Machine Learning

or: The Unofficial Notes on the Georgia Institute of Technology's **CS7641**: *Machine Learning*

George Kudrayvtsev
george.k@gatech.edu

Last Updated: January 21, 2020

Rough Draft

Preface

> *I read that Teddy Roosevelt once said, "Do what you can with what you have where you are." Of course, I doubt he was in the tub when he said that.*
>
> — Bill Watterson, *Calvin and Hobbes*

Before we begin to dive into all things algorithmic, I'll enumerate a few things I do in this notebook to elaborate on concepts:

- An item that is **highlighted like this** is a "term;" this is some vocabulary that will be used and repeated regularly in subsequent sections. I try to cross-reference these any time they come up again to link back to its first defined usage; most mentions are available in the Index.
- An item in a maroon box, like...

Boxes: A Rigorous Approach

... this often represents fun and interesting asides or examples that pertain to the material being discussed. They are largely optional, but should be interesting to read and have value, even if it's not immediately rewarding.

- An item in a blue box, like...

Quick Maffs: Proving That the Box Exists

... this is a mathematical aside; I only write these if I need to dive deeper into a concept that's mentioned in lecture. This could be proofs, examples, or just a more thorough explanation of something that might've been "assumed knowledge" in the text.

I also sometimes include margin notes like the one here (which just links back here) that reference content sources so you can easily explore the concepts further. Linky

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
...	

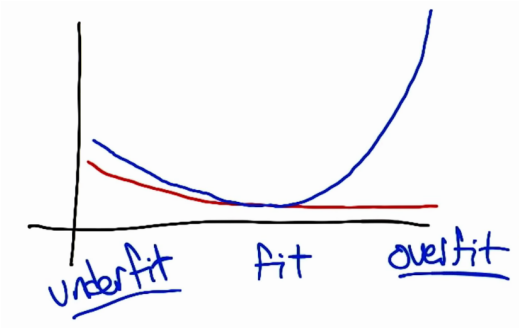
Contents

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.

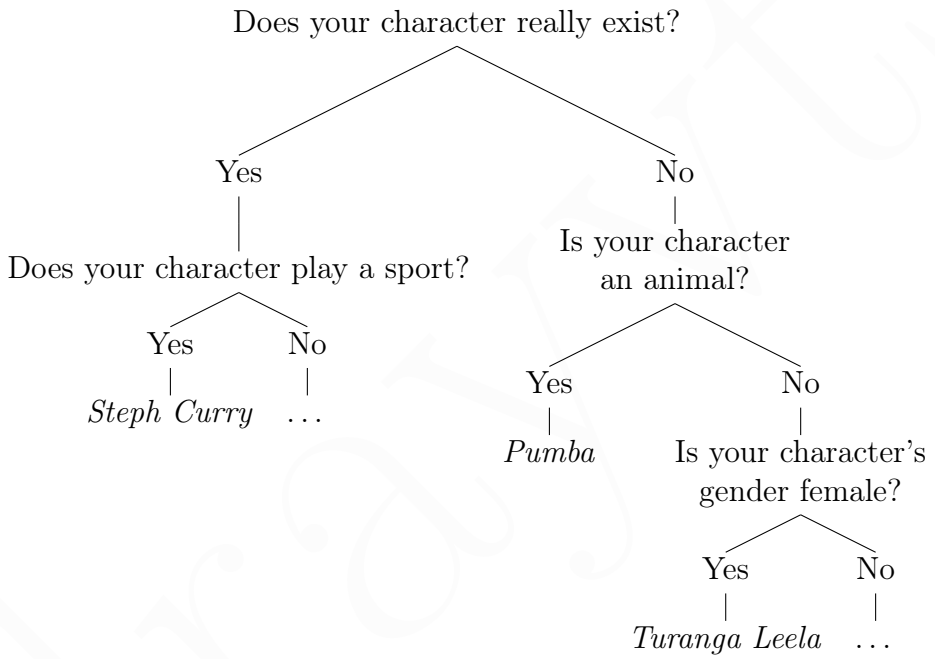
2.1 Decision Trees

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:"



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 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.

2.1.1 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.

2.1.2 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:

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in A} \frac{|S_v|}{|S|} \cdot \text{Entropy}(S_v) \tag{2.1}$$

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

$$\text{Entropy}(A) = \sum_{v \in A} \Pr[v] \cdot \log \Pr[v] \tag{2.2}$$

We'll dive into these more later when we get to Randomized Optimization, but 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: $\max_{A \in \mathcal{A}} \text{Gain}(S, A)$.

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."

2.1.3 Considerations

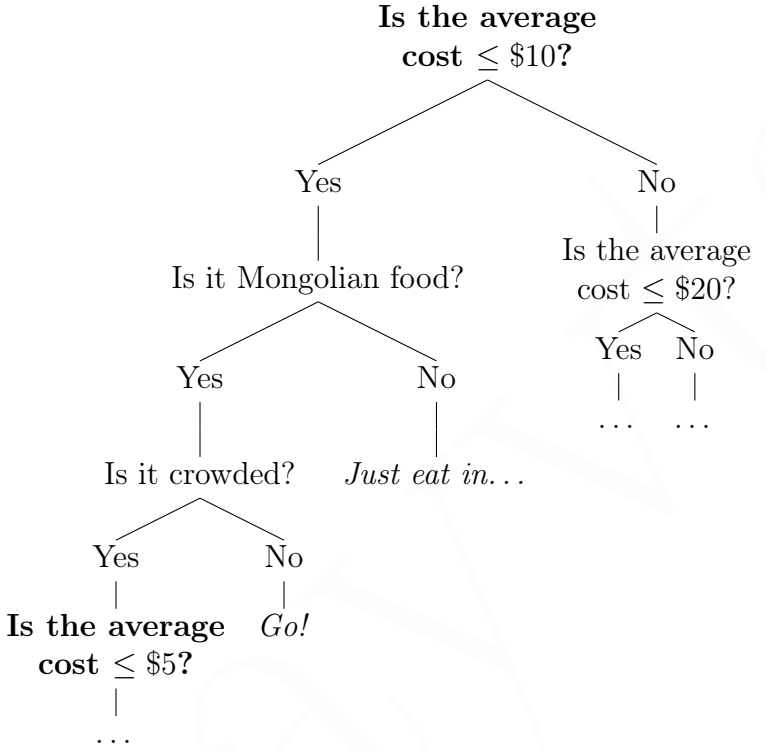
Asking (Continuous) Questions

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 **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.

Rough Draft

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*").

3.1 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 us this is the same as minimizing the **pro-**

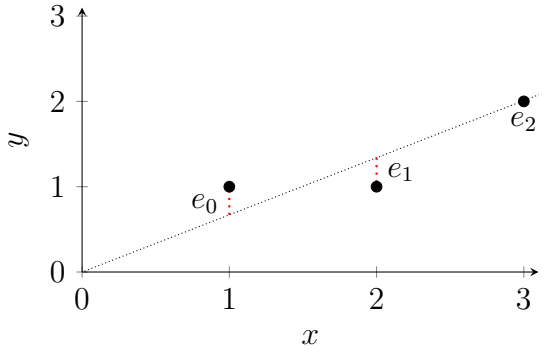


Figure 3.1: A set of points with "no solution": no line passes through all of them. The set of errors is plotted in red: (e_0, e_1, e_2).

jection 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 3.1. 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: $\mathbf{p} = \underset{C(\mathbf{A})}{\text{proj}}\ \mathbf{y}$. This is the closest possible vector in the column space to $\mathbf{y}$, which is exactly the distance we were trying to minimize! Thus,

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

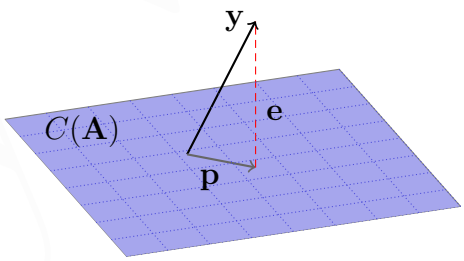


Figure 3.2: The vector $\mathbf{y}$ relative to the column space of $\mathbf{A}$, and its projection $\mathbf{p}$ onto the column space.

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:

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

which is exactly what the linear regression algorithm calculates.

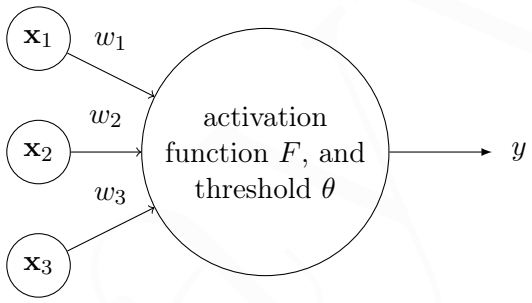
More Resources. This section basically summarizes and synthesizes this Khan Academy video, this lecture from our course (which goes through the *full* derivation), this section of *Introduction to Linear Algebra*, and this explanation from NYU. These links are provided in order of clarity.

3.2 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.

3.2.1 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 3.1: Basic Perceptron

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

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

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

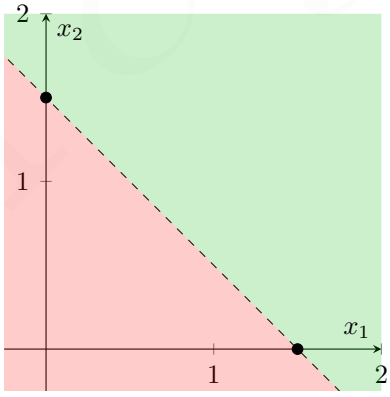
Well, pretty simply, we get

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

The perceptron doesn't fire!

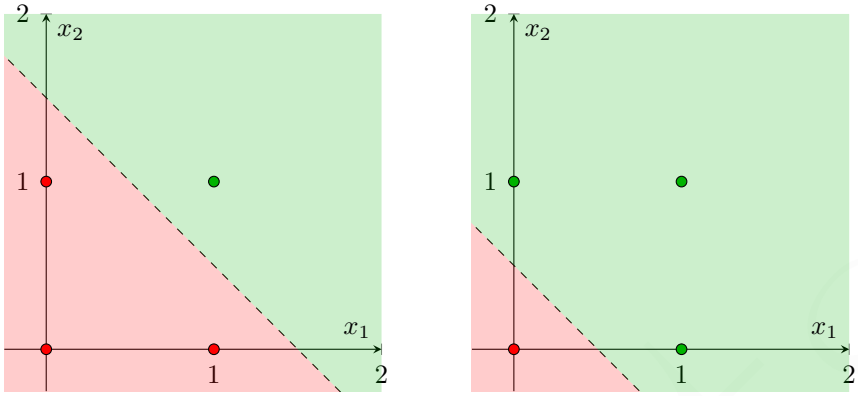
What kind of functions can we represent with a perceptron? Suppose we have two inputs, x_1 and x_2, along with equal weights $w_1 = w_2 = \frac{1}{2}, \theta = \frac{3}{4}$. For what values of x_i will we get an activation?

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

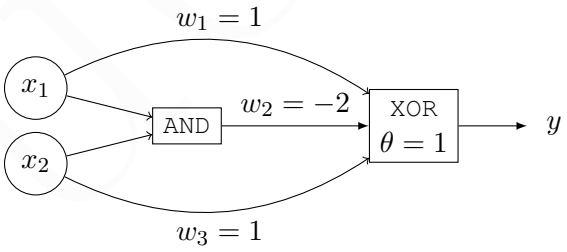
To build up an intuition for how perceptrons can be combined, let's binary XOR. It can't be described by a single perceptron because it's not a linear function; however, it *can* be described by several!

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 3.1: 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 3.3.



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 3.3: 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, $w_1 x_1 + w_2 F_{\text{AND}}(x_1, x_2) + w_3 x_2$.

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 $\hat{y}$.

Claim 3.1. *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.

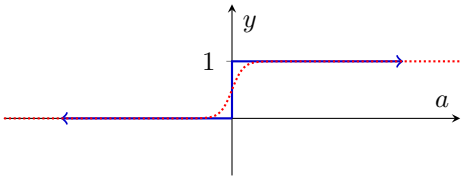
To reiterate our learning rules, we have:

$$\Delta w_i = \eta (y - \hat{y}) x_i \tag{3.1}$$

$$\Delta w_i = \eta (y - a) x_i \tag{3.2}$$

3.2.2 Sigmoids

The similarity between the learning rules in (3.1) and (3.2) 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:

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{3.3}$$

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.

3.2.3 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 3.4 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.

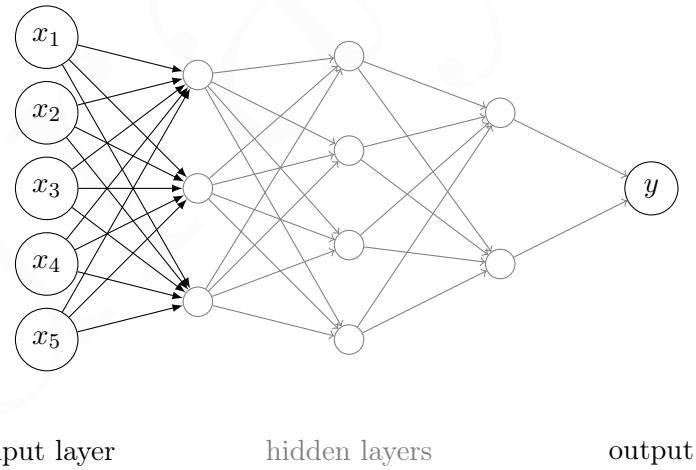


Figure 3.4: A 4-layer neural network with 5 inputs and 3 hidden layers.

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.

3.2.4 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.

3.3 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.

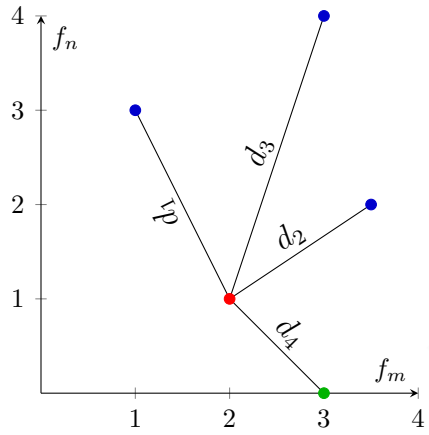
For a high-level example, consider our fundamental "line of best fit" problem. Given some input points, linear regression will determine the line that minimizes the error with the data, and then we can use the line directly to predict new values. With instance-based learning methods, we instead would 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.

3.3.1 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:

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

which is a nice, dark blue-green color.

An extremely simple (and non-performant) version of the kNN algorithm is formalized in algorithm 3.1; 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 up-front 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 3.1; 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 3.1: 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 := {∞, ...}
foreach x_i ∈ 𝒳 do
    foreach j ∈ [1, k] do
        if d(x_i, x') < dists[j] then
            closest[j] = x_i
            dists[j] = d(x_i, x')
        end
    end
end
// Return the distance-weighed average of the k closest neighbors.
return (Σ_{d_i ∈ dists} d_i · x_i) / (Σ_{d_i ∈ 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.

3.4 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**.

3.4.1 Bagging

Turns out, simply making choosing data uniformally randomly to form our subset (with replacement) works pretty well. Similarly-simply, combining the results with

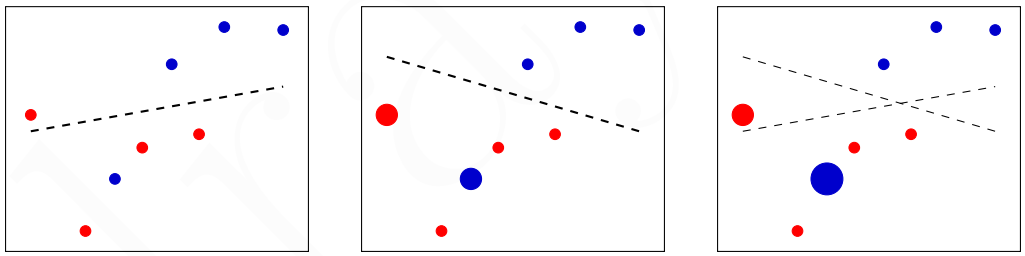
an average also works well. This technique is called **boostrap aggregation**, or more-commonly **bagging**.

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.

3.4.2 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 3.5: 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 3.2: 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,

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

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: a learner that does better than chance for *any* distribution of data (where ϵ is a number very close to zero):

$$\forall \mathbb{D} : \Pr_{\mathbb{D}}[\cdot] \leq 1/2 - \epsilon$$

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 3.2. 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.*

We return to the world of classification, where our training set maps from a feature vector to a "correct" or "incorrect" label, $y_i \in \{-1, 1\}$: Freund & Schapire, '99

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

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.[2]

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.

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

[2] 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 3.2), 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.

Final Hypothesis As you can see in algorithm 3.2, 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.

ALGORITHM 3.2: 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^*.

$\hat{\mathbf{w}} = \begin{bmatrix} 1/m & 1/m & \ldots \end{bmatrix}$ `// 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` α_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^*

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 3.6; from them, we can draw some straightforward conclusions:

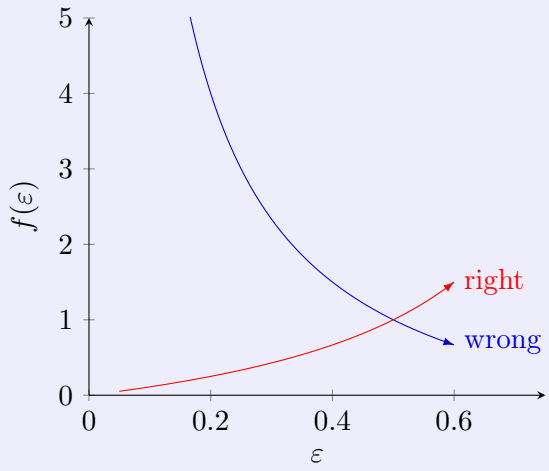


Figure 3.6: The two ways the exponential can go when boosting, depending on whether or not the classifier gets sample i right ($\frac{1-\varepsilon}{\varepsilon}$, in red) or wrong ($\frac{\varepsilon}{1-\varepsilon}$, in blue).

- 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.

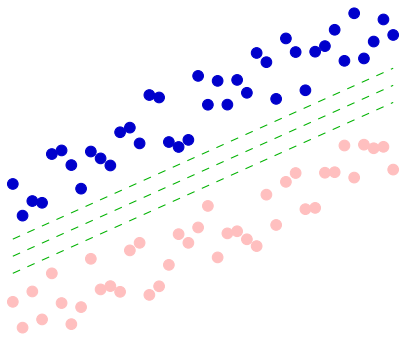
3.5 Support Vector Machines

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.

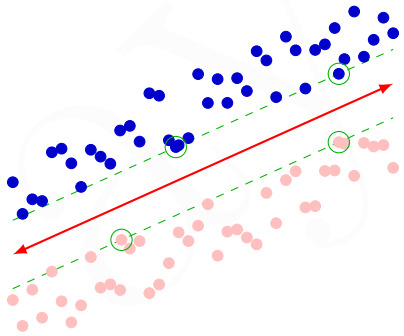
I covered SVMs briefly in my notes on computer vision. For a gentler introduction to the topic, refer there. This section will have a lot more assumed knowledge so that I can avoid repeating myself.

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



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?

3.5.1 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 $\begin{bmatrix} x & y & 1 \end{bmatrix}$:

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

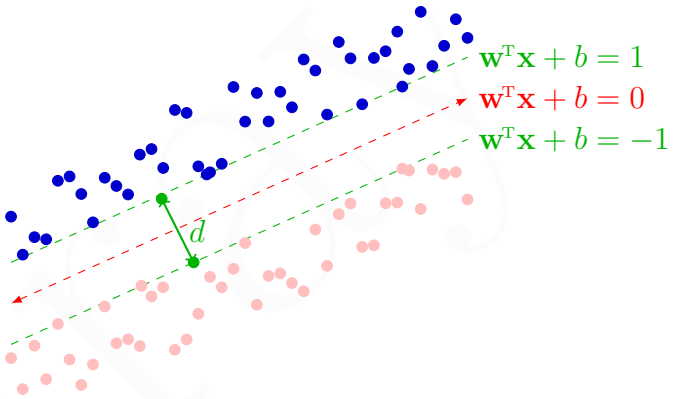
Thus if we let $\mathbf{s} = \begin{bmatrix} a & b & c \end{bmatrix}$ and $\mathbf{w} = \begin{bmatrix} x & y & 1 \end{bmatrix}$, then our line can be expressed simply as: $\mathbf{w}^{\mathrm{T}}\mathbf{s} = 0$. This lets us representing a line in vector form and use any number of dimensions. Our $\mathbf{w}$ 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: $\mathbf{w}^{\mathrm{T}}\mathbf{s} + c = 0$.

3.5.2 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 $y = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b$.

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 $\mathbf{w}^{\mathrm{T}}\mathbf{x} + b = 0$. 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:



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

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

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**:

$$\begin{aligned} &\text{Minimize:} && \frac{1}{2}\|\mathbf{w}\|^2 \\ &\text{Subject to:} && y_i(\mathbf{w}^\mathrm{T}\mathbf{x}_i + b) \geq 1 \end{aligned}$$

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: $f(\mathbf{x}) = \text{sign}\left(\sum_i \alpha_i y_i \boxed{\mathbf{x}_i^\mathrm{T}\mathbf{x}} + b\right)$

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. This is about to become the source of power of SVMs: we'll use this to find separation boundaries between our data points in higher dimensions than our features provide.

PART II

Unsupervised Learning

Machine Learning

Randomized Optimization

Machine Learning

Index of Terms