mlentary: basics of machine learning

These are optional notes for 6.86x. You can do all your assignments without these notes. These notes are here as a study aid. They are terse and don't cover all our topics. I'm happy to answer questions about the notes on Piazza. If you want to help improve these notes, ask me; I'll be happy to list your name among the contributors to these notes!

Clickable Table of Contents

A. prologue

what is learning?

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 part 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".

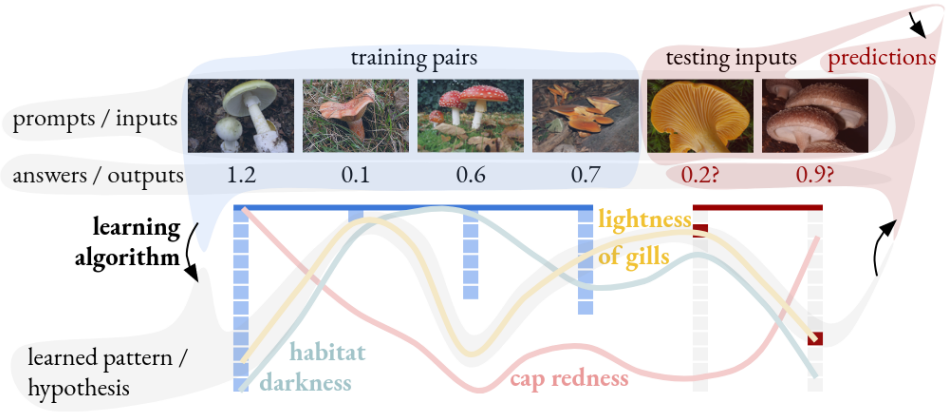


Figure 1: A program that learns to predict mushrooms' poison levels: first takes a list of labeled mushrooms as input (blue blob); searches through candidate patterns (here, the wiggly curves labeled `lightness of gills`, `habitat darkness`, and `cap redness`); and returns the pattern that best fits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

our first learning algorithm

a tiny example: handwritten digit classification

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

MEETING THE DATA — $\mathcal{X} = \{\text{grayscale } 28 \times 28\text{-pixel images}\}$; $\mathcal{Y} = \{1, 9\}$. 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 $x \in \mathcal{X}$.

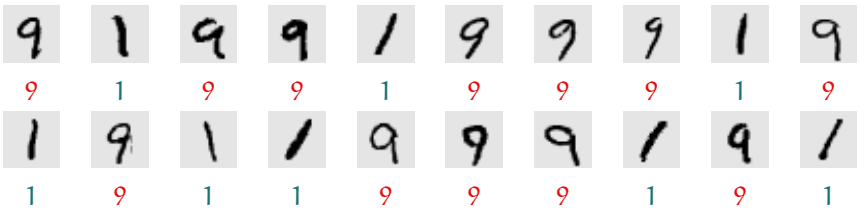


Figure 2: Twenty example pairs. Each photo x is a 28×28 grid of numbers representing pixel intensities. The light gray background has intensity 0.0; the blackest pixels, intensity 1.0. Below each photo x we display the corresponding label y: either $y = 1$ or $y = 9$. We'll adhere to this color code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

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**.

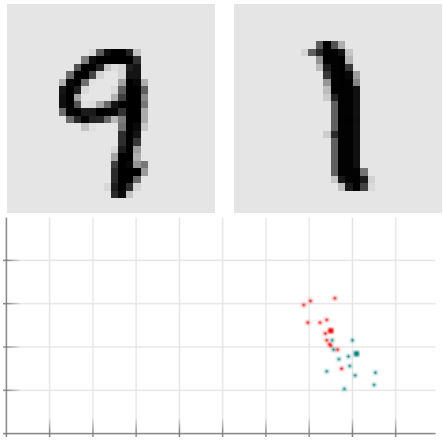


Figure 3: Our size-N = 25 set of training examples, viewed in the darkness-height plane. The vertical *darkness* axis ranges $[0.0, 0.25]$; the horizontal *height* axis ranges $[0.0, 0.5]$. The origin is at the lower left. Each cyan dot represents a $y = 1$ example; each red dot, a $y = 9$ one. The big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

```
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))
```

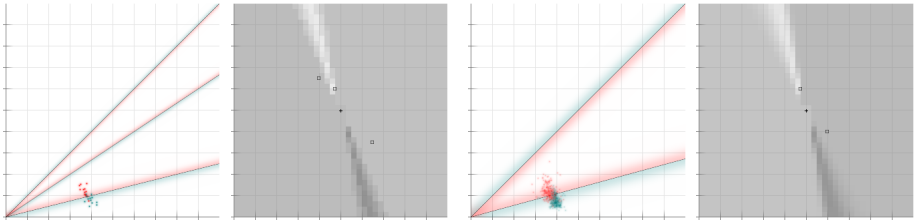


Figure 4: **Training (■■□□) and testing (□□■■).** 3 hypotheses classify training data in the darkness-height plane (■□□□); glowing colors distinguish a hypothesis' 1 and 9 sides. Each point in the (a, b) plane (□■□□) represents a hypothesis; darker regions misclassify a greater fraction of training data.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

how well did we do? (3 kinds of error : opt, approx, gen)

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 $\mathcal{H}$; and **(c)** 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?

FORMALISM — Here's how we can describe learning and our error decomposition in symbols. 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 $\text{tst}(f)$.

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".

how can we do better? (survey of rest of notes)

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. auto-predict by fitting lines to examples (unit 1)

linear approximation

B. linear classification

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. There, we 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. We judge a featurization not in a vacuum but 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

Here are three featurizations ideas that might inspire you in your own projects.

Hardcoded coordinate transforms. **The bias trick**.[◦] We send $x \mapsto (1, x)$, thus increasing dimension by one.

←a.k.a. **projectivization**

Generalizing the above, we can use **polynomial features**:

$$x \mapsto (x^{\otimes 0}, x^{\otimes 1}, \cdots, x^{\otimes d})$$

By $x^{\otimes d}$, we mean a multiaxis array a with d axes and defined (in an example where $d = 3$) by

$$a_{42,686,0} = x_{42} x_{686} x_0 \in \mathbb{R}$$

Note that this featurization is redundant since permuting a's axes doesn't change a.

coordinate transforms — e.g. arctan.

Encoding the discrete. **one-hot**

Binning

Data-dependent transforms. **Quantilization**

Landmarks

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

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?)

MORE CLASSES AND BEYOND — 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	$f_{(\vec{w}_i : 0 \le i < k)}(\vec{x}) = (\vec{w}_i \cdot \vec{x} : 0 \le i < k)$
multi-class soft classification	$f_{(\vec{w}_i : 0 \le i < k)}(\vec{x}) = \text{normalize}(\exp(\vec{w}_i \cdot \vec{x}) : 0 \le i < k)$

A key ideas here is *symmetry between classes*. Previously, we interpreted positive decision function values as meaning one class and negative values as meaning the other. This is asymmetrical — why should one class count as positive?! — and does not immediately generalize to multiple classes. Instead, let's do two-class classification by producing *two* values: the degree to which a point is of one class and the degree to which a point is of the other class. Then we'll classify according to which value is *greater*. This is redundant because we could add $+42$ to both values and get the same answer for all inputs. Redundancy is often the cost of symmetry.

More generally, if we have three classes *ostrich*, *emu*, and *roc*, then we compute three linear functions representing *ostrich-ness*, *emu-osity*, and *roc-iness*. And we

return whichever class has a greatest value. This explains the *multi-class hard classification*.

For *multi-output regression*, let's just skip the argmax.

For *multi-output soft classification*, we want to report probabilities instead of general real-valued scores. Probabilities ought to be non-negative and ought to sum to one. A nice way to turn general numbers to non-negative (in fact, positive) ones is to apply exp. A nice way to get positive numbers to sum to 1 is to divide by their sum. This leads us to **softmax**:

$$\text{softmax}(\omega_k \ : \ 0 \leq k < K) = \left(\frac{\exp(\omega_k)}{\sum_{k'} \exp(\omega_{k'})} \ : \ 0 \leq k < K \right)$$

TODO: show pictures of 3 classes, 4 classes (e.g. digits 0,1,8,9) TODO: talk about structured (trees/sequences/etc) output!

VISUALIZING HIGH DIMENSIONAL SPACES — FILL IN

iterative optimization

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).

Derivatives measure how changing a parameter a bit affects a result. So let's 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 seems 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	robust-ish	robust	robust	vulnerable
inliers	sensitive	blind	sensitive	blind-ish
humility	low	low	high-ish	high
acc bnd	good	bad	good	good-ish

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

PICTURES OF TRAINING —

test vs train curves: overfitting

random featurization: double descent

2. priors and generalization

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

priors and generalization

ON OVERFITTING — In the Bayesian framework, we optimistically assume that our model is "correct" and that the "true" posterior over parameters is

$$p(w|y;x) = p(y|w;x)p(w)/Z_x$$

a normalized product of likelihood and prior. Therefore, our optimal guess for a new prediction is:

$$p(y_\star;x_\star,x) = \sum_w p(y_\star|w;x_\star)p(y|w;x)p(w)/Z_x$$

For computational tractability, we typically approximate the posterior over hypotheses by a point mass at the posterior's mode $\operatorname{argmax}_{w'} p(w'|y;x)$:

$$p(w|y,x) \approx \delta(w - w^\star(y,x)) \qquad w^\star(y,x) = \operatorname{argmax}_{w'} p(w'|y;x)$$

Then

$$p(y_\star;x_\star,x) \approx p(y_\star|w^\star(y,x);x_\star)$$

What do we lose in this approximation?

$\mathbb{E}$ and max do not commute

PICTURE: "bowtie" vs "pipe"

BIC for relevant variables

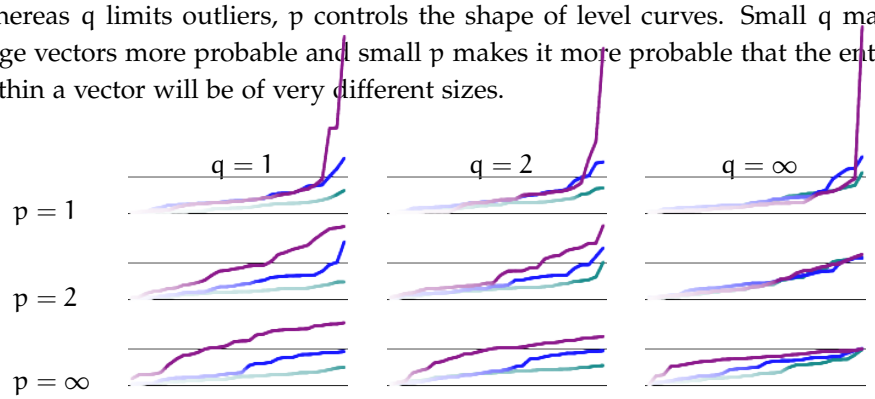
LOG PRIORS AND BAYES — fill in computation and bases visual illustration of how choice of L2 dot product matters ℓ^p regularization; sparsity eye regularization example! TODO: rank as a prior (for multioutput models)

For $1 \le p \le \infty$ and $1 \le q \le \infty$ we can consider this prior:[◦]

$$p(w) \propto \exp\left(-\left(\sum_i |\lambda w_i|^p\right)^{q/p}\right)$$

For $q = p$ this decomposes as a sum and thus has each coordinate independent. Whereas q limits outliers, p controls the shape of level curves. Small q makes large vectors more probable and small p makes it more probable that the entries within a vector will be of very different sizes.

← To define the cases $p = \infty$ or $q = \infty$ we take limits. To define the case $p = \infty = q$ we take limits while maintaining $p = q$.



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

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

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

For example, with $d = 16$ features and tolerance $\eta = 1/1000$, 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!

Thus, if $R = 1$ and $1 \leq Nm$ then with chance at least $1-\eta$:

$$\text{testing error} \leq \frac{(\text{number of margin violators})}{N} + \sqrt{\frac{(2/m^2)\log(6Nm^2)+\log(4/\eta)}{N}}$$

dimension/margin

OPTIMIZATION-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 $1-\eta$:

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

BAYES AND TESTING —

5. ideas in optimization

premature optimization is the root of all evil
— donald knuth

learning rate as metric; robustness to 2 noise structures nesterov momentum decaying step size; termination conditions batch normalization

LOCAL MINIMA —

IMPLICIT REGULARIZATION —

LEARNING RATE SCHEDULE —

LEARNING RATES AS DOT PRODUCTS —

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. bend those lines to capture rich patterns (units 2,3)

featurization

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 —

DOUBLE DESCENT —

learned featurizations

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" —

locality and symmetry in architecture

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

dependencies in architecture

5. stochastic gradient descent

6. loss landscape shape

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

D. thicken those lines to quantify uncertainty (unit 4)

bayesian models

examples of bayesian models

inference algorithms for bayesian models

combining with deep learning

E. beyond learning-from-examples (unit 5)

reinforcement ; bandits

state (dependence on prev xs and on actions) ; RL ; partial observations

deep q learning

learning-from-instructions ; farewell