optional 6.86x notes

You do not need to read these notes at all to get an A in this course; conversely, you may not cite these notes when solving homework or exams. Depending on your interests, you might want to start at chapter A (then BCDEF) or D (then ABCEF). Sample chapters G, H (review of some but not all prerequisites) as needed.

Clickable Table of Contents

I shall create! If not a note, a hole. If not an overture, a desecration.
— gwendolyn brooks

A. prologue

0. bird's eye view

KINDS OF LEARNING — How do we communicate patterns of desired behavior? We can teach:
by instruction: "to tell whether a mushroom is poisonous, first look at its gills..."
by example: "here are six poisonous fungi; here, six safe ones. see a pattern?"
by reinforcement: "eat foraged mushrooms for a month; learn from getting sick."
Machine learning is the art of programming computers to learn from such sources. We'll focus on the most important case: learning from examples.°

Exercise: What's a thing you know now but not last month? What kinds of signal taught you?

← In Chapter E, we'll see that learning by example is key to the other modes of learning.

FROM EXAMPLES TO PREDICTIONS — For us, a pattern of desired behavior is a function that for each given situation/prompt returns a favorable action/answer. Our goal is to write a program that, from a list of N examples of prompts and matching answers, determines an underlying pattern. We consider our program a success if this pattern accurately predicts answers corresponding to new, unseen prompts. We often define our program as a search, over some set $\mathcal{H}$ of candidate patterns, to minimize some notion of "discrepancy from the example data".

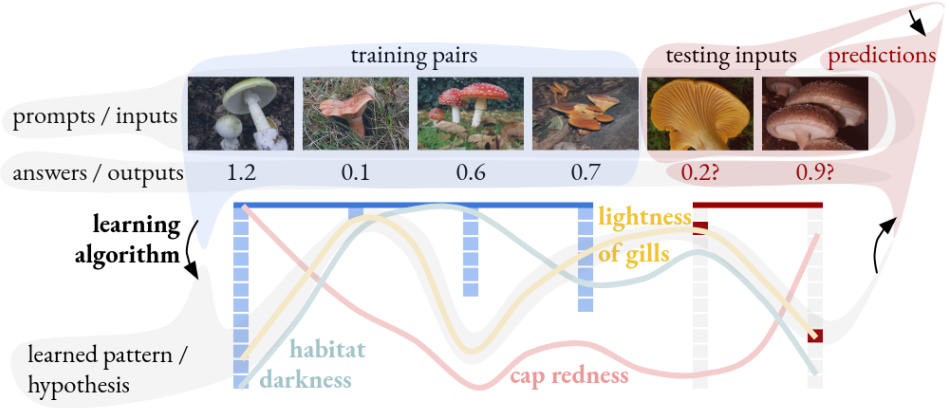


Figure 1: A program that learns to predict mushrooms' poison levels: first takes a list of labeled mushrooms as input (blue blob); searches through candidate patterns (here, the wiggly curves labeled `lightness of gills`, `habitat darkness`, and `cap redness`); and returns the pattern that best fits the examples. Evaluating this pattern on new mushrooms, we predict their poison levels (red blob).

Three arrows show how training examples induce a learned pattern, which, together with testing prompts, induces predictions. Part of specifying the learning program is specifying the set of candidate patterns to consider.

To save ink, say that $\mathcal{X}$ is the set of possible prompts; $\mathcal{Y}$, of possible answers.° In the mushrooms example, $\mathcal{X}$ contains all conceivable mushrooms and $\mathcal{Y}$ contains all conceivable poison levels (perhaps all the non-negative real numbers).

← If we like, we can now summarize the data flow in symbols. A pattern is a function of type $\mathcal{X} \to \mathcal{Y}$. And we can model the examples from which our program learns as a list of type $(\mathcal{X} \times \mathcal{Y})^N$. Then a program that learns from examples has type:

$$\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^N \to (\mathcal{X} \to \mathcal{Y})$$

Once we allow uncertainty by letting patterns map to *probability distributions* over answers, the type will change to:

$$\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^N \to (\mathcal{X} \to \text{DistributionsOn}(\mathcal{Y}))$$

SUPERVISED LEARNING — We'll soon allow uncertainty by letting patterns map to *probability measures* over answers. Even if $|X| = 1$ — say, $X = \{\text{"produce a beautiful melody"}\}$ — we may seek to learn the complicated distribution over answers, e.g. to generate a diversity of apt answers. So-called **unsupervised learning** thus concerns output structure. By contrast, **supervised learning** (our main subject), concerns the input-output relation; it's interesting when there are many possible prompts.

2. a tiny example: handwritten digit classification

The learning process is something you can incite, literally incite, like a riot.
— audre lorde

MEETING THE DATA — $\mathcal{X} = \{\text{grayscale } 28\times28\text{-pixel images}\}$; $\mathcal{Y} = \{1, 9\}$. Each datum (x, y) arises as follows: we randomly choose a digit $y \in \mathcal{Y}$, ask a human to write that digit in pen, and then photograph their writing to produce $x \in \mathcal{X}$.

Figure 2: Twenty example pairs. Each photo x is a 28×28 grid of numbers representing pixel intensities. The light gray background has intensity 0.0; the blackest pixels, intensity 1.0. Below each photo x we display the corresponding label y: either $y = 1$ or $y = 9$. We'll adhere to this color code throughout this tiny example.

When we zoom in, we can see each photo's 28×28 grid of pixels. On the computer, this data is stored as a 28×28 grid of numbers: 0.0 for bright through 1.0 for dark. We'll name these 28×28 grid locations by their row number (counting starting from the top) followed by their column number (counting starting from the left). So location $(0, 0)$ is the upper left corner pixel; location $(27, 0)$ is the lower left corner pixel.

Exercise: Where is location $(0, 27)$? In which direction is $(14, 14)$ off-center?

As part of getting to know the data, it's worth taking a moment to think about how we would go about hand-coding a digit classifier. The challenge is to complete the pseudocode "`if (?) then predict y=9 else predict y=1`". Well, 9s tend to have more ink than than 1s — should `(?)` threshold by the photo's darkness? Or: 1s and 9s tend to have different heights — should `(?)` threshold by the photo's dark part's height?

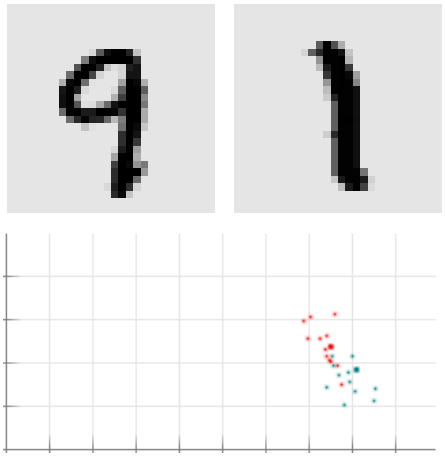


Figure 3: Our size-N = 25 set of training examples, viewed in the darkness-height plane. The vertical *darkness* axis ranges $[0.0, 0.25]$; the horizontal *height* axis ranges $[0.0, 0.5]$. The origin is at the lower left. Each cyan dot represents a $y = 1$ example; each red dot, a $y = 9$ one. The big 9 above has darkness and height $(0.118, 0.375)$; the big 1, $(0.092, 0.404)$. See where they are in this plot?

To make this precise, let's define a photo's *darkness* as its average pixel darkness; its *height* as the standard deviation of the row index of its dark pixels. For convenience let's normalize both height and darkness to have max possible value 1.0. Such functions from inputs in $\mathcal{X}$ to numbers are called **features**.

```
SIDE = 28
def darkness(x):
  return np.mean(np.mean(x, axis=0), axis=0)
def height(x):
  return np.std([row for col in range(SIDE)
                     for row in range(SIDE)
                     if 0.5 < x[row][col]  ])/(SIDE/2.0)
```

Exercise: Make a width feature. Plot the training data in the height-width plane.

So we can threshold by darkness or by height. But this isn't very satisfying, since sometimes there are especially dark 1s or tall 9s. We thus arrive at the idea of using *both* features: 9s are darker than 1s *even relative to their height*. So we might write something like `a*darkness(x)+b*height(x)>0` for our condition.

Exercise: Guess a good pair (a, b) based on the training data.

```
def hand_coded_predict(x):
  return 9 if (4.0)*darkness(x)+(-1.0)*height(x)>0 else 1
```

Exercise: Implement a crude "hole-detector" feature. False positives are okay.

Exercise: What further features might help us to separate digits 1 from 9?

CANDIDATE PATTERNS — We can generalize the hand-coded hypothesis from the previous passage to other coefficients besides $1 \cdot \text{height}(x) - 2 \cdot \text{darkness}(x)$. We let our set $\mathcal{H}$ of candidate patterns contain all "linear hypotheses" $f_{a,b}$ defined by:

$$f_{a,b}(x) = 9 \text{ if } a \cdot \text{darkness}(x) + b \cdot \text{height}(x) > 0 \text{ else } 1$$

Each $f_{a,b}$ makes predictions of ys given xs. As we change a and b, we get different predictors, some more accurate than others.

```
def predict(x,a,b):
  return 9 if a*darkness(x) + b*height(x) > 0 else 1
```

Exercise: See Fig. 4 and match up ■□□□'s 3 lines with □■□□'s 3 boxed points.

OPTIMIZATION — Let's write a program $\mathcal{L}$ that given a list of *training examples* produces a hypothesis in $h \in \mathcal{H}$ that helps us predict the labels y of yet-unseen photos x (*testing examples*). Insofar as training data is representative of testing data, it's sensible to return a $h \in \mathcal{H}$ that correctly classifies maximally many training examples. To do this, let's make $\mathcal{L}$ loop over all integer pairs (a, b) in $[-99, +99]$:

```
def accuracy_on(examples,a,b):
  return sum(1.0 for x,y in examples if predict(x,a,b)==y)/len(examples)

def best_hypothesis():
  return max((accuracy_on(training_data, a, b), (a,b))
             for a in range(-99,100)
             for b in range(-99,100))
```

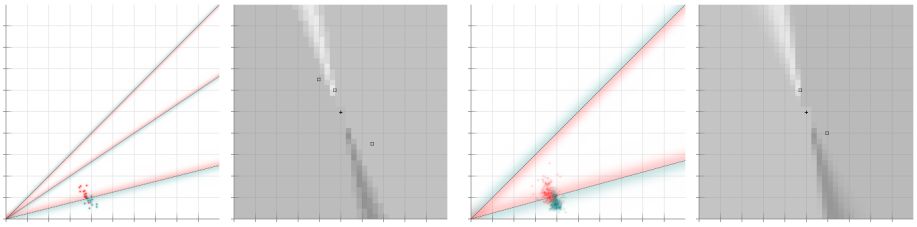


Figure 4: **Training (■■□□) and testing (□□■■).** 3 hypotheses classify training data in the darkness-height plane (■□□□); glowing colors distinguish a hypothesis' 1 and 9 sides. Each point in the (a, b) plane (□■□□) represents a hypothesis; darker regions misclassify a greater fraction of training data. Panes □□■■ show the same for *testing* data. ■□■□'s axes range $[0, 1.0]$. □■□■'s axes range $[-99, +99]$.

When we feed $N = 25$ training examples to $\mathcal{L}$, it produces $(a, b) = (80, -20)$ as a minimizer of **training error**, i.e., of the fraction of training examples misclassified. It misclassifies only 12% of training examples. Yet the same hypothesis misclassified a greater fraction — 18% — of fresh, yet-unseen testing examples. That latter number — called the **testing error** — represents our program's accuracy "in the wild"; it's the number we most care about. The difference between training and testing error is the difference between our score on a practice exam or homework, where we're allowed to review mistakes we made and do a second try, versus our score on an exam, where we don't know the questions beforehand and aren't allowed to change our answers once we get our grades back.

Exercise: visualize $f_{a,b}$'s error on $N = 1$ example as a function of (a, b).

ERROR ANALYSIS — Intuitively, our testing error of 18% comes from three sources: **(a)** the failure of our training set to be representative of our testing set; **(b)** the failure of our program to exactly minimize training error over $\mathcal{H}$; **(c)** and the failure of our hypothesis set $\mathcal{H}$ to contain "the true" pattern.

These are respectively errors of **generalization**, **optimization**, **approximation**.

Here, we got optimization error $\approx 0\%$ (albeit by *unscalable brute-force*). Because optimization error is zero in our case, the approximation error and training error are the same: $\approx 12\%$. The approximation error is so high because our straight lines are *too simple*: height and darkness lose useful information, and the "true" boundary between training digits looks curved. Finally, our testing error $\approx 18\%$ exceeds our training error. We thus suffer a generalization error of $\approx 6\%$: we *didn't perfectly extrapolate* from training to testing situations. In 6.86x we'll address all three italicized issues.

Exercise: why is generalization error usually positive?

3. supervised learning framework

Everybody gets so much information all day long that they lose their common sense.
— gertrude stein

FORMALISM — Draw training examples $\mathcal{S} : (\mathcal{X} \times \mathcal{Y})^N$ from nature's distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$. A pattern $f : \mathcal{X} \to \mathcal{Y}$ has **training error** $\text{trn}_\mathcal{S}(f) = \mathbb{P}_{(x,y)\sim\mathcal{S}}[f(x) \neq y]$, an average over examples; and **testing error** $\text{tst}(f) = \mathbb{P}_{(x,y)\sim\mathcal{D}}[f(x) \neq y]$, an average over nature. A *learning program* is a function $\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^N \to (\mathcal{X} \to \mathcal{Y})$; we want to design $\mathcal{L}$ so that it maps typical $\mathcal{S}$s to fs with low $\text{tst}(f)$.

LEARNING ERROR — As in the previous section's tiny example, we often define $\mathcal{L}$ to roughly minimize $\text{trn}_\mathcal{S}$ over a set $\mathcal{H} \subseteq (\mathcal{X} \to \mathcal{Y})$ of candidate patterns. Then tst decomposes into the failures of $\text{trn}_\mathcal{S}$ to estimate tst (generalization), of $\mathcal{L}$ to minimize $\text{trn}_\mathcal{S}$ (optimization), and of $\mathcal{H}$ to contain nature's truth (approximation):

$$\begin{aligned}
\text{tst}(\mathcal{L}(\mathcal{S})) = {} & \text{tst}(\mathcal{L}(\mathcal{S})) && - \text{trn}_\mathcal{S}(\mathcal{L}(\mathcal{S})) && \} \textbf{ generalization} \text{ error} \\
& + \text{trn}_\mathcal{S}(\mathcal{L}(\mathcal{S})) && - \inf_\mathcal{H}(\text{trn}_\mathcal{S}(f)) && \} \textbf{ optimization} \text{ error} \\
& + \inf_\mathcal{H}(\text{trn}_\mathcal{S}(f)) && && \} \textbf{ approximation} \text{ error}
\end{aligned}$$

These terms are in tension. For example, as $\mathcal{H}$ grows, the approx. error may decrease while the gen. error may increase — this is the "**bias-variance** tradeoff".

WORKFLOW — We first *frame*: what data will help us solve what problem? To do this, we *factor* our complex prediction problem into simple classification or regression problems; randomly *split* the resulting example pairs into training, dev(elopment), and testing sets; and *visualize* the training data to weigh our intuitions.

Next, we *model*: we present the data to the computer so that true patterns are more easily found. Here we inject our *domain knowledge* — our human experience and intuition about which factors are likely to help with prediction. Modeling includes *featurizing* our inputs and choosing appropriate *priors* and *symmetries*.

During *training*, the computer searches among candidate patterns for one that explains the examples relatively well. We used brute force above; we'll soon learn faster algorithms such as *gradient descent* on the training set for parameter selection and *random grid search* on the dev set for hyperparameter selection.

Finally, we may *harvest*: we derive insights from the pattern itself° and we predict outputs for to fresh inputs. Qualifying both applications is the pattern's quality. To assess this, we measure its accuracy on our held-out testing data.

← which factors ended up being most important?

B. linear models

0. linear approximations

He had bought a large map representing the sea,
Without the least vestige of land:
And the crew were much pleased when they found it to be
A map they could all understand.
— charles dodgson

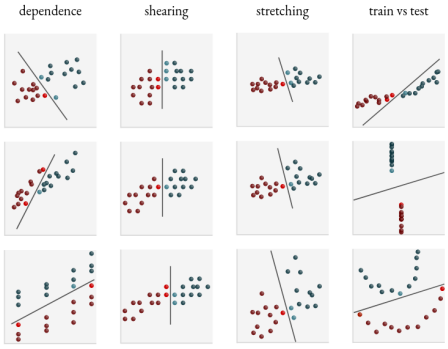
FEATURIZATION — As in the prologue, we represent our input x as a fixed-length list of numbers so that we can treat x with math. represented each photograph by 2 numbers: height and darkness. We could instead have represented each photograph by 784 numbers, one for the brightness at each of the $28 \cdot 28 = 784$ many pixels. Or by 10 numbers, each measuring the overlap of x's ink with that of "representative" photos of the digits 0 through 9.

When we choose how to represent x by a list of numbers, we're choosing a **featurization**. We call each number a "feature". For example, height and darkness are two features.

TODO: mention one-hot, etc TODO: mention LOWRANK (sketching; also, for multiregression) There are lots of interesting featurizations, each making different patterns easier to learn. So we judge a featurization with respect to the kinds of patterns we use it to learn. TODO: graphic of separability; and how projection can reduce it Learning usually happens more accurately, robustly, and interpretably when our featurization is abstract (no irrelevant details) but complete (all relevant details), compressed (hard to predict one feature from the others) but accessible (easy to compute interesting properties from features).

data-based featurizations via kernels will soon learn featurizations hand featurization in kaggle and medicine

TODO: projectivization (say this foreshadows kernel discussion?)



GEOMETRY OF FEATURE-SPACE — Now say we've decided on a **featurization** of our input data x.

$$f_{a,b}(x) = \begin{array}{l} 0 \text{ if } a \cdot \text{width}(x) + b \cdot \text{darkness}(x) < 0 \text{ else } 1 \end{array}$$

Just because two features both correlate with a positive label ($y = +1$) doesn't mean both features will have positive weights. In other words, it could be that the *blah*-feature correlates with $y = +1$ in the training set and yet, according to the best hypothesis for that training set, the bigger a fresh input's blah feature is, the *less* likely its label is to be $+1$, all else being equal. That last phrase "all else being equal" is crucial, since it refers to our choice of coordinates. Illustrate 'averaging' of good features vs 'correction' of one feature by another (how much a feature correlates with error) In fact, t This is the difference between *independence* and *conditional independence*.

Shearing two features together — e.g. measuring cooktime-plus-preptime together with cooktime rather than preptime together with cooktime — can impact the decision boundary. Intuitively, the more stretched out a feature axis is, the more the learned hypothesis will rely on that feature.

Stretching a single feature — for instance, measuring it in centimeters instead of meters — can impact the decision boundary as well. Intuitively, the more stretched out a feature axis is, the more the learned hypothesis will rely on that feature.

Note that

TODO: example featurization (e.g. MNIST again?)

RICHER OUTPUTS: LARGER $\mathcal{Y}$ — We've learned how to construct a set $\mathcal{H}$ of candidate patterns

$$f_{\vec{w}}(\vec{x}) = \text{step}(\vec{w} \cdot \vec{x})$$

that map (a featurization of) a prompt $\vec{x}$ to a binary answer $y = 0$ or $y = 1$.

What if we're interested in predicting a richer kind of y? For example, maybe there are k many possible values for y instead of just 2. Or maybe there are infinitely many possible values — say, if y is a real number or a length-k list of real numbers. Or maybe we want the added nuance of predicting probabilities, so that f might output "19% chance of label $y = 0$ and 80% chance of label $y = 1$ and 1% chance of label $y = 2$" instead of just "$y = 1$".

I'll write formulas and then explain.

multi-class hard classification	$f_{(\vec{w}_i : 0 \le i < k)}(\vec{x}) = \text{argmax}_i(\vec{w}_i \cdot \vec{x} : 0 \le i < k)$
multi-output regression	$f_{(\vec{w}_i : 0 \le i < k)}(\vec{x}) = (\exp(\vec{w}_i \cdot \vec{x}) : 0 \le i < k)$
multi-class soft classification	$f_{(\vec{w}_i : 0 \le i < k)}(\vec{x}) = \text{normalize}(\exp(\vec{w}_i \cdot \vec{x}) : 0 \le i < k)$

TODO: interpret

TODO: discuss measures of goodness!

TODO: talk about structured (trees/sequences/etc) output!

1. iterative optimization

Hey Jude, don't make it bad
Take a sad song and make it better
Remember to let her under your skin
Then you'll begin to make it
Better, better, better, better, better, better, ...
— paul mccartney, john lennon

(STOCHASTIC) GRADIENT DESCENT — We have a collection $\mathcal{H}$ of candidate patterns together with a function $1 - \text{trn}_{\mathcal{S}}$ that tells us how good a candidate is. In §A we found a nearly best candidate by brute-force search over all of $\mathcal{H}$; this doesn't scale to the general case, where $\mathcal{H}$ is intractably large. So: *what's a faster algorithm to find a nearly best candidate?*

← We view $1 - \text{trn}_{\mathcal{S}}$ as an estimate of our actual notion of "good": $1 - \text{tst}$.

A common idea is to start arbitrarily with some $h_0 \in \mathcal{H}$ and repeatedly improve to get $h_1, h_2, \cdots$. Two questions are: *how do we select* h_{t+1} *in terms of* h_t? And *how do we know when to stop*? We'll discuss termination conditions later — for now, let's agree to stop at h_{10000}.

As for selecting a next candidate, we'd like to use more detailed information on h_t's inadequacies to inform our proposal h_{t+1}. Intuitively, if h_t misclassifies a particular $(x_n, y_n) \in \mathcal{S}$, then we'd like h_{t+1} to be like h_t but nudged a bit in the direction of accurately classifying (x_n, y_n).

FILL IN FOR PROBABILITIES (LOGISTIC) MODEL

This is the idea of **gradient descent**. MOTIVATE AND GIVE PSEUDOCODE FOR STOCHASTIC GD

Given a list of training examples, a probability model, and a hypothesis $\vec{w}$, we can compute $\vec{w}$'s asserted probability that the training xs correspond to the training ys. It's reasonable to choose $\vec{w}$ so that this probability is maximal. This method is called **maximum likelihood estimation (MLE)**.

LOGISTIC MODELS —

The probability of a bunch of independent observations is the same as the product of probabilities of the observations. Taking logarithms turns products into more manageable sums. And — this is a historical convention — we further flip signs $\pm$ to turn maximization to minimization. After this translation, MLE with the logistic model means finding $\vec{w}$ that minimizes

$$\sum_i \log_2(1 + \exp(-y_i \vec{w} \cdot \vec{x}_i)) = \sum_i \text{softplus}(-y_i \vec{w} \cdot \vec{x}_i)$$

Here, softplus is our name for the function that sends z to $\log_2(1 + \exp(z))$.

mention convexity and convergence? show trajectory in weight space over time – see how certainty degree of freedom is no longer redundant? ("markov")

HUMBLE MODELS — Let's modify logistic classification to allow for *unknown unknowns*. We'll do this by allowing a classifier to allot probability mass not only among labels in $\mathcal{Y}$ but also to a special class $\star$ that means "no comment" or "alien input". A logistic classifier always sets $\mathbb{P}_{y|x}[\star|x] = 0$. Other probability models may put nonzero mass on "no comment"; different models give different learning programs. Here are four models we might consider:

	LOGISTIC	PERCEPTRON	SVM	GAUSS-GEN
$\mathbb{P}_{y\|x}[-1\|x]$	$u^-/(u^- + u^+)$	$u^- \cdot (u^- \wedge u^+)/2$	$u^- \cdot (u^-/e \wedge u^+)/2$	$u^- \cdot \text{bumps}$
$\mathbb{P}_{y\|x}[+1\|x]$	$u^+/(u^- + u^+)$	$u^+ \cdot (u^- \wedge u^+)/2$	$u^+ \cdot (u^- \wedge u^+/e)/2$	$u^+ \cdot \text{bumps}$
$\mathbb{P}_{y\|x}[\star\|x]$	$1 -$ above $= 0$	$1 -$ above	$1 -$ above	$1 -$ above
loss name	softplus$(\cdot)$	srelu$(\cdot)$	hinge$(\cdot)$	quad$(\cdot)$
formula	$\log_2(1 + e^{(\cdot)})$	$\max(1, \cdot) + 1$	$\max(1, \cdot + 1)$	??
update	$1/(1 + e^{+yd})$	step$(-yd)$	step$(1 - yd)$	??
outliers	vulnerable	robust	robust	vulnerable
inliers	sensitive	blind	sensitive	blind
humility	low	low	high	high

Table 1: **Four popular models for binary classification. Top rows:** Modeled chance given x that $y = +1$ or -1 or $\star$. We use $d = \vec{w} \cdot \vec{x}$, $u^{\pm} = e^{\pm d/2}$, $a \wedge b = \min(a, b)$ to save ink. And bumps $= (\phi(d-1) + \phi(d+1))/2$ with $\phi(\cdot)$ the standard normal density function. **Middle rows:** neg-log-likelihood losses. An SGD step looks like $\vec{w}_{t+1} = \vec{w}_t + (\eta \cdot \text{update} \cdot y\vec{x})$. **Bottom rows:** All models respond to misclassifications. But are they robust to well-classified outliers? Sensitive to well-classified inliers? Exercise: Fill in the ??s in the rightmost column of the table above.

MLE with the perceptron model, svm model, or gauss-gen model minimizes the same thing, but with $\text{srelu}(z) = \max(0, z) + 1$, $\text{hinge}(z) = \max(0, z + 1)$, or $\text{quad}(z) = \cdots$ instead of $\text{softplus}(z)$.°

← Other models we might try will induce other substitutes for softplus. E.g. $z \mapsto \text{hinge}(z)^2$ or $z \mapsto \text{avg}(z, \sqrt{z^2 + 4})$.

Two essential properties of softplus are that: (a) it is convex and (b) it upper bounds the step function. Note that srelu, hinge, and quad also enjoy these properties. Property (a) ensures that the optimization problem is relatively easy — under mild conditions, gradient descent is guaranteed to find a global minimum. By property (b), the total loss on a training set upper bounds the rate of erroneous classification on that training set. So loss is a *surrogate* for (in)accuracy: if the minimized loss is nearly zero, then the training accuracy is nearly 100%.

The perceptron satisfies (b) in a trivial way that yields a trivial bound of 100% on the error rate.

training behavior!! response to outliers support vectors

IDEAS IN STOCHASTIC GRADIENT DESCENT — learning rate as metric; robustness to 2 noise structures nesterov momentum decaying step size; termination conditions batch normalization

2. priors and generalization

A child's education should begin at least 100 years before [they are] born.
— oliver wendell holmes jr

ON OVERFITTING — $\mathbb{E}$ and $\max$ do not commute point estimates vs bayesian decision theory interpolation does not imply overfitting

LOG PRIORS AND BAYES — fill in computation and bases visual illustration of how choice of L2 dot product matters ℓ^p regularization; sparsity eye regularization example! TODO: rank as a prior (for multioutput models)

HIERARCHY, MIXTURES, TRANSFER — k-fold cross validation bayesian information criterion

ESTIMATING GENERALIZATION — k-fold cross validation dimension-based generalization bound bayesian information criterion

3. model selection

All human beings have three lives: public, private, and secret.
— gabriel garcìa marquez

4. generalization bounds

A foreign philosopher rides a train in Scotland. Looking out the window, they see a black sheep; they exclaim: "wow! at least one side of one sheep is black in Scotland!"
— unknown

DOT PRODUCTS AND GENERALIZATION —

HYPOTHESIS-CLASS-BASED BOUNDS — Suppose we are doing binary linear classification with N training samples of dimension d. Then with probability at least $1-\delta$ the gen gap is at most:

$$\sqrt{\frac{d\log(6N/d)+\log(4/\delta)}{N}}$$

For example, with $d = 16$ features and tolerance $\delta = 1/1000$, we can achieve a gen. gap of less than 5% once we have more than $N \approx 64000$ samples. This is pretty lousy. It's a worst case bound in the sense that it doesn't make any assumptions about how orderly or gnarly the data is.

If we normalize so that $\|x_i\| \leq R$ and we insist on classifiers with margin at least $0 < m \leq R$, then we may replace d by $\lceil 1+(R/m)^2 \rceil$ if we wish, so long as we count each margin-violator as a training error, even if it is correctly classified.

CHECK ABOVE!

In particular, with chance at least $1-\delta$:

$$\text{testing error} \leq \frac{(\text{number of support vectors})}{N} + \sqrt{\frac{\lceil 1+(R/m)^2\rceil \log(6N/\lceil 1+(R/m)^2\rceil)+\log(4/\delta)}{N}}$$

dimension/margin

DATA-BASED BOUNDS — Another way to estimate testing error is through **leave-one-out cross validation** (LOOCV). This requires sacrifice of a single training point in the sense that we need $N+1$ data points to do LOOCV for an algorithm that learns from N training points. The idea is that after training on the N, the testing-accuracy-of-hypotheses-learned-from-a-random-training-sample is unbiasedly estimated by the learned hypothesis's accuracy on the remaining data point. This is a very coarse, high-variance estimate. To address the variance, we can average over all $N+1$ choices° of which data point to remove from the training set. When the different estimates are sufficiently uncorrelated, this drastically reduces the variance of our estimate.

← In principle, LOOCV requires training our model $N+1$ many times; we'll soon see ways around this for the models we've talked about.

Our key to establishing sufficient un-correlation lies in *algorithmic stability*: the hypothesis shouldn't change too much as a function of small changes to the training set; thus, most of the variance in each LOOCV estimate is due to the testing points, which by assumption are independent.

If all xs, train or test, have length at most R, then we have that with chance at least $1-\delta$:

$$\mathbb{E}_{\mathcal{S}}\text{testing error} \leq \mathbb{E}_{\mathcal{S}}\text{LOOCV error} + \sqrt{\frac{1+6R^2/\lambda}{2N\delta}}$$

BAYES AND TESTING —

5. ideas in optimization

premature optimization is the root of all evil
— donald knuth

6. kernels enrich approximations

FEATURES AS PRE-PROCESSING —

ABSTRACTING TO DOT PRODUCTS —

KERNELIZED PERCEPTRON AND SVM —

KERNELIZED LOGISTIC REGRESSION —

... animals are divided into (a) those belonging to the emperor; (b) embalmed ones; (c) trained ones; (d) suckling pigs; (e) mermaids; (f) fabled ones; (g) stray dogs; (h) those included in this classification; (i) those that tremble as if they were mad; (j) innumerable ones; (k) those drawn with a very fine camel hair brush; (l) et cetera; (m) those that have just broken the vase; and (n) those that from afar look like flies.

— jorge luis borges

C. nonlinearities

0. fixed featurizations

Doing ensembles and shows is one thing, but being able to front a feature is totally different. ... there's something about ... a feature that's unique.
— michael b. jordan

SKETCHING —

—

—

SENSITIVITY ANALYSIS —

1. learned featurizations

IMAGINING THE SPACE OF FEATURE TUPLES — We'll focus on an architecture of the form

$$\hat{p}(y{=}{+}1 \,|\, x) \;=\; (\sigma_{1\times 1} \circ A_{1\times(h+1)} \circ f_{(h+1)\times h} \circ B_{h\times d})(x)$$

where A, B are linear maps with the specified (input $\times$ output) dimensions, where σ is the familiar sigmoid operation, and where f applies the leaky relu function elementwise and concatenates a 1:

$$f((v_i : 0 \le i < h)) = (1,) \mathbin{+\!\!+} (\mathrm{lrelu}(v_i) : 0 \le i < h) \qquad \mathrm{lrelu}(z) = \max(z/10, z)$$

We call h the **hidden dimension** of the model. Intuitively, $f \circ B$ re-featurizes the input to a form more linearly separable (by weight vector A).

THE FEATURIZATION LAYER'S LEARNING SIGNAL —

EXPRESSIVITY AND LOCAL MINIMA —

"REPRESENTER THEOREM" —

2. multiple layers

We can continue alternating learned linearities with fixed nonlinearities:

$$\begin{aligned}\hat{p}(y{=}{+}1 \,|\, x) \;=\; (\sigma_{1\times 1} \circ A_{1\times(h''+1)} \circ f_{(h''+1)\times h''} \circ B_{h''\times(h'+1)} \circ \\ f_{(h'+1)\times h'} \circ C_{h'\times(h+1)} \circ \\ f_{(h+1)\times h} \circ D_{h\times d})(x)\end{aligned}$$

FEATURE HIERARCHIES —

BOTTLENECKING —

HIGHWAYS —

3. architecture and wishful thinking

REPRESENTATION LEARNING —

4. architecture and symmetry

About to speak at [conference]. Spilled Coke on left leg of jeans, so poured some water on right leg so looks like the denim fade.
— tony hsieh

The key to success is failure.
— michael j. jordan

The virtue of maps, they show what can be done with limited space, they foresee that everything can happen therein.
— josé saramago

5. stochastic gradient descent

6. loss landscape shape

D. a weekend hackathon

This chapter is for folks who learn by doing. We've designed it to be accessible if it's the first chapter you read. Those who've already read Chapters A,B,C will recognize that our project in this chapter illustrates the concepts there.

a project

One beginning and one ending for a book was a thing I did not agree with. A good book may have three openings entirely dissimilar ...
— flann o'brien

AUTO EMAILER — Here's a fun project: let's make an email response drafter. The 'right' way to do this would be to use a pre-trained language model such as GPT-3. But we'll do it from scratch and get worse but still interesting results. To give the user a bit of control, let's allow them to specify the output's tone in the form of a thematic word like "coy" or "whimsical" or "pleading" or "authoritative". Specifically, let's use machine learning to create a stochastic function of type

$$\begin{aligned} h : (\text{set of paragraphs}) \times (\text{set of words}) &\to (\text{set of paragraphs}) \\ (\text{prompt}, \text{tone word}) &\mapsto \text{response} \end{aligned}$$

By *stochastic function*, I mean a randomized function that can give different outputs for the same input. By a *paragraph*, I mean a finite list whose every element is a word or one of a handful of special symbols such as punctuation or `END-OF-MESSAGE`. These lists will have length in $[0, 250)$. By a *word*, I mean any of the top ten thousand english words most common according to some standard catalogue. When distinguishing words, we'll forget capitalization (so `Cat` is the same as `cat`) but not conjugations/declensions (so `running` is not the same as `run`).

Where will we get training data for this? Well, let's take some of my personal emails (scrubbed for privacy). For tone words we'll be a bit more clever... FILL IN

FACTORIZATION —

ACQUIRING, SPLITTING, DATA —

VISUALIZING —

modeling

PRIORS —

—

—

—

tuning

—

—

—

—

deployment

—
—
—
—

E. structured inference

0. generative graphical models

... [to treat] complicated systems in simple ways[,] probability ... implements two principles[:] [the approximation of parts as independent] and [the abstraction of aspects as their averages].
— michael i. jordan

2. learning the parameters

3. hierarchy and mixtures

I like crossing the imaginary boundaries people set up between different fields — it's very refreshing.
— maryam mirzakhani

4. hierarchy and transfer

5. variational and sampling methods

Take a chance, won't you? Knock down the fences that divide.
— thurgood marshall

6. amortized inference

Reminding myself that I have a tailbone keeps me in check.
— tig notaro

F. reductions to supervision

G. (python) programming refresher

python setup

If I have not seen as far as others, it is because giants were standing on my shoulders.
— hal abelson

WHAT'S PYTHON? — Python is a programming language. Its heart is the **Python interpreter**, a computer program that takes a plain text file such as this two-liner° —

```
print('hello, world!')
print('cows and dogs are fluffy')
```

— and executes the instructions contained in that text file. We call these textual instructions are **source code**.

← The instruction `print` just displays some text onto our screen. For example, the first line of this program displays `hello, world!` onto our screen. This instruction doesn't rely on or activate any ink-on-paper printing machines.

The instructions have to be in a certain, extremely rigid format in order for the interpreter to understand and execute them. That's why they call Python a *language*: it has its own rigid grammar and its own limited vocabulary. If the interpreter encounters incorrectly formatted instructions — if even a single punctuation mark is missing or a — the interpreter will display a bit of text in addition to the word `Error` and immediately after abort its mission.°

← Adventure boldly when learning Python! It might feel catastrophic when you encounter an error and the interpreter 'dies'. But (unless you go out of your way to use special instructions we won't teach in class) these errors won't hurt your computer. There aren't any lasting effects. So **errors are hints, not penalties**. If you encounter an error, just modify your instructions to address that error, then run the interpreter again. Engage in a fast feedback cycle (*I'll try this... error... okay how about this... different error... hmm let's think...*) to learn to program well.

We'll use Python in 6.86x to instruct our computer to analyze data. The gigantic project of instructing a computer is a bit like teaching a person by mail. We never see them; we exchange only written words. They have never seen a horse, and yet we want to teach them to distinguish horses from zebras from donkeys from tapirs from rhinos. Though severely limited in their vocabulary and their ability to appreciate analogies and similarities, they are extraordinarily meticulous, patient, and efficient. That's what programming will be like.

At this point, you might have several questions:

Picking up the pen: How do I install and use the Python interpreter?

Writing sentences: What syntax rules must my instructions obey in order for the interpreter to understand that I want it do such-and-such task?

Composing essays: How do I organize large programs?

Teaching via mail: What instructions make the computer analyze data?

This and the next three sections address these four questions in turn.

WHICH THINGS WE'LL SET UP — Let's set up Python by installing the Python interpreter. Actually, I should say *a* Python interpreter: each of the many software tools we'll use in 6.86x has a zillion versions; it can get confusing tracking which versions coexist and which clash. We will use **Python version 3.8** throughout 6.86x.

Beyond the Python interpreter, there is an ecosystem of useful tools: machine learning modules that help us avoid reinventing the wheel; a rainbow of feature-rich text editors specialized for writing Python code; and a package manager called `conda` that eases the logistics nightmare of coordinating the versions we have of each tool.

SETUP ON WINDOWS — Mohamed, please fill this out (mention windows 10 and higher's 'linux subsystem?')

SETUP ON MACOS — Karene, please fill this out

SETUP ON LINUX — I'll assume we're working with Ubuntu 20.04. If you're on a different Linux, then similar steps should work — google search to figure out how.

CHECKING THE SETUP — Let's create a new plain text file containing this single line:°

```
print('hello, world!')
```

← This line contains **Python source code**. When we include Python source code in these notes, we will format it for readability, e.g. by making key parts of it bold. However, this depiction of source code is supposed to represent *plain text*.

We can name the file whatever we want — say, `greetings.py`. Then in your terminal (navigate to the same directory as the file and) enter this command:

```
python3 greetings.py
```

A new line should appear in your terminal:

```
hello, world!
```

After that line should be another shell prompt.

Now append three lines to the file `greetings.py` so that its contents look like:

```
print('hello, world!')
fahr = int(input('please enter a number... '))
celc = int((fahr - 32.0) * 5.0/9.0)
print('{} fahrenheit is roughly {} celcius'.format(fahr, celc))
```

Again, enter the following command in your terminal:

```
python3 greetings.py
```

Two new lines should appear in your terminal:

```
hello, world!
please enter a number...
```

Enter some number — say, 72; a new line should appear in your terminal so that the overall session looks something like:

```
hello, world!
please enter a number... 72
72 fahrenheit is roughly 22 celcius
```

After that line should be another shell prompt.

If you did the above but something different from the predicted lines appeared, that means something is amiss with Python setup. Please let us know in this case! We can try to help fix.

elements of programming

Displace one note and there would be diminishment, displace one phrase and the structure would fall.
— antonio salieri, on wolfgang mozart's music, as untruthfully portrayed in *Amadeus*

STATE AND CONTROL FLOW —

ROUTINES — Say there's an operation we do a lot. Maybe it's fahrenheit-to-celcius conversion. Instead of typing out the formula each time, we can *name* that formula and then invoke it by typing its name. Like so:

```
celc_from_fahr = lambda f: (f-32.)*(5./9.)
print(celc_from_fahr( 32.)) # prints   0.
print(celc_from_fahr(212.)) # prints 100.
```

Introducing such abstraction aligns the code's structure with our mental conceptual structure, yielding two related advantages: (a) it makes our source code easier to reason about (easier to read, easier to check correctness of, easier to change); (b) hierarchies of such formulae allows us to express concisely what otherwise would be an intractably long program.

To illustrate (a)

```
celc_from_fahr = lambda f: (f-32.)*(5./9.)
kelv_from_celc = lambda c: c - 273.15
kelv_from_fahr = lambda f: kelv_from_celc(celc_from_fahr(f))
kelv_from_kelv = lambda k: k
kelv_from_str  = lambda s: {'K':kelv_from_kelv,
                            'C':kelv_from_celc,
                            'F':kelv_from_fahr}[s[-1]](float(s[:-1]))
kelvs_from_csv = lambda ss: max(kelv_from_str(s.strip())
                                for s in ss.split(','))
#
avg        = lambda ts: sum(ts)/len(ts)
avg_square = lambda ts: average([t**2 for t in ts])
max_min_avg= lambda ts: (max(ts), min(ts), average(ts))
fancy_stats= lambda ts: (lambda mx,mn,av: {'max'     :mx,
                                           'min'     :mn,
                                           'avg':    :av,
                                           'variance':avg_square(ts)-av**2,
                                           'skew'    :av-(mn+mx)/2})(max_min_avg(ts))
#
csv = '50␣C,␣40␣F,␣200␣K,␣90␣F,␣80␣F,␣70␣F'
print(max_temp(kelvs_from_csv(csv)))
```

For more complicated routines, we can use this notation:

```
def celc_from_fahr(f):
  return (f-32.)*(5./9.)
```

We can use multiple lines and include assignments and control flow:

```
def celc_from_fahr(f):
  shifted = f-32.
  if shifted < 0:
    print('brrr!')
  return shifted*(5./9.)
```

Sometimes we want to

DATA IN AGGREGATE — lists and numpy arrays dictionaries comprehensions functional idioms

INPUT/OUTPUT — print formatting and flushing basic string manipulations (strip, join, etc) file io command line arguments example: read csv random generation and seeds??

how not to invent the wheel

```
def get_random_number():
    return 4 # chosen by a fair dice roll
             # guaranteed to be random
```

— randall munroe, translated by sam to Python

OS, NUMPY, MATPLOTLIB, ETC — matplotlib.plot numpy os package managers etc

ARRAYS AKA TENSORS —

PYTORCH IDIOMS —

GIT FOR VERSION CONTROL —

how to increase confidence in correctness

So little of what could happen does happen.
— salvador dalí

DEDUCTION: —

INDUCTION: ISOLATION AND BISECTION —

PERIODIC TABLE OF COMMON ERRORS —

H. math refresher

types and functions

This section concerns *language*. Unless you're a nerd like me, this is probably the most boring section in all these notes.

SIGNS AND REFERENTS —

THINGS AND THEIR TYPES — We often want to express that *the thing we've named "x" belongs to the type we've named "$\mathcal{X}$"*. The following expresses that statement in shorthand:

$$x : \mathcal{X}$$

Number types.

Function types.

NEW TYPES FROM OLD —

If $\mathcal{X}$ and $\mathcal{Y}$ are types, then so are $\mathcal{X}+\mathcal{Y}$, $\mathcal{X}\times\mathcal{Y}$, and $\mathcal{X}\to\mathcal{Y}$. We call them *sum types*, *product types*, and *function types*, respectively.

The sum and product types enjoy a pleasant symmetry:

$$\begin{array}{cc} \text{left} : \mathcal{X} \to (\mathcal{X}+\mathcal{Y}) & \text{first} : (\mathcal{X}\times\mathcal{Y}) \to \mathcal{X} \\ \text{right} : \mathcal{Y} \to (\mathcal{X}+\mathcal{Y}) & \text{second} : (\mathcal{X}\times\mathcal{Y}) \to \mathcal{Y} \\ \text{glue} : (\mathcal{X}\to\mathcal{Z}) \to (\mathcal{Y}\to\mathcal{Z}) \to ((\mathcal{X}+\mathcal{Y})\to\mathcal{Z}) & \text{pair} : (\mathcal{W}\to\mathcal{X}) \to (\mathcal{W}\to\mathcal{Y}) \to (\mathcal{W}\to(\mathcal{X}\times\mathcal{Y})) \end{array}$$

SAMENESS, APPROXIMATION, ABSTRACTION —

linear algebra and approximation

Stand firm in your refusal to remain conscious during algebra. In real life, I assure you, there is no such thing as algebra.
— fran lebowitz

Linear algebra is the part of geometry that focuses on when a point is the origin, when a 'line' is a straight, and when two straight lines are parallel. Linear algebra thus helps us deal with the preceding pictures mathematically and automatically. The concept of 'straight lines' gives a simple, flexible model for extrapolation from known points to unknown points. That is intuitively why linear algebra will be crucial at every stage of 6.86x.

COLUMN VECTORS AND ROW VECTORS — The elements of linear algebra are **column vectors** and **row vectors**.° We have a set V of "column vectors". We're allowed to find V's zero vector and to add or scale vectors in V to get other vectors in V. V is the primary object we hold in our mind; perhaps, if we're doing image classification, then each column vector represents a photograph. We use the word "space" or "vector space" when talking about V to emphasize that we'd like to exploit visual intuitions when analyzing V. In short: we imagine each column vector as a point in space, or, if we like, as an arrow from the zero vector to that point.

← Though we represent the two similarly in a computer's memory, they have different geometric meanings. We save much anguish by remembering the difference.

Now, associated to V is the set of "row vectors". Under the hood, a row vector is a linear function from V to the real numbers $\mathbb{R}$. We imagine each row vector not as an arrow but as a "linear" heat map or altitude map that assigns to each point in V a numeric "intensity". We can visualize a row vector the same way makers of geographic maps do: using contours for where in V the row vector attains values $\cdots, -2, -1, 0, +1, +2, \cdots$. These will be a bunch of

uniformly spaced parallel "planes". The spacing and orientation of the planes depends on and determines the row vector. In short, we imagine each row vector as a collection of parallel planes in space.

Informally: a column vector is a noun or thing whereas a row vector is a adjective or property. The degree to which a property holds on a thing (or a description is true of a thing) is gotten by evaluating the row vector on the column vector — remember, a row vector is a function, so we can do this evaluation. Geometrically, if a row vector is a bunch of planes and a column vector is an arrow, the two evaluate to a number: the number of planes that the arrow pierces. Intuitively, an example of a column vector might be "this particular photo of my pet cow"; an example of a row vector might be "redness of the left half (of the input photo)". If we evaluate this row vector on this column vector, then we get a number indicating how intensely true it is that the left half of that particular photo is red.

We can draw an analogy with syntax vs semantics. This pair of concepts pops up in linguistics, philosophy, circuit engineering, quantum physics, and more, but all we need to know is that: semantics is about things while syntax is about descriptions of things. The two concepts relate in that, given a set of things, we can ask for the set of all descriptions that hold true for all those things simultaneously. And if we have a set of descriptions, we can ask for the set of all things that satisfy all those descriptions simultaneously. These two concepts stand in formal opposition in the sense that: if we have a set of things and make it bigger, then the set of descriptions that apply becomes smaller. And vice versa. Then a column vector is an object of semantics. And a row vector is an object of syntax.

INNER PRODUCTS — Now, here is the key point: the engine behind generalization in machine learning (at least, the machine learning we'll do in Units 1 and 2; and less visibly but still truly in more than half of each of Units 3,4,5) is the ability to translate between things and properties. If "my pet cow" is a thing, then "similar to my pet cow" is a property. The whole project of machine learning is to define and implement this word "similar to". When we define and implement well, our programs can generalize successfully from training examples to new, previously unseen situations, since they will be able to see which of the training examples the new situations are similar to. Since "similar to" transforms things to properties, the linear algebra math behind "similar to" is a function from column vectors to row vectors. This brings us to...

... inner products, aka kernels. An inner product is just a fancy word for a (linear) function from column vectors to row vectors. We actually demand that this linear function has two nice properties: FIRST, it should have an inverse. That is, it ought to be a two way bridge between column vectors and row vectors, allowing us to translate things to descriptions and vice versa. SECOND, it should be symmetric. This means that if we have two things, say "my pet cow" and "my pet raccoon", then the degree to which "my pet raccoon" has the property "is similar to my pet cow" ought to match the degree to which "my pet cow" has the property "is similar to my pet raccoon". Any invertible, symmetric, linear function from column vectors to row vectors is called an inner product. Kernel is a synonym here.

Beware: the same word, "kernel", has different meanings depending on context.

There are generally infinitely many inner products. Which one we choose changes the generalization properties of our machine learning program. Practically, if we are doing machine learning in a concrete situation, then we want to choose an inner product that reflects our human intuition and experience and domain knowledge about the right notion of "similarity" in that situation.

Any inner product P from column vectors to row vectors induces notions of length and angle. We just define a column vector v's length by $\sqrt{P(v)(v)}$. Call that quantity $\|v\|$. And we define the angle $\alpha(v, w)$ between two non-zero column

vectors v,w by $P(v)(w) = \|v\| \cdot \|w\| \cdot \cos \alpha(v, w)$. We make these definitions so as to match the Pythagorean theorem from plane geometry. So once we choose which inner product we'll use (out of the infinitely many choices), we get the concepts of euclidean geometry for free. Immediately following from that definition of angle, we get that if two column vectors have vanishing inner product (i.e., if $P(v)(w) = 0$), then those vectors are at right angles (i.e. $\alpha(v)(w) = \pi/2$).

You can google up proofs of the Pythagorean theorem (many are quick and beautiful) if you wish to dig deeper.

Now, sometimes (but most of the time not), we are blessed in that we know more about our situation than just the space V of things. Specifically, V might come with a canonical basis. This just means that V comes marked with "the right" axes with respect to which we ought to analyze vectors in V. In this fortunate case, there is also a canonical inner product. It's called dot product. Again, I want to emphasize that a plain vector space doesn't come with a dot product. We need a basis for that.

The dot product is defined as follows. Say there are D axes and that the "unit" vectors along each axis (aka the basis vector) are named $(v)i : 0 \leq i < D)$. Then we define $P(v_j)(v_i) = 1 \text{if} i = j \text{else} 0$ — the 1 expresses that each "unit" basis vector ought to have length 1. The 0 expresses that different basis vectors ought to be at right angles to each other. This determines P on all inputs, by linearity: $P(\sum c'_j v_j)(\sum c_i v_i) = \sum_i c'_i c_i$ where c'_j, c_i are numbers. In short: given a basis, there is a unique inner product such that those basis elements all have length 1 and are at right angles to each other. We call that inner product the dot product.

FILL IN LINEAR DECISION BOUNDARY! (remark on featurization and argmax nonlinearities)

We may **evaluate** a row vector on a column vector. FILL IN A **dot product** is a way of translating between row and column vectors. FILL IN: DISCUSS GENERALIZATION; (DISCUSS ANGLE, TOO)

LINEAR MAPS —

SINGULAR VALUE DECOMPOSITION —

probability and generalization

Can I just say Chris for one moment that I have a new theory about the brontosaurus. ... This theory goes as follows and begins now. All brontosauruses are thin at one end, much, much thicker in the middle and then thin again at the far end. That is my theory, it is mine, and belongs to me and I own it, and what it is too.
— john cleese

BELIEF AND BAYES —

THE KEY ABSTRACTION: AVERAGES —

THE KEY APPROXIMATION: INDEPENDENCE —

UNIFORM CONCENTRATION —

calculus and optimization

The self is not something ready-made, but something in continuous formation through choice of action.
— john dewey