derivative is zero, so we'll take the derivative with respect to a weight:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \left[\frac{1}{2} \sum_{(x,y) \in D} (y - a)^2 \right] \\
&= \sum_{(x,y) \in D} (y - a) \cdot \frac{\partial}{\partial w_i} \left[- \sum_j w_j x_j \right] && \text{chain rule, and only } a \text{ is in terms of } w_i \\
&= \sum_{(x,y) \in D} (y - a)(-x_i) && \text{when } j \neq i \text{, the derivative will be zero} \\
&= - \sum_{(x,y) \in D} (y - a) x_i && \text{rearranged to look like the perceptron rule}
\end{aligned}$$

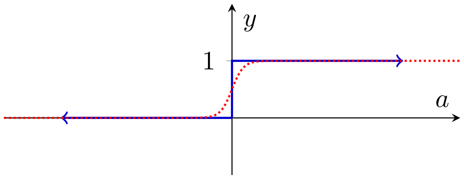
Notice that we essentially end up with a version of the perceptron rule where $\eta = -1$, except we now use the summation a instead of the binary output $\hat{y}$. Unlike the perceptron rule, we have no guarantees about finding a separation, but it is far more robust to non-separable data. In the limit (thank u Newton very cool), though, it will converge to a local optimum.

To reiterate our learning rules, we have:

$$\Delta w_i = \eta(y - \hat{y}) x_i \tag{2.1}$$
$$\Delta w_i = \eta(y - a) x_i \tag{2.2}$$

2.2.2 Sigmoids

The similarity between the learning rules in (2.1) and (2.2) begs the question, why didn't we just use calculus on the thresholded $\hat{y}$?



The simple answer is that the function isn't differentiable. Wouldn't it be nice, though if we had one that was very similar to it, but smooth at the hard corners that give us problems? Enter the **sigmoid** function:

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{2.3}$$

By introducing this as our activation function, we can use gradient descent all over the place. Furthermore, the sigmoid's derivative itself is beautiful (thanks to the fact that $\frac{d}{dx}e^x = e^x$):

$$\dot{\sigma}(a) = \sigma(a)(1 - \sigma(a))$$

Note that this isn't the only function that smoothly transitions between 0 and 1; there are other activation functions out there that behave similarly.

2.2.3 Structure

Now that we have the ability to put together differentiable individual neurons, let's look at what a large neural network presents itself as. In Figure 2.4 we see a collection of sigmoid units arranged in an arbitrary pattern. Just like we combined two perceptrons to represent the non-linear function XOR, we can combine a larger number of these sigmoid units to approximate an arbitrary function.

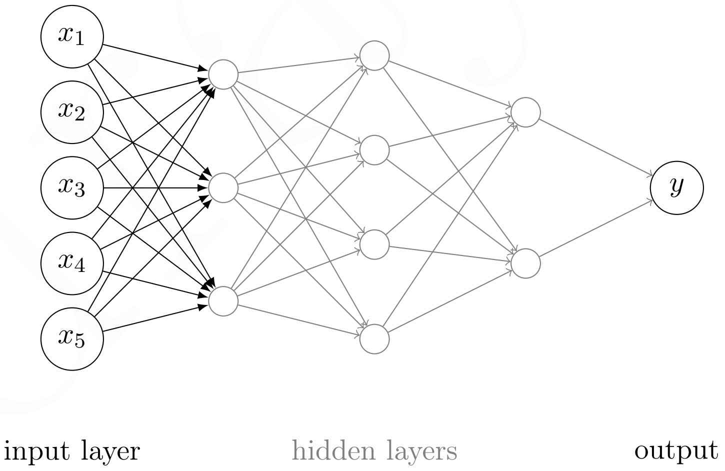


Figure 2.4: A 4-layer neural network with 5 inputs and 3 hidden layers.

Because each individual unit is differentiable, the entire mapping from $\mathbf{x} \mapsto y$ is differentiable! The overall error of the system enables a bidirectional flow of information: the error of y impacts the last hidden layer, which results in its own error, which impacts the second-to-last hidden layer, etc. This layout of computationally-beneficial organizations of the chain rule is called **back-propagation**: the error of the network propogates to adjust each unit's weight individually.

Optimization Methods

It's worth reiterating that we've departed from the guarantees of perceptrons since we're using $\sigma(a)$ instead of the binary activation function; this means that gradient descent can get stuck in local optima and not necessarily result in the best *global* approximation of the function in question.

Gradient descent isn't the only approach to training a neural network. Other, more advanced methods are researched heavily. Some of these include **momentum**, which allows gradient descent to "gain speed" if it's descending down steep areas in the function; higher-order derivatives, which look at combinations of weight changes to try to grasp the bigger picture of how the function is changing; randomized optimization; and the idea of penalizing "complexity," so that the network avoids overfitting with too many nodes, layers, or even too-large of weights.

2.2.4 Biases

What kind of problems are neural networks appropriate for solving?

Restriction Bias A neural network's restriction bias (which, if you recall, is the representation's ability to consider hypotheses) is basically non-existent if you use sigmoids, though certain models may require arbitrarily-complex structure.

We can clearly represent Boolean functions with threshold-like units. Continuous functions with no "jumps" can actually be represented with a single hidden layer. We can think of a hidden layer as a way to stitch together "patches" of the function as they approach the output layer. Even arbitrary functions can be approximated with a neural network! They require two hidden layers, one stitching at seams and the other stitching patches.

This lack of restriction does mean that there's a significant danger of overfitting, but by carefully limiting things that add complexity (as before, this might be layers, nodes, or even the weights themselves), we can stay relatively generalized.

Preference Bias On the other hand, we can't yet answer the question of the preference bias of a neural network (which, if you recall, describes which hypotheses *from the restricted space* are preferred). We discussed the algorithm for updating weights (gradient descent), but have yet to discuss how the weights should be initialized in the first place.

Common practice is choosing small, random values for our initial weights. The randomness allows for variability so that the algorithm doesn't get stuck at the same local minima each time; the smallness allows for relative adjustments to be impactful and reduces complexity.

Given this knowledge, we can say that neural networks—when all other things are equal—prefer simpler explanations to complex ones. This idea is an embodiment of **Occam's Razor**:

> *Entities should not be multiplied unnecessarily.*

More colloquially, it's often expressed as the idea that the simpler explanation is likelier to be true.

2.3 Instance-Based Learning

The learning algorithms presented in this section take a radically-different approach to modeling a function approximation. Instead of inducing an abstract model from the training set that compactly represents most of the data, these algorithms will actually regularly refer to the data itself to create approximations.

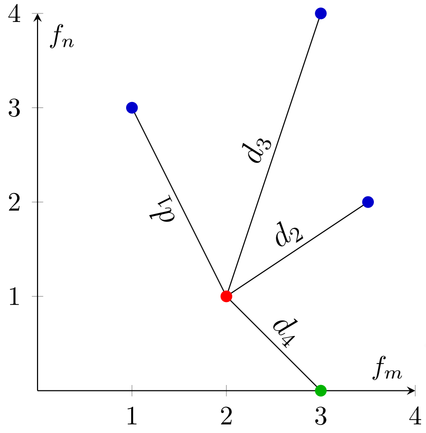
For a high-level example, consider our fundamental "line of best fit" problem. Given some input points, linear regression will determine the line that minimizes the error with the data, and then we can use the line directly to predict new values. With instance-based learning methods, we instead would base our predictions from the points themselves. We might predict the output of a novel input x' as being the same as the output of whatever known x is closest to it, or perhaps the average of the outputs of the three known inputs closest to it.

There are some pretty clear benefits to this paradigm shift: we no longer need to actually spend time "learning" anything; the model perfectly-remembers the training data rather than remembering an abstract generalizing; and the approach is dead-simple. The downsides, of course, are that we have to store all of our training data (which might potentially require massive amounts of storage) and we are not really generalizing from it (we're pretty obviously overfitting) at all.

2.3.1 Nearest Neighbors

This idea of referring to similar known inputs for a novel input is exactly the intuition behind the **k-nearest neighbor** learning algorithm. While k is the number of knowns to consider, we also need a notion of "distance" to determine how close or similar an $x_i \in \mathcal{X}$ is to the novel input x'.

The distance is our expression of domain knowledge about the space. If we're classifying restaurants into cheap, average, and expensive, our distance metric might be the difference between the average entreé price. We might even be talking about literal distances: if we had some arbitarily-colored dots (whose color was determined by the features f_m and f_n) and wanted to determine the color of a novel dot (encoded in red below) with $f_m = 2, f_n = 1$, we'd use the standard Euclidean distance.



Notice that x_4 is closest, followed by x_2. If we were doing classification, we would probably choose specifically blue or green (depending on our tie-breaking scheme), but in regression, we might instead color the novel dot by the weighted sum of its $k = 2$ nearest neighbors:

$$\begin{aligned} y' &= \frac{d_4 y_4 + d_2 y_2}{d_4 + d_2} \approx \frac{2.8 \cdot \text{blue} + 1.8 \cdot \text{green}}{2.8 + 1.8} \\ &= (0, 155, 100) && \text{in RGB} \in [0, 255] \text{ terms} \end{aligned}$$

which is a nice, dark blue-green color. ■

An extremely simple (and non-performant) version of the kNN algorithm is formalized in algorithm 2.1; notice that it has $O(k\,|\mathcal{X}|)$ complexity, meaning it grows linearly both in k, the number of neighbors to consider, and in $|\mathcal{X}|$, the size of the training data.

Obviously we can improve on this. With a sorted list ($O(n \log n)$ time), we can apply binary search ($O(\log n + k)$ time for k values) and query far more efficiently. For features with high dimensionality (i.e. when $n \gg 0$ for $\mathbf{x}_i = \{f_1, f_2, \ldots, f_n\}$), we can also leverage the kd-tree algorithm[1]—which subdivides the data into sectors to search through them more efficiently—but is still $O(kn \log n)$. This is the main downside of kNN: because we use the training data itself as part of the querying process, things can get slow and unwieldy (in both time and space) very quickly.

This is in contrast with something like linear regression, which calculates a model upfront and makes querying very cheap (constant time, in fact); in this regard, kNN is referred to as a **lazy learner**, whereas linear regression would be an **eager learner**.

In the case of regression, we'll likely be taking the weighted average like in algorithm 2.1; in the case of classification, we'll likely instead have a "vote" and choose

[1] Game developers might already be somewhat familiar with the algorithm: quadtrees rely on similar principles to efficiently perform collision detection, pathfinding, and other spatially-sensitive calculations. The game world is dynamically divided into recursively-halved quadrilaterals to group closer objects together.

Algorithm 2.1: A naïve kNN learning algorithm. Both the number of neighbors, k, and the similarity metric, $d(\cdot)$, are assumed to be pre-defined.

```
Input: A series of training samples, $\mathcal{X} := \mathbf{x}_i \mapsto y_i$.
Input: The novel input, $\mathbf{x}'$.
Result: The predicted value of $y'$.
closest := {}
dists := $\{\infty, \ldots\}$
foreach $\mathbf{x}_i \in \mathcal{X}$ do
    foreach $j \in [1, k]$ do
        if $d(\mathbf{x}_i, \mathbf{x}') <$ dists[$j$] then
            closest[$j$] = $\mathbf{x}_i$
            dists[$j$] = $d(\mathbf{x}_i, \mathbf{x}')$
        end
    end
end
// Return the distance-weighed average of the $k$ closest neighbors.
return $\dfrac{\sum_{d_i \in \text{dists}} d_i \cdot \mathbf{x}_i}{\sum_{d_i \in \text{dists}} d_i}$
```

the label with plurality. We can also do more "sophisticated" tie-breaking: for regression, we can consider *all* data points that fall within a particular shortest distance (in other words, we consider the k shortest distances rather than the k closest points); for classification, we have more options. We could choose the label that occurs the most globally in $\mathcal{X}$, or randomly, or…

Biases

As always, it's important to discuss what sorts of problems a kNN representation of our data would cater towards.

Preference Bias Our belief of what makes a good hypothesis to explain the data heavily relies on **locality**—closer points (based on the domain-aware distance metric) are in fact similar—and **smoothness**—averaging neighbors makes sense and feature behavior smoothly transitions between values.

Notice that we've also been treating the features of our training sample vector $\mathbf{x}_i$ equally. The scales for the features may of course be different, but there is no notion of weight on a particular feature. This is critical: obviously, whether or not a restaurant is within your budget is far more important than whether the atmosphere inside fits your vibe (being the broke graduate students that we are). However, the kNN model has difficulty with making this differentiation.

In general, we are encountering the **curse of dimensionality** in machine learning: as the number of features (the dimensionality of a single $\mathbf{x}_i \in \mathcal{X}$) grows linearly, the amount of data we need to accurately generalize the grows **exponentially**. If we have a lot of features and we treat them all with equal importance, we're going to need a LOT of data to determine which ones are more relevant than others.

It's common to treat features as "hints" to the machine learning algorithm you're using, following the rationale that, "If I give it more features, it can approximate the model better since it has more information to work with;" however, this paradoxically makes it *more* difficult for the model, since now the feature space has grown and "importance" is *harder* rather than easier to determine.

Restriction Bias If you can somehow define a distance function that relates to feature points together, you can represent the data with a kNN.

Computational Learning Theory

> *People worry that computers will get too smart and take over the world, but the real problem is that they're too stupid and they've already taken over the world.*
>
> — Pedro Domingos

COMPUTATIONAL learning theory lets us define and understanding learning problems better. It's a powerful tool that can let us show that specific algorithms work for specific problems (like, for example, when a kNN learner would be best for a problem), and it also lets us show when a certain problem is fundamentally "hard."

The kinds of methods used in analysing learning questions are often analogous to the methods used in algorithm analysis in a more "traditional" computer science setting (think big-O notation). There, we talk about an algorithm's complexity in *time* and *space*; analogously, in machine learning problems, while we also care about time and space, we also care about *sample complexity*. Does a particular algorithm do well with small amounts of data? How does metrics like prediction accuracy grow with increased data?

Generally-speaking, inductive learning is *learning from examples*. There are many factors in a way such a learning problem is set up that can affect the resulting inductions; these include:

- the **number of samples** to learn from: it should come as no surprise that the amount of data we have can affect our inductions significantly;
- the **complexity of the hypothesis** class: the more difficult a concept or idea, the harder it may be for most algorithms to model; similarly, the risk of overfitting is high when you model a complex hypothesis;
- the **accuracy** to which the target concept is approximated: obviously, whether or not a model is "good" correlates directly with how well it performs on novel

data;

- how **samples are *presented***: until now, we've been training models on entire "batches" of data at once, but this isn't the only option; online learning (one at a time) is another alternative; and
- how **samples are *selected***: is random shuffling the best way to choose data to train on?

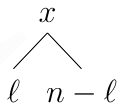
3.1 Learning to Learn: Interactions

We'll look at the last two items from the above list first because deep within them are some important subtleties.

The ways that training examples are selected to learn from can drastically affect the overall quantity of data necessary to learn something meaningful. There are a number of ways to approach the learner-teacher relationship:

1. *query-based*, where the learner "asks questions" that the teacher responds to. For example, "I have x; what's $c(x)$?"
2. *friendly*, where the teacher provides the example x and its resulting $c(x)$, selecting x to be a helpful way to learn.
3. *natural*, where nobody chooses and there's just some nature-provided distribution of samples.
4. *adversarial*, where the teacher provides *un*helpful $(x, c(x))$ pairings designed to make the learner fail (and thereby learn better when it figures out how *not* to fail–think "trick questions" on exams).
5. . . . and more!

So if we have a query-based learner, coming up with and asking questions, what should they ask? An effective question one whose answer gives the most information. In a binary world of yes-or-no questions with n possible hypotheses, the question always eliminates some ℓ number of possibilities:



A good question, then, ideally halves the space, so $\ell \approx n - \ell$; then it'll take

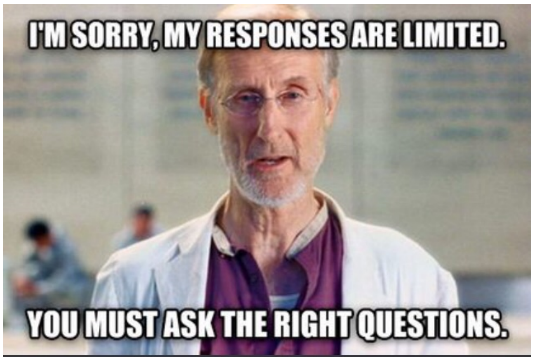

I'M SORRY, MY RESPONSES ARE LIMITED.
YOU MUST ASK THE RIGHT QUESTIONS.

Figure 3.1: From the movie *I, Robot*, a holographic version of a scientist appears posthumously to help Will Smith solve the riddle of the robot uprising.

$\log_2 |\mathcal{H}|$ questions overall to nail the answer and find h.

Teaching

If the teacher is providing friendly examples, wouldn't a very ambitious teacher—one that wants the learner to figure out the hypothesis $h \in \mathcal{H}$ as quickly as possible—simply tell the learner to "ask," "Is the hypothesis h?" Though effective, that's not a very realistic way to learn: the question-space is never "the entire set of possible questions in the universe." Teachers have a constrained space of questions they can suggest.

Example 3.1: A Good Teacher

Suppose we have some Boolean function that operates on some (possibly proper) subset of the bits x_1 through x_5. Our job is to determine the conjunction (logical-AND) of literals or negations that will fulfill the table of examples:

x_1	x_2	x_3	x_4	x_5	h
1	0	1	1	0	1
0	0	0	1	0	1
1	1	1	1	0	0
1	0	1	0	0	0
1	0	1	1	1	0

How should we tackle this? Well notice that in the first two rows, x_1 is flipped yet h stays the same. We can infer from this that x_1 is not relevant, and likewise for x_3. What's consistent across these rows? Well, both $x_2 = x_5 = 0$. Yet this is also the case in row 4, so this may be necessary but not sufficient. We also see that $x_4 = 1$ in both top rows, so a reasonable guess overall for the Boolean function in question is:

$$f(x_1 \cdots x_5) = \overline{x_2} \wedge x_4 \wedge \overline{x_5}$$

This holds for all of the rows.

Learning: Constrained Queries

Notice that in the above example, we can come up with f from the first two rows (we can think of these as "positive" examples), then use the next three rows to verify it (these being "negative" examples). This isn't because we're some sort of genius, but rather that we were given *very* good examples to learn from: each one gave us a lot of information. **This is exactly what it means to be a good teacher**: it can teach us a generic f with just $k + 2$ samples.

What if we weren't provided good examples and had to ask questions blindly? Since we don't really learn anything from $h = 0$, there are many possible fs that are extremely hard to guess. For a general k, it'll take 2^k *questions* in the worst case (consider when *exactly* one k-bit string gives $h = 1$).

Learning: Mistake Bounds

When the game is too hard, change the game. Instead of measuring how many samples we need, why not measure how many mistakes we make?

We'll modify the teacher-learner interaction thus: the learner is now given an input and has to guess the output. If it guesses wrong, it gets PUNISHED. This is an example of **online learning**, since the learner adjusts with each sample. Its algorithm is then:

1. To start, assume it's possible for each variable to be both positive and negated:

   $$h' = x_1 \land \overline{x_1} \land \ldots \land x_k \land \overline{x_k}$$

   Obviously this is logically impossible, but this lets us have a meaningful baseline for Step 3 to work.

2. Given an input, compute the output based on our h'. For a while (that is, until we guess wrong), we will just output "false."

3. If we're wrong, set positive variables (the x_is) that were 0 to absent and negative variables (the $\overline{x_i}$s) that were 1 to be absent.

4. Go to Step 2.

Basically whenever the learner guesses wrong, they will know to eliminate some variables. Even though the number of examples to learn may be the same in the worst case (2^k), we will now never make more than $k + 1$ *mistakes*. A good teacher that *knows* that its learner has this algorithm can actually teach h with $k + 1$ samples, teaching it one bit every time.

3.2 Space Complexity

As we've already established, data is king in machine learning. In this section we'll derive a theoretical way to approximate space complexity: how much data do we need to solve a problem to a particular degree of certainty?

First, we'll need to rigorously define some terminology:

- a **training set**: $S \subseteq \mathcal{X}$ is made up of some samples x
- $\mathcal{H}$ is the hypothesis space
- the ***true* hypothesis** or **concept**: $c \in \mathcal{H}$ is the thing we want to learn, while the ***candidate* hypothesis**: $h \in \mathcal{H}$ is what the learner is currently considering

- a **consistent learner**: one that produces the correct result for all of the training samples:

$$\forall x \in S : \quad c(x) = h(x)$$

3.2.1 Version Spaces

Given the above, the **version space** is all of the possible hypotheses that are consistent with the data; in other words, the only hypotheses worth considering[1] at some point in time given the training data:

$$\text{VS}(S) = \{h : c(x) = h(x) \mid \forall x \in S, h \in \mathcal{H}\}$$

Example 3.2: **Version Spaces**

Given the target concept of XOR (left) and some training data (right):

x_1	x_2	$c(x)$
0	0	0
0	1	1
1	0	1
1	1	0

x_1	x_2	$c(x)$
0	0	0
0	1	1
1	1	$\boxed{?}$

and the hypotheses:

$$\mathcal{H} = \{x_1, \overline{x_1}, x_2, \overline{x_2}, T, F, \texttt{OR}, \texttt{AND}, \texttt{XOR}, \texttt{EQUIV}\}$$

which of the hypotheses are in the version space?

Well, which of the hypotheses would be consistent with the training samples?

$$\left\{x_1, \overline{x_1}, \boxed{x_2}, \overline{x_2}, T, F, \boxed{\texttt{OR}}, \texttt{AND}, \boxed{\texttt{XOR}}, \texttt{EQUIV}\right\}$$

3.2.2 Error

Given a candidate hypothesis h, how do we evaluate how good it is? We define the **training error** as the fraction of training examples misclassified by h, while the **true error** is the fraction of examples that *would be* misclassified on the sample drawn from a distribution $\mathcal{D}$:

$$\text{error}_{\mathcal{D}}(h) = \Pr_{x \sim \mathcal{D}}[c(x) \neq h(x)] \tag{3.1}$$

This lets us minimize the punishment that comes from getting rare examples wrong.

[1] Author's note: this is such an unfortunate term and is a classic example of the machine learning field intentionally gatekeeping newcomers with obscure jargon. Wouldn't it make much more sense to simply call this the *viable* hypotheses?

So if there's a very obscure case that comes up once in a blue moon, it should be treated differently than getting a commonly-occurring case incorrect.

3.2.3 PAC Learning

First, some more definitions:

- C is the **concept class**
- L is the **learner**
- $\mathcal{D}$ is the probability distribution over the inputs
- the **error goal** is $0 \leq \varepsilon \leq 1/2$, while the **certainty goal** is $0 \leq \delta \leq 1/2$.

Then, we say C is **PAC-learnable**[2] by L using $\mathcal{H}$ **if and only if**: L will, with probability $1 - \delta$, output a hypothesis $h \in \mathcal{H}$ such that $\text{error}_{\mathcal{D}}(h) \leq \varepsilon$ with **polynomial** *time* and *sample* complexity in $1/\varepsilon, 1/\delta, n$. That is, finding h such that

$$\Pr\left[\text{error}_{\mathcal{D}}(h) \leq \varepsilon\right] = 1 - \delta$$

is achievable within polynomial time. **To rephrase this in English**: a concept is PAC-learnable if it can be learned to a reasonable degree of correctness within a reasonable amount of time.

(couldn't we have just said that straight up instead of needing 4 pages of terminology?)

3.2.4 Epsilon Exhaustion

A version space is considered **ε-exhausted** if and only if all of its hypotheses have low error. Formally, iff:

$$\forall h \in \text{VS}(S): \quad \text{error}_{\mathcal{D}}(h) \leq \varepsilon$$

Remember that ε varies, so how do we find the smallest-possible ε for a particular version space? Apply the **Haussler theorem**: it allows us to bound the true error (3.1) in terms of the number of necessary data samples.

Lets consider the hypotheses $h_{1..k}$ that have high true error:

$$\text{error}_{\mathcal{D}}(h_1, h_2, \ldots, h_k \in \mathcal{H}) > \varepsilon$$

how much data do we need to "knock out" these hypotheses from the version space?

The probability of being right for any of these hypotheses is then:

$$\Pr_{x \sim \mathcal{D}}\left[h_i(x) = c(x)\right] \leq 1 - \varepsilon$$

[2] PAC—**p**robably **a**pproximately **c**orrect, where $1 - \delta$ defines the "probably," ε defines the "approximately," and $\text{error}_{\mathcal{D}}(h) = 0$ defines the "correct."

Being consistent on m samples then exponentiates this for a specific h_i:

$$\begin{aligned}
\Pr\left[\text{all } h_i(x) = c(x) \text{ on } m \text{ samples}\right] \leq \;& \Pr\left[h_1(x) = c(x) \text{ on } m \text{ samples}\right] \cdot \\
& \Pr\left[h_2(x) = c(x) \text{ on } m \text{ samples}\right] \cdot \\
& \ldots \\
& \Pr\left[h_k(x) = c(x) \text{ on } m \text{ samples}\right] \\
\leq \;& (1-\varepsilon)^m
\end{aligned}$$

but *at least one* staying consistent is:

$$\begin{aligned}
\Pr\left[\text{any } h_i(x) = c(x) \text{ on } m \text{ samples}\right] \leq \;& \Pr\left[h_1(x) = c(x) \text{ on } m \text{ samples}\right] + \\
& \Pr\left[h_2(x) = c(x) \text{ on } m \text{ samples}\right] + \\
& \ldots \cdot \\
& \Pr\left[h_k(x) = c(x) \text{ on } m \text{ samples}\right] \\
\leq \;& k(1-\varepsilon)^m \\
\leq \;& |\mathcal{H}|\,(1-\varepsilon)^m
\end{aligned}$$

Now we use an interesting truth (shown without proof here, just refer to the lectures or a real textbook lol): $-\varepsilon \geq \ln(1-\varepsilon)$. Thus,

$$|\mathcal{H}|\,(1-\varepsilon)^m \leq \boxed{|\mathcal{H}|\,e^{-\varepsilon m}} \leq \delta$$

This gives us **an upper bound that the version space is *not* ε-exhausted after m samples**, and this is related to our certainty goal δ. To solve this inequality in terms of m, we get:

$$\begin{aligned}
|\mathcal{H}|\,e^{-\varepsilon m} &\leq \delta && (3.2) \\
\ln|\mathcal{H}| - \varepsilon m &\leq \ln\delta && \text{logarithm product rule} \\
m &\geq \frac{1}{\varepsilon}\left(\ln|\mathcal{H}| + \ln\frac{1}{\delta}\right) && \text{flip inequality when dividing by } -\varepsilon
\end{aligned}$$

Notice that this upper bound is polynomial in all of the terms we need for a problem to be PAC-learnable.

Example 3.3: PAC-Learnability

Given a 10-bit input x, suppose we have a hypothesis space that guesses one of those bits being set. That is,

$$\mathcal{H} = \{h_i(x) = x_i\}$$

Given that $\mathcal{D}$ is a uniform distribution, $\varepsilon = 0.1, \delta = 0.2$, how many samples do we need to PAC learn the hypothesis set?

Well, refer to the upper bound from before:

$$\begin{aligned} m &\geq \frac{1}{\varepsilon}\left(\ln|\mathcal{H}| + \ln\frac{1}{\delta}\right) && (3.3)\\ &\geq \frac{1}{0.1}\left(\ln 10 + \ln\frac{1}{0.2}\right) && (3.4)\\ &\geq 10(\ln 10 + \ln 5) \approx 39.12 && (3.5)\\ &\boxed{\geq 40} && (3.6) \end{aligned}$$

This is pretty good: it's only 4% of the total 2^{10}-element input space. Also notice that $\mathcal{D}$ was irrelevant.

3.3 Infinite Hypothesis Spaces

Consider the ~~flaw~~ limitation of the Haussler theorem presented above:

$$m \geq \frac{1}{\varepsilon}\left(\ln|\mathcal{H}| + \ln\frac{1}{\delta}\right) \tag{3.7}$$

if $|\mathcal{H}| = \infty$, things kinda... blow up. We need to deal with this somehow in our theoretical; in fact, all of the concrete, practical algorithms we learned in the previous chapters have handled this snafu just fine. In fact, linear separators, neural networks, and decision trees with continuous inputs all have infinite hypothesis spaces.[3]

3.3.1 Intuition

Let's work through a concrete example to build up some intuition about how to tackle this problem. Suppose we're given a simple input space, and a simple set of infinite hypotheses: is x bigger than some number θ?

$$\begin{gathered} X = \{1, 2, 3, 4, \ldots, 10\} \\ \mathcal{H} : \{h(x) = x \geq \theta \mid \forall\theta \in \mathbb{R}\} \end{gathered}$$

Obviously since θ is a real number, $|\mathcal{H}| = \infty$. However, notice that many of those hypotheses are completely meaningless. More specifically, any choice of $\theta > 10$ gives the same result as any $\theta < 1$. Similarly, since $X \subset \mathbb{N}$, only integer θs really make sense. This gives us a notion of a "semantic" hypothesis space: **hypotheses that are meaningfully different**, as opposed to all of the hypotheses in the conceivable universe that we could write down...

TL;DR: We want to differentiate between hypotheses that matter (of which there are few) from the hypotheses that don't (of which there are infinite).

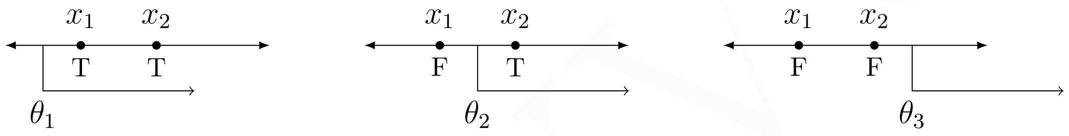
[3] Obviously, there are an infinite number of lines, and any model with continuous inputs has an infinite number of ways to change up the input.

3.3.2 Vapnik-Chervonenkis Dimension

Let's turn this intuition into something more formal. Our definition of "meaningful" will be something far more specific and elegant:

> What is the largest set of inputs that the hypothesis class can $\overbrace{\text{label in all possible ways}}^{\text{also termed “shattering”}}$?

Example To continue with our previous example, the answer is simply one. The hypotheses are all binary, true-or-false questions, so any one input can be labeled in all of the two possible ways. However, since we only consider $x \geq \theta$, we can't properly map a pair of inputs. Think of θ as being the "separator" between some training data $S = \{x_1, x_2\}$; then, slide it along:



It's impossible to capture a case where $x_1 = T, x_2 = F$. Thus, despite being infinite, the hypothesis space $\mathcal{H}$ cannot express very much; it's a weak space.

In general, this is called the **VC dimension** of $\mathcal{H}$; it allows us to relate the amount of data we need to learn to describe a hypothesis space.

Example 3.4: Measuring the VC Dimension

Suppose we're given an input space of all real numbers and a hypothesis space that can tell us whether or not an input value is within a particular interval:

$$
\begin{aligned}
X &= \mathbb{R} \\
\mathcal{H} &= \{h(x) = (x \in [a, b])\}
\end{aligned}
$$

parameterized by $a, b \in \mathbb{R}$

What is the largest set of inputs that we can shatter?

Answer: two

We solve this methodically. Confirming that we can shatter one input is easy: the interval either goes around or outside of the input. For two, we have four combinations to confirm: true-true can be done by wrapping the inputs just like in the single-input case, true-false is done by only wrapping the former, vice-versa for false-true, and false-false is done by wrapping something else. What about three? Consider, specifically, the true-false-true case. This is impossible to capture with a single interval, and thus we can't shatter it.

To show a lower bound for a VC dimension, all you need to do is find a single example of inputs for which all labeling combinations can be done.

Practical VC Dimensions

What's the VC dimension for something more practical in machine learning, like a linear separator? That is, for

$$\begin{aligned} X &= \mathbb{R}^2 \\ \mathcal{H} &= \{h(x) = \mathbf{w}^{\mathrm{T}}\mathbf{x} \geq \theta\} \end{aligned}$$

The VC dimension is three. The use of the second (Euclidean) dimension lets us solve the problem from the above example, but four inputs (one inside of the convex hull of the other three, for example) are impossible to shatter.

This might lead to an inductive hypothesis: with one parameter we had one VC dimension; with two, we had two; with three, we had three... can we expect to find a VC dimension of four when we decide to work with 3D space?

In fact, this is precisely the case: **for any n-dimensional hyperplane (or hypothesis class), the VC dimension will be $n + 1$.**

Sample Complexity

The whole divergence into VC dimensions was to find a way to deal with infinite hypothesis spaces in the Haussler theorem (3.7) so that we can make estimates about how many m samples we need to find good hypotheses.

The derivation (which I'm sure is long and arduous) leads us to the following:

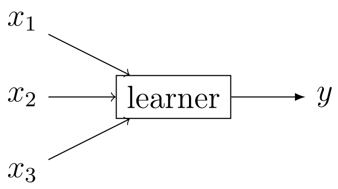
$$m \geq \frac{1}{\varepsilon}\left(8 \cdot \mathrm{VC}(\mathcal{H}) \cdot \log_2 \frac{13}{\varepsilon} + 4 \log_2 \frac{2}{\delta}\right) \tag{3.8}$$

Can we determine the VC dimensions of *finite* hypothesis spaces, too? Turns out, finding an upper bound isn't hard. If $d = \mathrm{VC}\mathcal{H}$, then there are 2^d distinct concepts (each picking a different $h \in \mathcal{H}$). Thus, since $2^d \leq |\mathcal{H}|$, we know that $\boxed{d \leq \log_2 |\mathcal{H}|}$.

Theorem 3.1. *A hypothesis space $\mathcal{H}$ is PAC-learnable if and* only *if its VC dimension is finite.*

3.4 Information Theory

This brief foray into a completely separate field will give us a bit of knowledge that is useful in understanding our machine learning models better. Our algorithms fit a pretty typical abstract model:



We want to be able to answer questions like:

- Which of our x_is will give us the *most information* about the output?
- Are these input feature vectors similar? We'll call this metric **mutual information**.
- Does this feature have any information? This is called **entropy**.

The key first step is simply quantifying this idea of "information."

Suppose we want to transmit the result of 10 coin flips. However, we're comparing a fair coin ($^1/_2$ probability of each result) and a biased coin whose sides are identical. Suppose our flip results were as follows:

fair: `HTHHTHTTHT`
unfair: `HHHHHHHHHH`

Which of these messages has more information? Many answers may make sense intuitively: perhaps we sent one bit per flip regardless; perhaps all we're sending is the flip ratio; perhaps we can compress the unfair flips into a single bit; etc. In actuality, we need 10 bits for the first message and *zero* for the second: the output is completely predictable without extra information.

3.4.1 Entropy: Information Certainty

This notion that certain messages are harder to transmit than others leads to **entropy**: the more unpredictable a particular piece of data, the more information needs to be transmitted to explain it. Thus, sending `HHHHHHHHHH` actually transmits no information.

Let's work through something a little more complex. Suppose we have a four-letter alphabet: $L = \{A, B, C, D\}$. This set can clearly be represented with 2 bits per letter: $\{A = 00, B = 01, C = 10, D = 11\}$.

However, suppose the letters don't occur with equal probability: a message with an A occurs 50% of the time. Specifically, we have the ratios on the right. Can we get away with a different bit-representation?

Letter	Frequency
A	50%
B	12.5%
C	12.5%
D	25%

Since A occurs half the time, we can let $A = 0$. Then, we can represent half of the possible occurrences of letters with a single bit. However, now we need three bits for B and C: if $D = 10$, then $B = 110$ and $C = 111$. We're still saving information, though, because these cases occur less frequently.

How much are we saving? To answer that, we'll need to use *math*. What's the **expected** message size of a single letter? It's the product of a probability's letter and size:

$$\mathbb{E} = \sum_i \left(|\ell_i| \cdot \Pr\left[\ell_i\right]\right) = 0.5 \cdot 1 + 0.125 \cdot 3 + 0.125 \cdot 3 + 0.25 \cdot 2 = 1.75 \text{ average bits}$$

In general, the size of a piece of information x is $\log_2 \frac{1}{x}$. Thus, entropy in general is expressed as:

$$h(S) = -\sum_{s \in S} \Pr\left[s\right] \cdot \log_2 \Pr\left[s\right] \tag{3.9}$$

3.4.2 Joint Entropy: Mutual Information

Will there be thunder today? If you're sitting alone in a windowless room, it's hard to say, but if I told you it's raining outside, you can make a better guess. Thus, there is a relationship between variables: knowing one helps you learn something about another.

In information theory, this is represented by **joint entropy**:

$$h(X, Y) = -\sum_{x \in X, y \in Y} \Pr\left[x, y\right] \log_2 \Pr\left[x, y\right]$$

There is also the idea of **conditional entropy**, where the entropy for a variable changes based on another:

$$h(Y \mid x) = -\sum_{y \in Y} \Pr\left[x, y\right] \log_2 \Pr\left[Y \,|\, x\right]$$

Obviously, if two variables are independent of each other, $h(Y \mid x) = h(Y)$, and $H(X, Y) = H(X) + H(Y)$.

We need to differentiate between two cases: perhaps x tells us a lot about Y, then Y's conditional entropy will be small; however, it's possible that Y's entropy is small to begin with. To solve this, we introduce the formula formutual information:

$$I(x, Y) = h(Y) - h(Y \mid x) \tag{3.10}$$

3.4.3 Kullback-Leibler Divergence

This metric measures the difference between any two distributions; mutual information is a special case of **KL-divergence**. It's given by:

$$D_{\text{KL}}(p \parallel q) = \int \Pr[x] \log_2 \frac{\Pr[p]\, x}{\Pr[q]\, x}\, dx \tag{3.11}$$

Bayesian Learning

> *The combination of Bayes and Markov Chain Monte Carlo has been called "arguably the most powerful mechanism ever created for processing data and knowledge." Almost instantaneously MCMC and Gibbs sampling changed statisticians' entire method of attacking problems. In the words of Thomas Kuhn, it was a paradigm shift. MCMC solved real problems, used computer algorithms instead of theorems, and led statisticians and scientists into a worked where "exact" meant "simulated" and repetitive computer operations replaced mathematical equations. It was a quantum leap in statistics.*
>
> — Sharon Bertsch McGrayne, *The Theory That Would Not Die*

One of the most important, fundamental rules of statistics is **Bayes' rule**: it relates the conditional probability of one event to the condition itself:

$$\Pr[x \mid y] = \frac{\Pr[y \mid x] \cdot \Pr[x]}{\Pr[y]} \tag{4.1}$$

It also comes with some bonus corollaries:

$$\begin{aligned}
\Pr[x, y] &= \Pr[a \mid b] \cdot \Pr[b] \\
\Pr[x, y] &= \Pr[b \mid a] \cdot \Pr[a] && \text{since order doesn't matter}
\end{aligned}$$

These simple rule is what powers Bayesian learning. Our learning algorithms aim to learn the "best" hypothesis given data and some domain knowledge. If we reframe this as being the *most probable* hypothesis, now we're cooking with Bayes, since this can be expressed as:

$$\operatorname*{arg\,max}_{h \in \mathcal{H}} \Pr[h \mid D]$$

where D is our given data. Let's apply Bayes' rule to our novel view of the world:

$$\Pr[h \mid D] = \frac{\Pr[D \mid h] \cdot \Pr[h]}{\Pr[D]} \tag{4.2}$$

But what does $\Pr[D]$—the probability of the data—really mean? Well D is our set of training examples: $D = \{(x_1, d_1), \ldots, (x_n, d_n)\}$ and they were chosen/sampled/given (recall the variations we discussed in section 3.1) from some probability distribution.

What about $\Pr[D \mid h]$, this strange notion of data occurring *given* our hypothesis? This is the **likelihood**—for a particular label, how likely is our hypothesis to output it? The beauty of this formulation is that it's a lot easier to figure out $\Pr[D \mid h]$ than the original quantity.

Finally, what does $\Pr[h]$—the prior probability of the hypothesis h—mean? This represents our **domain knowledge**: hypotheses that represent the world or problem space better will be likelier to occur.

Example 4.1: **Likelihood**

Suppose we have the following simple hypothesis:

$$h(x) = \begin{cases} 1 & \text{if } x \geq 10 \\ 0 & \text{otherwise} \end{cases}$$

If we have $x = 7$, what's the *likelihood* that $h(x) = 1$? Obviously zero, since $h(7)$ will never output 1. Thus, $\Pr[\{7\} \mid h] = 0$.

When working with probabilities, it's crucial to remember the paradox of priors. Consider a man named Dwight diagnosed with *dental hydroplosion*—an extremely rare disease that occurs in 8 of 1000 people—based on a test with a 98% correct positive rate and a 97% correct negative rate. Is it actually plausible that his teeth will spontaneously explode and drip down his esophagus in his sleep?

Well, apply Bayes' rule and see:

$$\begin{aligned}
\Pr\left[\text{has it} \,\middle|\, \substack{\text{test says}\\\text{has it}}\right] &= \frac{\Pr\left[\substack{\text{test says}\\\text{has it}} \,\middle|\, \text{has it}\right] \cdot \Pr[\text{has it}]}{\Pr\left[\substack{\text{test says}\\\text{has it}}\right]} \\
&= \frac{0.98 * 0.008}{\underbrace{0.98 * 0.008}_{\Pr\left[\substack{\text{test says}\\\text{has it}}\right] \cdot \Pr[\text{has it}]} + \underbrace{0.03 * (1 - 0.008)}_{\Pr\left[\substack{\text{test says}\\\text{has it}}\right] \cdot \Pr[\text{not has it}]}} \\
&\approx 20\%
\end{aligned}$$

In other words, Dwight only has a 20% chance of actually having dental hydroplosion despite the fact that the test has a 98% accuracy! The disease is so rare that the probability of actually being infected dominates the test accuracy itself. The prior (domain knowledge) of a random person having the disease (0.8%) is critical here.

4.1 Bayesian Learning

The trip to the doctor has given us a bit of an algorithm: for each $h \in \mathcal{H}$, calculate $\Pr[h \mid D] \propto \Pr[D \mid h] \cdot \Pr[h]$. Then, just output the $\arg\max_h$.[1]

4.1.1 Finding the Best Hypothesis

Sometimes it's quite hard to know $\Pr[h]$, so there are two possible versions:

- The **maximum *a posteriori*** hypothesis (or MAP):

$$h_{\text{map}} = \arg\max_h \Pr[h \mid D] = \arg\max_h \Pr[D \mid h] \Pr[h]$$

- The **maximum likelihood** hypothesis (or ML):

$$h_{\text{ml}} = \arg\max_h \Pr[D \mid h]$$

The latter just assumes a uniform probability: all hypotheses are equally-likely to occur. Unfortunately, enumerating every hypothesis is not practical, but fortunately it gives us a theoretically-optimal approach.

Let's put theory into action.[2] Suppose we're given a generic set of labeled examples: $\{(x_i, d_i\}$ where each label comes from a function and has some normally-distributed noise, so:

$$\begin{aligned} d_i &= f(x_i) + \varepsilon_i \\ \varepsilon_i &\sim \mathcal{N}\left(0, \sigma^2\right) \end{aligned}$$

The maximum likelihood is clearly defined by the Gaussian: the further from the mean an element is, the less likely it is to occur.

$$h_{\text{ml}} = \arg\max_h \Pr[D \mid h]$$

[1] We dropped the denominator from (4.1) because it's just a normalization term—the resulting probabilities will still be proportional and thus equally comparable.

[2] At this point, the lectures go through a length derivation for two specific cases, then actually apply it in general. We are skipping the former here because it's Tuesday, the project is due on Sunday, and I have like a dozen more lectures to get through before I can properly start.

$$
\begin{aligned}
&= \arg\max_h \prod_i \Pr\left[d_i \,|\, h\right] \\
&= \arg\max_h \prod_i \frac{1}{-\sqrt{2\pi\sigma^2}} e^{\frac{(d_i - h(x_i))^2}{2\sigma^2}} \\
&= \arg\max_h \prod_i e^{\frac{(d_i - h(x_i))^2}{-2\sigma^2}} && \text{the constant has no effect on the } \arg\max \\
\ln h_{\text{ml}} &= \arg\max_h \sum_i -\frac{1}{2} \cdot \frac{(d_i - h(x_i))^2}{\sigma^2} && \ln(\cdot) \text{ both sides, then the product of logs is the sum of its components} \\
&= \arg\max_h -\sum_i (d_i - h(x_i))^2 && \text{drop constants like before} \\
&= \arg\min_h \sum_i (d_i - h(x_i))^2 && -\max(\cdot) = \min(\cdot)
\end{aligned}
$$

But this final expression is simply the sum of squared errors! Thus, the log of the maximum likelihood is simply expressed as:

$$\ln h_{\text{ml}} = \arg\min_h \text{SSD}(d, h) \tag{4.3}$$

Thus, our intuition about minimizing the SSD in the algorithms we covered earlier is correct: gradient descent, linear regression, etc. are all the way to go to find a good hypothesis. Bayesian learning just confirmed it!

Note our foundational assumptions, though: our dataset is built upon a true underlying function and is perturbed by Gaussian noise. If this assumption doesn't actually hold for our problem, then minimizing the SSD will not give the optimal results that we're looking for.

4.1.2 Finding the Best Label

Consider the following table that tells us some results about the hypotheses given the data and what the hypotheses say about some input x:

$h(x)$	$\Pr\left[h \,\vert\, D\right]$
$h_1 : +$	0.4
$h_2 : -$	0.3
$h_3 : -$	0.3

What should we label x, given our hypotheses' outputs? Obviously, h_1 is the likeliest, and it votes $+$. However, h_2 and h_3's *combined* likelihood is higher than h_1 and they would vote $-$. What's the best call? As intuition may suggest, $-$ is the better vote.

The important takeaway is that the best hypothesis does not always provide the best label. However, allowing all hypotheses to (do a *weighted*) vote leads to **Bayes'**

optimal classifier. This is another important result: on average, you *cannot* do better than a weighted vote from all of the hypotheses.

4.2 Bayesian Inference

First, let's review joint distributions. Given the following toy example, which tells us the likelihood of various weather patterns in Atlanta on a random summer afternoon,

storm	lightning	$\Pr[\cdot]$
T	T	0.25
T	F	0.40
F	T	0.05
F	F	0.30

what's the probability of there not being a storm? Simple enough: $0.25 + 0.05 = 0.30$.

What about the probability of there being lightning given that there's a storm going on? Well, we need to renormalize the distribution of *just* the "is storming" cases:

$$\Pr[\text{lighting} \mid \text{storm}] = \frac{0.25}{0.25 + 0.40} \approx 0.38$$

If we added another variable (like, "Is it thundering?"), our table would double in size. For n Boolean variables, we have a massive 2^n-entry joint distribution. We can represent it a different, more-efficient that instead takes $2n$ entries via **factoring**.

4.2.1 Bayesian Networks

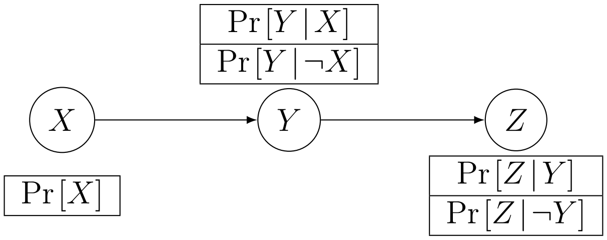
Recall the definition of conditional independence between two variables:

> *X is **conditionally independent** of Y given Z if the probability distribution governing X is independent of the value of Y when given the value of Z:*
>
> $$\begin{aligned} &\forall x \in X, y \in Y, z \in Z : \\ &\quad Pr[X = x \mid Y = y, Z = z] = Pr[X = x \mid Z = z] \end{aligned}$$

More compactly, we typically write $\Pr[X \mid Y, Z] = \Pr[X \mid Z]$.[3]

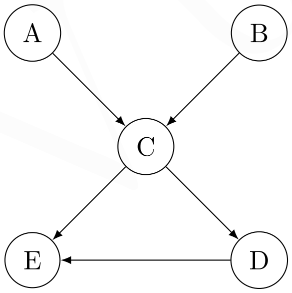
If we have a system of three variables (say X, Y, Z) in which two (say X, Z) are conditionally independent, we can arrange them in a **Bayesian network** (also called a **graphical model**) that represents this relationship as well as some tables—known as conditional probability tables (CPTs)—that represent the probabilities at each step:

[3] Don't forget about the definition of two variables being **independent**: $\Pr[X, Y] = \Pr[X] \cdot \Pr[Y]$ as well as the **chain rule** that comes out of Bayes' rule: $\Pr[X, Y] = \Pr[X \mid Y] \cdot \Pr[Y]$.



Optimizing CPTs Now, given an arbitrary Bayesian network:

$$\begin{aligned} A &\sim \Pr[A] \\ B &\sim \Pr[B] \\ C &\sim \Pr[C \mid A, B] \\ D &\sim \Pr[D \mid B, C] \\ E &\sim \Pr[E \mid C, D] \end{aligned}$$



to get a sample over the entire joint distribution (that is, to sample some $X \sim \Pr[A, B, C, D, E]$), we need to sample from the Bayesian network in its **topological order** (an easily-achievable configuration on an acyclic digraph), because:

$$\Pr[A, B, C, D, E] = \Pr[A] \cdot \Pr[B] \cdot \Pr[C \mid A, B] \cdot \Pr[D \mid B, C] \cdot \Pr[E \mid C, D]$$

Notice the space savings of this representation: the full joint distribution (if A through E were Boolean variables) requires $2^5 - 1 = 31$ probability specifications to recover, whereas under the specific conditional independencies we described, only $1 + 1 + 4 + 4 + 4 = \mathbf{14}$ probabilities are needed. If they were all completely independent, we'd only need 5 probabilities (the product of the unconditionals).

4.2.2 Making Inferences

Sampling Why do we even care about probability distributions? Well, if the distributions model some complex, real-world processes, *simulating* them accurately can let us learn things about it. Furthermore, we can use the distributions to make *predictions* or approximate *inferences* based on partial information.

We say "approximate" inferences because they are far easier to calculate; finding *exact* inferences efficiently would mean $P = NP$.

In both of these cases, samples from the distribution (rather than the distribution itself, since that's often all we can have) let us model the underlying truth and do those things.

Inferencing Rules Let's make some inferences. For clarity, though, let's first reiterate the rules we're going to be using:

- **total probability**: the probability of a random variable X is the sum of the probabilities of the individual events that compose it (for example, if X is "weather," then it's the sum of "rainy," "sunny," "cloudy," etc.):

$$\Pr[X] = \sum_{x \in X} \Pr[X = x]$$

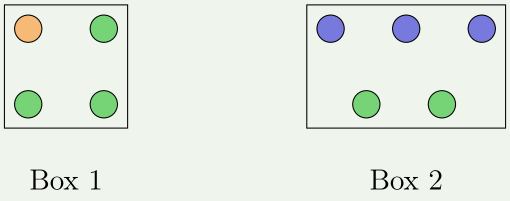
- **marginalization**: if given another random variable, Y, and their joint probabilities we can find the probability of *just* X by summing all the joint probabilities regardless of Y's value:

$$\Pr[X] = \sum_{y \in Y} \Pr[X, Y = y]$$

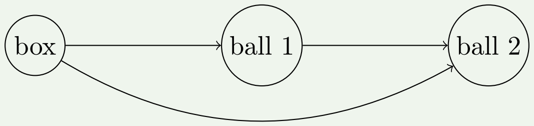
- **chain rule**: $\Pr[X, Y] = \Pr[X] \cdot \Pr[Y \mid X] = \Pr[Y] \cdot \Pr[X \mid Y]$
- **Bayes' rule**: $\Pr[Y \mid X] = \dfrac{\Pr[X \mid Y] \cdot \Pr[Y]}{\Pr[X]}$

Example 4.2: Inference By Hand

Suppose we're given two boxes with colored balls inside of them (this a classic probability example; I have no idea why statisticians love boxes with balls, or bags of marbles, or containers of jelly beans, or . . .)



For every sample, we first choose a box uniformally at random then fish out a ball. The Bayesian network of this sampling process is:



What's the probability of drawing a green ball then a blue ball?

$$\Pr[2 = \text{blue} \mid 1 = \text{green}] = \boxed{?}$$

Answer: $^6/_{23} \approx 0.26$

The Bayes net itself gives us an insight on how we should compute this! By combining the marginalization and chain rules, we get

$$\begin{aligned}
\Pr[2 = \text{blue} \,|\, 1 = \text{green}] &= \Pr[2 = \text{blue} \,|\, 1 = \text{green}, \text{box}] \\
&= \Pr[2 = \text{blue} \,|\, 1 = \text{green}, \text{box} = 1] \cdot \Pr[\text{box} = 1 \,|\, \text{green}] + \\
&\quad \Pr[2 = \text{blue} \,|\, 1 = \text{green}, \text{box} = 2] \cdot \Pr[\text{box} = 2 \,|\, \text{green}] \\
&= 0 \cdot \Pr[\text{box} = 1 \,|\, 1 = \text{green}] + \frac{3}{4} \cdot \Pr[\text{box} = 2 \,|\, 1 = \text{green}]
\end{aligned}$$

To find these latter values, we apply Bayes' rule:

$$\begin{aligned}
\Pr[\text{box} = 1 \,|\, 1 = \text{green}] &= \frac{\Pr[1 = \text{green} \,|\, \text{box} = 1] \Pr[\text{box} = 1]}{\Pr[1 = \text{green}]} \\
&= \frac{\frac{3}{4} \cdot \frac{1}{2}}{\Pr[1 = \text{green}]} \\
&= \frac{^3/_8}{\Pr[1 = \text{green}]}
\end{aligned}$$

and similarly,

$$\begin{aligned}
\Pr[\text{box} = 2 \,|\, 1 = \text{green}] &= \frac{\Pr[1 = \text{green} \,|\, \text{box} = 2] \Pr[\text{box} = 2]}{\Pr[1 = \text{green}]} \\
&= \frac{\frac{2}{5} \cdot \frac{1}{2}}{\Pr[1 = \text{green}]} \\
&= \frac{^1/_5}{\Pr[1 = \text{green}]}
\end{aligned}$$

Since we're combining the two values, $\Pr[1 = \text{green}]$ will actually disappear entirely if we normalize, so we don't need to calculate it separately:

$$\frac{\frac{1}{5}}{\frac{1}{5} + \frac{3}{8}} = \frac{\frac{8}{40}}{\frac{8}{40} + \frac{15}{40}} = \frac{8}{23} \qquad\qquad \frac{\frac{3}{8}}{\frac{1}{5} + \frac{3}{8}} = \frac{\frac{15}{40}}{\frac{8}{40} + \frac{15}{40}} = \frac{15}{23}$$

Finally, we can plug this back into our original expansion:

$$\begin{aligned}
\Pr[2 = \text{blue} \,|\, 1 = \text{green}] &= 0 \cdot \Pr[\text{box} = 1 \,|\, 1 = \text{green}] + \frac{3}{4} \cdot \Pr[\text{box} = 2 \,|\, 1 = \text{green}] \\
&= 0 \cdot \frac{15}{23} + \frac{3}{4} \cdot \frac{8}{23} \\
&= \boxed{\frac{6}{23} \approx 0.26}
\end{aligned}$$

Solving that example was ***miserable***, but we followed a pretty clear process based on the Bayesian network. Luckily for us, that means it should be possible to automate.

4.2.3 Naïve Bayes

Suppose we cut down on the complexity of allowable relationships, only allowing edges between immediate family; that is, parents are connected directly to children, and that's it. The beauty of the structure of such a **naïve Bayesian network** is that it breaks down larger conditional probabilities into products of simpler ones.

For example, given the Bayesian network and CPTs in Figure 4.1, can we compute the probability of a particular message being spam given its contents?

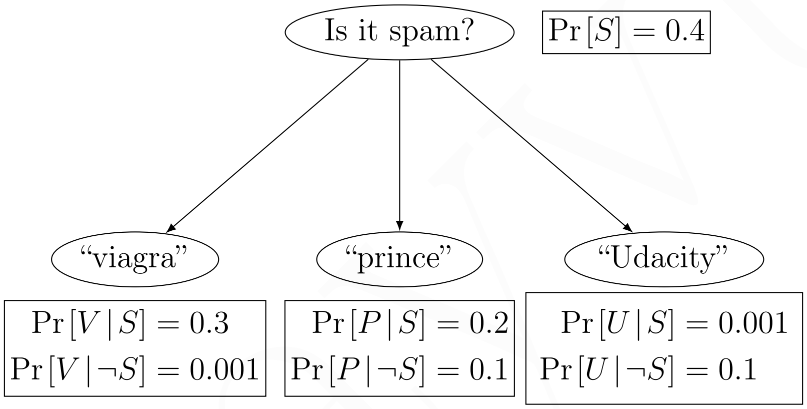


Figure 4.1: An example *naïve* Bayesian network for spam email.

$$\begin{aligned}
\Pr\left[\text{spam?} \,\middle|\, \substack{\text{contains}\\\text{``viagra''}}, \neg\substack{\text{contains}\\\text{``prince''}}, \neg\substack{\text{contains}\\\text{``Udacity''}}\right] = \\
&\propto \Pr\left[\substack{\text{contains}\\\text{``viagra''}}, \neg\substack{\text{contains}\\\text{``prince''}}, \neg\substack{\text{contains}\\\text{``Udacity''}} \,\middle|\, \text{spam?}\right] \cdot \Pr[\text{spam?}] && \text{Bayes' rule} \\
&\propto \Pr\left[\substack{\text{contains}\\\text{``viagra''}} \,\middle|\, \text{spam?}\right] \cdot \Pr\left[\neg\substack{\text{contains}\\\text{``prince''}} \,\middle|\, \text{spam?}\right] \cdot \Pr\left[\neg\substack{\text{contains}\\\text{``Udacity''}} \,\middle|\, \text{spam?}\right] \cdot \Pr[\text{spam?}] \\
&\propto 0.3 \cdot 0.8 \cdot 0.9999 \cdot 0.4 \\
&= \frac{0.096}{z} && z \text{ is the normalization factor}
\end{aligned}$$

In general, when we have this format of a single parent V and its many children, $a_1, a_2, \ldots, a_n$, then the probability is solveable as:

$$\Pr[V \mid a_1, a_2, \ldots, a_n] = \frac{\Pr[V]}{z} \cdot \prod_i^n \Pr[a_i \mid V]$$

This beautiful relationship lets us do classification (yay, we're finally talking about machine learning again!): given the features / attributes (the children), we can make inferences about the parent! The maximum *a posteriori* class (that is, the best class label) is then simply the best $v \in V$:

$$\underset{v \in V}{\arg\max} \prod_{i}^{n} \Pr\left[a_i \,|\, V = v\right]$$

Benefits This simple limitation on inference networks results in powerful constructs. There are many perks:

- **inference is cheap**: despite being difficult in general, inference under naïve Bayesian assumptions makes the calculation a series of simple products

- **few parameters**: even with massive parameter quantities, the conditional probability tables are small overall, growing *linearly* rather than *exponentially*

- **allow estimation**: we've only considered hard-coded CPTs and inferences made from those, but we can actually estimate the CPTs themselves when given labeled data. The simplest way to do this is simply by counting: to find $\Pr\left[a_i \,|\, V = v\right]$, count the number of occurrences of a_i with $V = v$ and divide it by the total number of Vs in general.

- **allows classification**: as already mentioned, this structure lets us both do inference about features from classes (top-down) as well as classification from feature values (bottom-up).

- **empirically successful**: with enough data, naïve Bayesian networks do an incredible job. Apparently, Google uses it extensively.

Downsides This seems almost too good to be true. How can a truly-naïve model that assumes that there is complete independence between attributes perform well? There's no way that email containing the word "prince" is no more-or-less likely to contain the word "viagra" compared to email without the word "prince." In fact, I would expect spam emails targeting male genitalia to be a separate "class" of spam than emails that target people with ties to Nigeria (though they may both target gullible suckers).

This is exactly its downside: when there are strong inter-relationships between the attributes, the model can't make good inferences. However, these relationships (and their probabilities) don't matter so much when doing classification, since it just needs to be "good enough," and guess in the right direction from the limited information that it has.

There's also a flaw in the "counting" approach: notice that if a particular attribute a_i has never been seen or associated with a particular $v \in V$. Then, the entire product is zero. This is not intuitive and is exactly why this doesn't happen in practice: the probabilities need to be smoothed out so that one bad apple doesn't spoil the bunch.

PART II

Unsupervised Learning

Rather than providing labeled training pairs to an algorithm like we did before, in *un*supervised learning we are focused on inferring patterns from the data alone.

Contents

Randomized Optimization

> *People believe the only alternative to randomness is intelligent design.*
>
> — Richard Dawkins

In an optimization problem, we're given an input space X and an objective or **fitness function** $f : X \mapsto \mathbb{R}$; the goal is to and find the best $x^* \in X$ such that $f(x^*) = \max_x f(x)$. This chapter discusses algorithmic ways to find said x^*. Sometimes, calculus-based methods and even input space exhaustion can work, but this is rare to encounter in the real world; more often than not, we're dealing with complex functions with massive input spaces and many local optima. To tackle this complexity, we'll need **randomized optimization** algorithms.

5.1 Hill Climbing

The most straightforward of these is the **hill climbing** algorithm, which randomly guesses a starting point, then drifts to the best input within a local neighborhood of that point.

This is not super effective since it's highly-dependent on the starting point: it's easy to get stuck in a local optimum. However, we can run the algorithm many, many times to ensure that we don't always get stuck in the same one; even if we don't find the global optimum, we'll at least get stuck in a really good local one. This is called **random-restart hill climbing**.

5.2 Simulated Annealing

Here, the idea of random hill climbing is taken a step further: we don't always necessarily need to improve to be on a "good path" to an optimum. Sure, we could hope that another restart down the line will let us avoid the pending economic downturn,

ALGORITHM 5.1: A hill-climbing algorithm.

```
Input: $(X, f)$, the parameters for an optimization problem.
Input: $N(\cdot)$, a function that returns a list of neighbors to $x \in X$; this can be
       user-defined (encoding domain knowledge) or defined as part of the
       optimization problem itself.
Result: $x$ such that $f(x)$ is a local optimum.
$x \xleftarrow{\$} X$                                   // randomly choose $x \in X$
repeat
    $n^* \leftarrow \underset{n \in N(x)}{\arg\max}\ f(n)$
    if $f(n^*) > f(x)$ then
        $x = n^*$
    end
until $f(x) < f(n^*)$
return $x$
```

but it also makes sense to simply do a little exploration of the neighboring space beyond just $\underset{n \in N}{\arg\max}$.

This concept of **simulated annealing** leads to an eternal balance between **exploitation** and **exploration**: in hill-climbing, we always exploit the best possible direction, whereas now we'll do some additional exploration.

> FUN FACT: **Annealing**
>
> The concept of *annealing* comes from metallurgy, where metals are repeatedly heated and cooled to increase their ductility (basically bendability). It's a silly thing to name an algorithm after, but this idea of "temperature" is baked into the simulated annealing (see algorithm 5.2).

The key to simulated annealing is this idea of an "acceptance probability" function, which lets us smoothly interpret how seriously we should take "bad" points based on how bad they are.

$$P(x, x', T) = \begin{cases} 1 & \text{if } f(x') \geq f(x) \\ \exp\left(\frac{f(x') - f(x)}{T}\right) & \text{otherwise} \end{cases} \tag{5.1}$$

For high temperatures, it's likely to accept x' regardless of how bad it is; contrarily, a low T is likely to prefer "decent" x's that aren't much worse than x. In other words, $T \to 0$ behaves like hill climbing while $T \to \infty$ behaves like a random walk through f. In practice, slowly decreasing T is the way to go: the "cooling factor" α in algorithm 5.2 below controls this.

ALGORITHM 5.2: The simulated annealing algorithm.

```
Input: $(X, f)$, the parameters for an optimization problem.
Input: $N(\cdot)$, a function that returns a list of neighbors to $x \in X$; this can be
       user-defined (encoding domain knowledge) or defined as part of the
       optimization problem itself.
Result: $x$ such that $f(x)$ is a local optimum.
$\alpha \in [0, 1)$                                    // a "cooling factor"
for a finite number of iterations do
    $x' \xleftarrow{\$} N(x)$                          // randomly choose $x \in X$
    if $P(x, x', T) = 1$ then
        $x = x'$
    end
    $T = \alpha T$
end
return $x$
```

Simulated annealing comes with an interesting fact: we can actually determine the overall probability of ending at any given input. Namely,

$$\Pr[\text{end at } x] = \frac{\exp \frac{f(x)}{T}}{z_T} \qquad z_T \text{ is just a normalization term}$$

Notice that this[1] is actually in terms of the fitness of x, so the probability of ending at a particular input is directly proportional to how good it is! That's a really nice property.

5.3 Genetic Algorithms

Our third class of random optimization techniques are called **genetic algorithms**. It takes its inspiration from biology:

- We begin with *populations of individuals*—these are our various input sampling points.
- They undergo *mutations*—this is a local search (much like the $N(x)$ neighborhoods from before).

[1] This probability distribution is called the **Boltzmann distribution** and is used in physics.

- Then, there's *cross-over*, where different population groups have their attributes combined together to hopefully produce something novel and better-performing (like sexual selection).
- This happens over *generations*—this is simply the number of iterations of improvement.

The biggest deviation (and not the sexual kind) from what we've seen thus far is the idea of cross-over; otherwise, each population essentially operates like a random-restart in parallel.

5.3.1 High-Level Algorithm

We typically start with a randomly-generated initial population, P_0, of some size k. Then, we repeat an "evolution" process until convergence:

1. Compute the fitness of all $x \in P_t$.
2. Select the "most fit" individuals. This is where things can vary: under a strategy of **truncation selection**, you might just select the top half of individuals; on the other hand, under a **roulette wheel** strategy, you might base it off of a weighted probability so that low-fitness individuals still have a chance of sticking around in the gene pool. This is another place where we encounter of exploitation vs. exploration.
3. Pair up individuals, replacing the "least fit" individuals (those that weren't chosen in the previous step) via cross-over and mutation.

5.3.2 Cross-Over

Again, we've (intentionally) left cross-over as a "black box." Some concrete examples might help expand on what it means and how it can be useful.

Example: Bitstrings

Suppose, as we've been doing, that our input space is bitstrings. Let X be the set of 8-bit strings, and we're performing a cross-over with the following two "fit" individuals:

```
01101100
11010111
```

Perhaps a strategy might be to combine halves, resulting in two "offspring": `01100111` and `11011100`.

$$M : \boxed{\texttt{0110}}\boxed{\texttt{1100}}$$
$$D : \boxed{\texttt{1101}}\boxed{\texttt{0111}}$$

This strategy obviously encodes some important assumptions: the locality of the bits themselves has to matter; furthermore, it assumes that these "subspaces" can be independently optimized.

Suppose this previous assumption (bit locality) is incorrect. We can cross over in a different way, then: instead, let's either keep or flip the bits at random.

$$\begin{array}{rl} M: & \texttt{01101100} \\ D: & \texttt{11010111} \\ \cline{2-2} & \texttt{01100110} \\ & \texttt{11011101} \\ & \texttt{kkkkfkfk} \end{array}$$

where `k`/`f` correspond to "keep" and "flip," respectively. This is called **uniform** crossover.

5.3.3 Challenges

Representing an optimization problem as input to a genetic algorithm is difficult. However, if done correctly, it's often quite effective. It's commonly considered to be "the second-most effective approach" in a machine learning engineer's toolbelt.

5.4 MIMIC

The algorithms we've looked at thus far: hill climbing, simulated annealing, and genetic algorithms all feel relatively primitive because of a simple fact—they always just end at one point. They learn nothing about the space they're searching over, they remember nothing about where they've been, and they build no recollection over the underlying probability distribution that they're searching over.

Isbell '97 The main principle behind **MIMIC** is that it should be possible to directly model a probability distribution, successively refine it over time, and that should lead to a semblance of structure.

The probability distribution in question depends on a threshold, θ. It's formulated as:

$$\Pr^{\theta}[x] = \begin{cases} \frac{1}{z_\theta} & \text{if } f(x) \geq \theta \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

Namely, we're basically only considering samples $x \in X$, uniformally, that have a "good enough" fitness level.

5.4.1 High-Level Algorithm

Notice that $\Pr^{\theta_{\max}}[x]$ spans only the optima of f. Contrarily, $\Pr^{\theta_{\min}}[x]$ is simply the uniform distribution over X. MIMIC will start from this latter baseline, then iteratively improve, hoping to eventually reach the optima: $\Pr^{\theta_{\max}}[x] \rightsquigarrow \Pr^{\theta_{\min}}[x]$.

From a high level, the algorithm is as follows:

1. First, generate samples from the current $\Pr^{\theta_t}[x]$.
2. Then, set θ_{t+1} to be the n^{th} percentile of the dataset, like the truncation selection strategy of genetic algorithms.
3. Now, given only the samples for which $f(x) \geq \theta_{t+1}$, we estimate the new $\Pr^{\theta_{t+1}}[x]$.
4. Finally, repeat from Step 1 until n meets some thresholding criteria.

This is quite similar to genetic algorithms on the surface, but the way we represent our probability distribution at the step t will encode structure and meaning about our search space.

There are some key underlying assumptions here: first of all, estimating a probability distribution needs to be possible; second, we are hoping that choosing good samples from $\Pr^{\theta_t}[x]$ actually also gives us good samples at $\Pr^{\theta_{t+1}}[x]$.[2]

5.4.2 Estimating Distributions

Given a feature vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}$, the probability of seeing all particular features is the joint distribution over all of those features:

$$\Pr[\mathbf{x}] = \Pr[x_1 \,|\, x_2, \dots, x_n] \Pr[x_2 \,|\, x_3, \dots, x_n] \dots \Pr[x_n]$$

Obviously this is typically computationally-infeasible to calculate directly. However, we can make some assumptions to make it easier to calculate. Namely, we're going to craft **dependency trees**, which are special case of Bayesian networks that are trees:

$$\widehat{\Pr_{\pi}}[x] = \prod_i \Pr[x_i \,|\, \pi(x_i)]$$

Here, $\pi(\cdot)$ represents the "parent" of a node in the dependency tree. Since each node only has exactly one parent (aside from the root, so $\pi(x_0) = x_0$), the table of conditional probabilities stays much smaller than the full joint above.[3]

Why dependency trees, though? Well, they let us represent relationships between variables, and they do so compactly. It's worth noting that we're not *stuck* using

[2] A little more rigorously, we're basically assuming that $\Pr^{\theta}[x] \approx \Pr^{\theta+\varepsilon}[x]$.

[3] Specifically, the size of the table is quadratic in the number of features.

dependency trees; they're simply a means to an end of approximating the probability distribution (Step 3 above) and underlying structure. And in fact this specific choice of structure lets us represent similar relationships to cross-over in GAs.

Finding Depedency Trees

We'll do this in general, not referincing MIMIC or θ or any of that jazz. We have a true probability distribution P that we want to estimate, and $\widehat{P}_\pi$ is our dependency tree formulation from above.

Recall the KL-divergence (3.11) from Information Theory, which will let us measure the similarity between any two distributions:

$$\begin{aligned}
D_{\text{KL}}\left(P \parallel \widehat{P}_\pi\right) &= \sum P\left(\log_2 P - \log_2 \widehat{P}_\pi\right) \\
&= -h(p) + \sum_i h(x_i \mid \pi(x_i)) && p\log_2 p \text{ is just entropy, } -h(p) \\
J_\pi &= \sum_i h(x_i \mid \pi(x_i))
\end{aligned}$$

At the end, we end up with a sort of cost function J_π that we're aiming to minimize: find each feature's parent such that overall entropy is low. In other words, find parents that give us a lot of information about their child features.

To make this J_π easy to compute—for reasons beyond my understanding—we'll introduce a new term: the entropy of each feature. This is okay because it doesn't affect $\pi(\cdot)$ which is what we're minimizing.

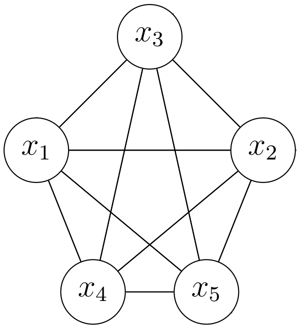
$$\begin{aligned}
\min_\pi J_\pi &= \sum_i h(x_i \mid \pi(x_i)) \\
\min_\pi J'_\pi &= -\sum_i h(x_i) + \sum_i h(x_i \mid \pi(x_i))
\end{aligned}$$

Both of these will give us the same $\pi(\cdot)$. This term is related to the mutual information (see (3.10) in Information Theory):

$$\begin{aligned}
\min_\pi J'_\pi &= -\sum_i h(x_i) + \sum_i h(x_i \mid \pi(x_i)) \\
&= -\sum_i I\left(x_i; \pi(x_i)\right) \\
\max_\pi J'_\pi &= \sum_i I\left(x_i; \pi(x_i)\right)
\end{aligned}$$

In English, what we've come to show is that in order to maximize the similarity between a true probability distribution and our guess *(I know it's been a while, but this is* *what we started with)*, we want to maximize the mutual information between each feature and its parent.

Finally, we need to actually calculate this optimization. This equation can actually be represented byy a fully connected graph: the nodes are the x_i features and the edges are the mutual information $I(x_i; x_j)$ between them:



And since we want to find the tree that maximizes the total mutual information, this is equivalent to finding the *maximum* spanning tree. Since more-traditional algorithms find the *minimum* spanning trees, this is equivalent to just inverting the I edge weights.[4]

Coming Full Circle

Now that we understand how to estimate a probability distribution as well as pull samples from it, we can turn the high-level algorithm into reality: our $\Pr^{\theta_t}[x]$ is simply $\widehat{P}_\pi$, and estimation is the spanning tree over mutual information.

5.4.3 Practical Considerations

MIMIC does well with structure: when the problems you're trying to solve depend on relationships between features rather than specific feature values, MIMIC can handle them well; MIMIC does not care about "ties" in optima.

If a fitness function can't be represented by a probability distribution for all possible θs, **MIMIC struggles**. Furthermore, it still can get stuck in local optima. Finally, time complexity is an issue—though in practice MIMIC takes orders of magnitude fewer iterations, a single iteration of MIMIC takes far more time.

However, if information about the problem is important, though, MIMIC is the only way to go since it provides, as we've said time and time again, structure. If evaluating the fitness function $f(x)$ has a high cost, MIMIC can avoid far more evaluations relative to the other randomized algorithms.

[4] Since this is a fully-connected graph, Prim's algorithm is the way to go.

Index of Terms