A. prologue

what is learning?

By the end of this section, you'll be able to
- recognize whether a learning task fits the paradigm of *learning from examples* and whether it's *supervised* or *unsupervised*.
- identify within a completed learning-from-examples project: the *training inputs(outputs)*, *testing inputs(outputs)*, *hypothesis class*, *learned hypothesis*; and describe which parts depend on which.

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**.°

← Food For Thought: What's something you've learned by instruction? By example? By reinforcement? In unit 5 we'll see that learning by example unlocks 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. We seek a program that, from a list of examples of prompts and matching answers, determines an underlying pattern. Our program is a success if this pattern accurately predicts answers for new, unseen prompts. We often define our program as a search, over some class $\mathcal{H}$ of candidate patterns (jargon: **hypotheses**), to maximize some notion of "intrinsic-plausibility plus goodness-of-fit-to-the-examples".

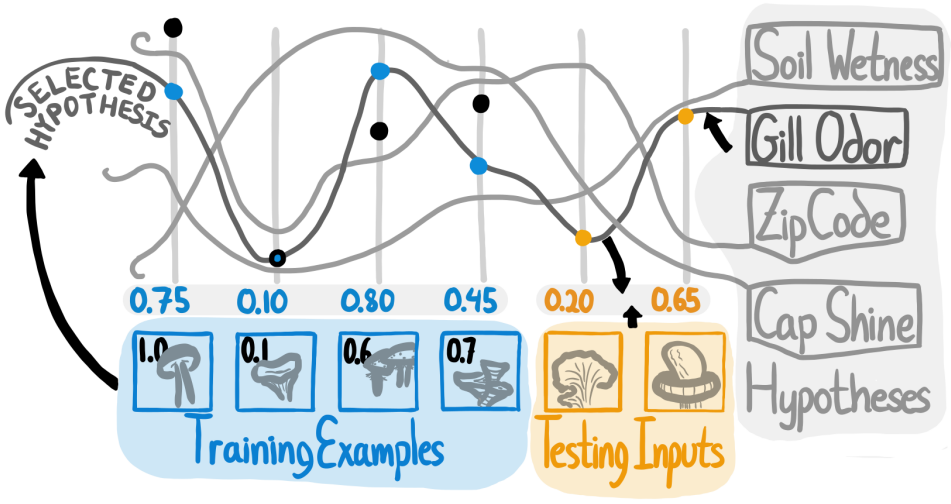

Figure 1: **Predicting mushrooms' poisons.** Our learning program selects from a class of hypotheses (gray blob) a plausible hypothesis that well fits (**blue dots** are close to **black dots**) a given list of poison-labeled mushrooms (blue blob). Evaluating the selected hypothesis on new mushrooms, we predict the corresponding poison levels (orange numbers).

The arrows show dataflow: how the hypothesis class and the mushroom+poisonlevel examples determine one hypothesis, which, together with new mushrooms, determines predicted poison levels. Selecting a hypothesis is called **learning**; predicting unseen poison levels, **inference**. The examples we learn from are **training data**; the new mushrooms and their true poison levels are **testing data**.

For example, say we want to predict poison levels (answers) of mushrooms (prompts). Among our hypotheses,° the GillOdor hypothesis fits the examples well: it guesses poison levels close to the truth. So the program selects GillOdor.

'Wait!', you say, *'doesn't Zipcode fit the example data more closely than GillOdor?'*. Yes. But a poison-zipcode proportionality is implausible: we'd need more evidence before believing Zipcode. We can easily make many oddball hypotheses; by chance some may fit our data well, but they probably won't predict well! Thus "intrinsic plausibility" and "goodness-of-fit-to-data" *both* play a role in learning.°

In practice we'll think of each hypothesis as mapping mushrooms to *distributions* over poison levels; then its "goodness-of-fit-to-data" is simply the chance it allots to the data.° We'll also use huge $\mathcal{H}$s: we'll *combine* mushroom features (wetness, odor, and shine) to make more hypotheses such as $(1.0 \cdot \text{GillOdor} - 0.2 \cdot \text{CapShine})$.° Since we can't compute "goodness-of-fit" for so many hypotheses, we'll guess a hypothesis then repeatedly nudge it up the "goodness-of-fit" *slope*.°

← We choose four hypotheses: respectively, that a mushroom's poison level is close to:
— *its ambient soil's percent water by weight*;
— *its gills' odor level, in kilo-Scoville units*;
— *its zipcode (divided by 100000)*;
— *the fraction of visible light its cap reflects*.

← We choose those two notions (and our $\mathcal{H}$) based on **domain knowledge**. This design process is an art; we'll study some rules of thumb.

← That's why we'll need **probability**.

← That's why we'll need **linear algebra**.

← That's why we'll need **derivatives**.

SUPERVISED LEARNING — We'll soon allow uncertainty by letting patterns map prompts to *distributions* over answers. Even if there is only one prompt — say, "*produce a beautiful melody*" — we may seek to learn the complicated distribution over answers, e.g. to generate a diversity of apt answers. Such **unsupervised learning** concerns output structure. By contrast, **supervised learning** (our main subject), concerns the input-output relation; it's interesting when there are many possible prompts. Both involve learning from examples; the distinction is no more firm than that between sandwiches and hotdogs, but the words are good to know.

a tiny example: classifying handwritten digits

By the end of this section, you'll be able to

- write a (simple and inefficient) image classifying ML program
- visualize data as lying in feature space; visualize hypotheses as functions defined on feature space; and visualize the class of all hypotheses within weight space

MEETING THE DATA — Say we want to classify handwritten digits. In symbols: we'll map $\mathcal{X}$ to $\mathcal{Y}$ with $\mathcal{X} = \{\text{grayscale } 28\times28\text{-pixel images}\}$, $\mathcal{Y} = \{1, 3\}$. 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 $x \in \mathcal{X}$.

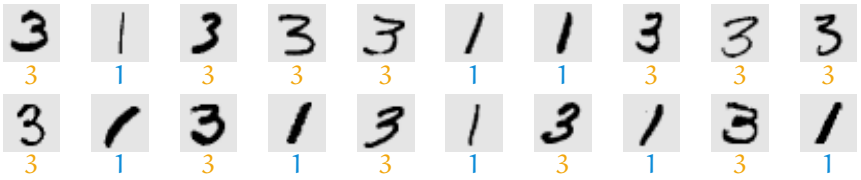

Figure 2: Twenty example pairs. Each photo x is a 28×28 grid of numbers representing pixel intensities. The light gray background has intensity 0.0; the blackest pixels, intensity 1.0. Below each photo x we display the corresponding label y: either $y = 1$ or $y = 3$. We'll adhere to this color code 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 from the top) followed by their column number (counting from the left). So location $(0, 0)$ is the upper left corner pixel; $(27, 0)$, the lower left corner pixel.
Food For Thought: Where is location $(0, 27)$? Which way is $(14, 14)$ off-center?

To get to know the data, let's wonder how we'd hand-code a classifier (worry not: soon we'll do this more automatically). We want to complete the code

```
def hand_coded_predict(x):
  return 3 if condition(x) else 1
```

Well, 3s tend to have more ink than than 1s — should `condition` threshold by the photo's brightness? Or: 1s and 3s tend to have different widths — should `condition` threshold by the photo's dark part's width?

To make this precise, let's define a photo's *brightness* as 1.0 minus its average pixel brightness; its *width* as the standard deviation of the column index of its dark pixels. Such functions from inputs in $\mathcal{X}$ to numbers are called **features**.

```
SIDE = 28
def brightness(x):  return 1. - np.mean(x)
def width(x):       return np.std([col for col in range(SIDE)
                                   for row in range(SIDE)
                                   if 0.5 < x[row][col]  ])/(SIDE/2.0)
# (we normalized width by SIDE/2.0 so that it lies within [0., 1.])
```

So we can threshold by brightness or by width. But this isn't very satisfying, since sometimes there are especially dark 1s or thin 3s. Aha! Let's use *both* features: 3s are darker than 1s *even relative to their width*. Inspecting the training data, we see that a line through the origin of slope 4 roughly separates the two classes. So let's threshold by a combination like `-1*brightness(x)+4*width(x)`:

```
def condition(x):
  return -1*brightness(x)+4*width(x) > 0
```

Intuitively, the formula $-1 \cdot \text{brightness} + 4 \cdot \text{width}$ we invented is a measure of *threeness*: if it's positive, we predict $y = 3$. Otherwise, we predict $y = 1$.
Food For Thought: What further features might help us separate digits 1 from 3?

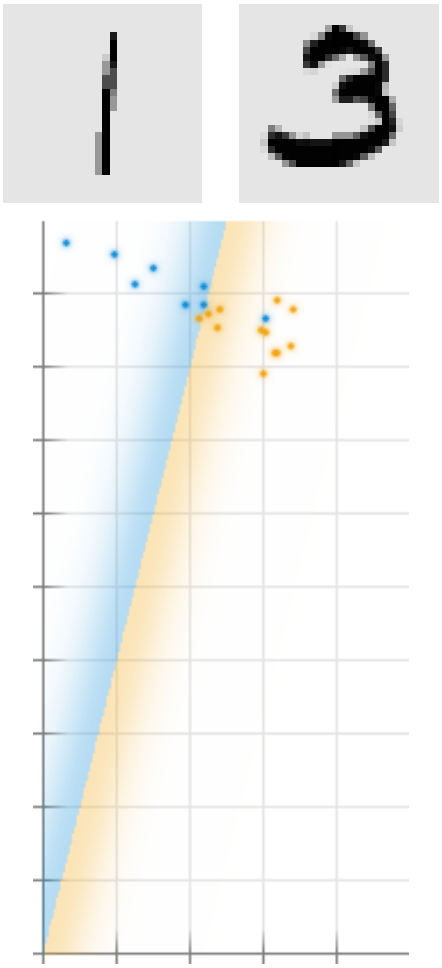

Figure 3: **Featurized training data.** Our $N = 20$ many training examples, viewed in the brightness-width plane. The vertical *brightness* axis ranges $[0.0, 1.0]$; the horizontal *width* axis ranges $[0.0, 0.5]$. The origin is at the lower left. Orange dots represent $y = 3$ examples; blue dot, $y = 1$ examples. We eyeballed the line $-1 \cdot \text{brightness} + 4 \cdot \text{width} = 0$ to separate the two kinds of examples.

CANDIDATE PATTERNS — We can generalize the hand-coded hypothesis from the previous passage to other coefficients besides $-1 \cdot \text{brightness}(x) + 4 \cdot \text{width}(x)$. We let our set $\mathcal{H}$ of candidate patterns contain all "linear hypotheses" $f_{a,b}$ defined by:

$$f_{a,b}(x) = 3 \text{ if } a \cdot \text{brightness}(x) + b \cdot \text{width}(x) > 0 \text{ else } 1$$

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 3 if a*brightness(x) + b*width(x) > 0 else 1
```

The brightness-width plane is called **feature space**: its points represent inputs x in terms of chosen features (here, brightness and width). The (a, b) plane is called **weight space**: its points represent linear hypotheses h in terms of the coefficients — or **weights** — h places on each feature (e.g. $a = -1$ on brightness and $b = +4$ on width).

Food For Thought: Which of Fig. 4's 3 hypotheses best predicts training data?

Food For Thought: What (a, b) pairs might have produced Fig. 4 shows 3 hypotheses? Can you determine (a, b) for sure, or is there ambiguity (i.e., can multiple (a, b) pairs make exactly the same predictions in brightness-width space)?

OPTIMIZATION — Let's write a program to automatically find hypothesis $h = (a, b)$ from the training data. We want to 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 just loop over a bunch (a, b)s — say, all integer pairs in $[-99, +99]$ — and pick one that misclassifies the least training examples:

```
def is_correct(x,y,a,b):
  return 1.0 if predict(x,a,b)==y else 0.0
def accuracy_on(examples,a,b):
  return np.mean(is_correct(x,y,a,b) for x,y in examples)
def best_hypothesis():
  # returns a pair (accuracy, hypothesis)
  return max((accuracy_on(training_data, a, b), (a,b))
             for a in np.arange(-99,+100)
             for b in np.arange(-99,+100)            )
```

Fed our $N = 20$ training examples, the loop finds $(a, b) = (-20, +83)$ as a minimizer of **training error**, i.e., of the fraction of training examples misclassified. It misclassifies only 10% of training examples. Yet the same hypothesis misclassifies a greater fraction — 17% — 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 our second try on a practice exam (after we've reviewed our mistakes) versus our score on a real exam (where we don't know the questions beforehand and aren't allowed to change our answers once we get our grades back).

Food For Thought: In the (a, b) plane shaded by training error, we see two 'cones', one dark and one light. They lie geometrically opposite to each other — why?

Food For Thought: Sketch $f_{a,b}$'s error on $N = 1$ example as a function of (a, b).

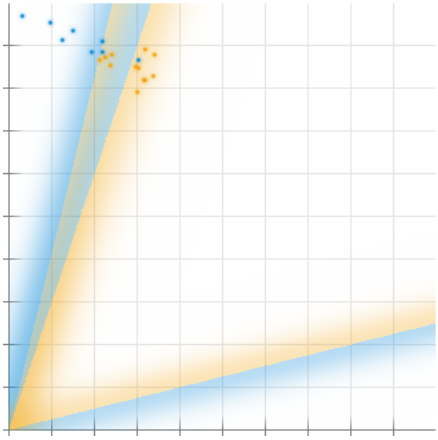

Figure 4: **Hypotheses differ in training accuracy: feature space.** 3 hypotheses classify training data in the brightness-width plane (axes range $[0, 1.0]$). Glowing colors distinguish a hypothesis' 1 and 3 sides. For instance, the bottom-most line classifies all the training points as 3s.

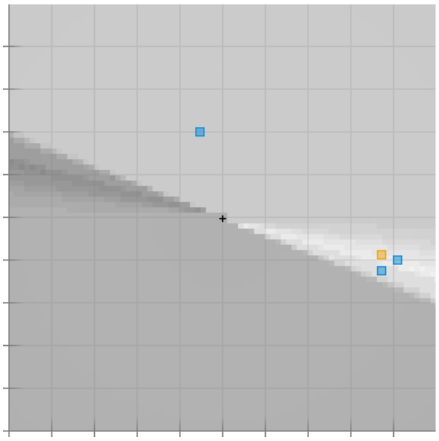

Figure 5: **Hypotheses differ in training accuracy: weight space.** We visualize $\mathcal{H}$ as the (a, b)-plane (axes range $[-99, +99]$). Each point determines a whole line in the brightness-width plane. Shading shows training error: darker points misclassify more training examples. The least shaded, most training-accurate hypothesis is $(-20, 83)$: the rightmost of the 3 blue squares. The orange square is the hypothesis that best fits our unseen testing data.

Food For Thought: Suppose Fig. 4's 3 hypotheses arose from the 3 blue squares shown here. Which hypothesis arose from which square? **Caution**: the colors in the two Figures on this page represent unrelated distinctions!

how well did we do? analyzing our error

By the end of this section, you'll be able to

- automatically compute training and testing misclassification errors and describe their conceptual difference.
- explain how the problem of achieving low testing error decomposes into the three problems of achieving low *generalization*, *optimization*, and *approximation* errors.

ERROR ANALYSIS — Intuitively, our testing error of 17% 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 $\mathcal{H}$; and **(c)** the failure of our hypothesis set $\mathcal{H}$ to contain "the true" pattern.

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

We can see generalization error when we plot testing data in the brightness-width plane. The hypotheses $h = (20, 83)$ that we selected based on the training in the brightness-width plane misclassifies many testing points. we see many misclassified points. Whereas h misclassifies only 10% of the training data, it misclassifies 17% of the testing data. This illustrates generalization error.

In our plot of the (a, b) plane, the blue square is the hypothesis h (in $\mathcal{H}$) that best fits the training data. The orange square is the hypothesis (in $\mathcal{H}$) that best fits the testing data. But even the latter seems suboptimal, since $\mathcal{H}$ only includes lines through the origin while it seems we want a line — or curve — that hits higher up on the brightness axis. This illustrates approximation error.°

← To define *approximation error*, we need to specify whether the 'truth' we want to approximate is the training or the testing data. Either way we get a useful concept. In this paragraph we're talking about approximating *testing* data; but in our notes overall we'll focus on the concept of error in approximating *training* data.

Optimization error is best seen by plotting training rather than testing data. It measures the failure of our selected hypothesis h to minimize training error — i.e., the failure of the blue square to lie in a least shaded point in the (a, b) plane, when we shade according to training error.

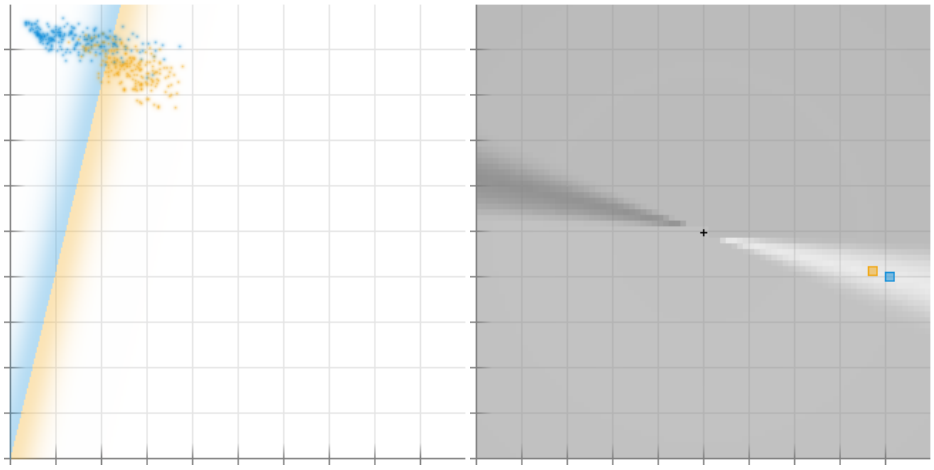

Figure 6: **Testing error visualized two ways.**. — **Left: in feature space.** The hypotheses $h = (20, 83)$ that we selected based on the training set classifies testing data in the brightness-width plane; glowing colors distinguish a hypothesis' 1 and 3 sides. Axes range $[0, 1.0]$. — **Right: in weight space.** Each point in the (a, b) plane represents a hypothesis; darker regions misclassify a greater fraction of testing data. Axes range $[-99, +99]$.

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: $\approx$ 10%. The approximation error is so high because our straight lines are *too simple*: brightness and width lose useful information and the "true" boundary between digits — even training — may be curved. Finally, our testing error $\approx$ 17% exceeds our training error. We thus suffer a generalization error of $\approx$ 7%: we *didn't perfectly extrapolate* from training to testing situations. In 6.86x we'll address all three italicized issues.

Food For Thought: why is generalization error usually positive?

FORMALISM — Here's how we can describe learning and our error decomposition in symbols. ← VERY OPTIONAL PASSAGE
Draw training examples $\mathcal{S} : (\mathcal{X} \times \mathcal{Y})^N$ from nature's distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$. A hypothesis $f : \mathcal{X} \to \mathcal{Y}$ 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 $\text{tst}(f)$.

So 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".