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.

A. prologue

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

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 Unit 5 of 6.86x, 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".

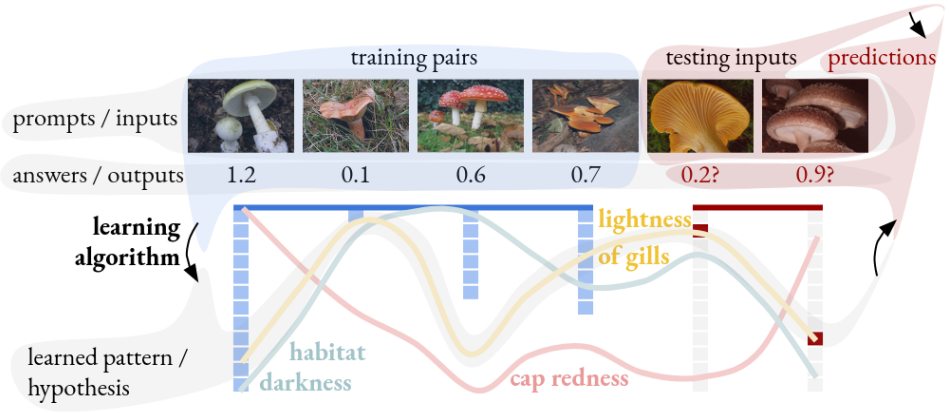


Figure 1: A program that learns to predict mushrooms' poison levels: first takes a list of labeled mushrooms as input (blue blob); searches through candidate patterns (here, the wiggly curves labeled `lightness of gills`, `habitat darkness`, and `cap redness`); and returns the pattern that best fits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

The three bold black arrows show the flow of data from training examples to a learned pattern; and from that pattern together with testing prompts to 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 distributions* over answers. Even if the prompt is always the same — say, "produce a beautiful melody", we may seek to understand the complicated distribution over answers. We might regard our program a success if it can generate a variety of good answers. So-called **unsupervised learning** focuses in that way on output structure. By contrast, **supervised learning** (our subject in Unit 1), focuses on the input-output relation; it's interesting when the space of prompts is large.

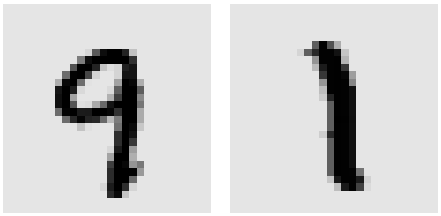
a tiny example: handwritten digit classification

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. Our convention will be to name these 28×28 grid locations by the number of their row (counting starting from the top) and then of their column (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?

Exercise: What pixel intensities do these two photos have at these two locations?

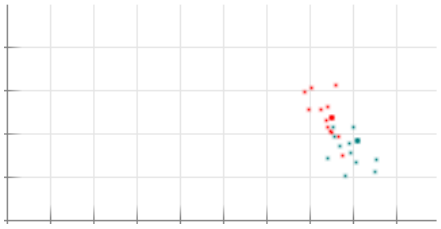


Figure 3: Our size-N = 25 set of training examples, viewed in the darkness-height plane. The vertical *darkness* axis ranges $[0.0, 0.25]$; the horizontal *height* axis ranges $[0.0, 0.5]$. The origin is at the lower left. Each cyan dot represents a $y = 1$ example; each red dot, a $y = 9$ one. The big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

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?

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)
```

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 `2*darkness(x)-height(x)>0` for° our condition.

```
def hand_coded_predict(x):
  return 9 if 2*darkness(x)-height(x)>0 else 1
```

← That factor of 2 comes from our observation that darkness tends to be bigger than height. We'll soon see that this eyeballed slope doesn't work well. It's better to tune by machine.

Exercise: Beyond height and darkness, what features do you think 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: Fig **??**: 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))
```

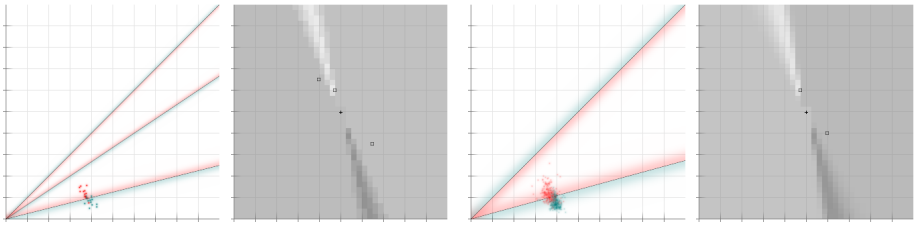


Figure 4: **Training (■■□□) and testing (□□■■).** 3 hypotheses classify training data in the darkness-height plane (■□□□); glowing colors distinguish a hypothesis' 1 and 9 sides. Each point in the (a, b) plane (□■□□) represents a hypothesis; darker regions misclassify a greater fraction of training data. 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?

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 subsection'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?