microcosm

By the end of this section, you'll be able to
- define a class of linear hypotheses for given featurized data
- compute and efficiently optimize the perceptron and svm losses suffered by a hypothesis

WHAT HYPOTHESES WILL WE CONSIDER? A RECIPE FOR $\mathcal{H}$ — Remember: we want our machine to find an input-to-output rule. We call such rules **hypotheses**. As engineers, we carve out a menu $\mathcal{H}$ of rules for the machine to choose from. Let's generalize our previous digit classifying example to consider hypotheses of this format: *extract features* of the input to make a list of numbers; then *linearly combine* those numbers to make another list of numbers; finally, *read out* a prediction from the latter list. Our digit classifying hypotheses, for instance, look like:°

```
def predict(x):
  features = [brightness(x), width(x)]              # featurize
  threeness = [ -1.*features[0] +4.*features[1] ]   # linearly combine
  prediction = 3 if threeness[0]>0. else 1          # read out prediction
  return prediction
```

← Our list `threeness` has length one: it's just a fancy way of talking about a single number. We'll later use longer lists to model richer outputs: to classify between > 2 labels, to generate a whole image instead of a class label, etc.

The various hypotheses differ only in those coefficients (jargon: **weights**, here $-1, +4$) for their linear combinations; it is these degrees of freedom that the machine *learns* from data. We can diagram such hypotheses by drawing arrows:

$$\mathcal{X} \xrightarrow[\text{not learned}]{\text{featurize}} \mathbb{R}^2 \xrightarrow[\textbf{learned!}]{\text{linearly combine}} \mathbb{R}^1 \xrightarrow[\text{not learned}]{\text{read out}} \mathcal{Y}$$

Our Unit 1 motto is to *learn linearities flanked by hand-coded nonlinearities*. We design the nonlinearities to capture domain knowledge about our data and goals.

For now, we'll assume we've already decided on our featurization and we'll use the same readout as in the code above:

```
3 if threeness[0]>0. else 1
```

Of course, if we are classifying dogs vs cows that line would read

```
cow if bovinity[0]>0. else dog
```

In the next three passages we address the key question: *how do we compute the weights by which we'll compute `threeness` or `bovinity` from our features?*

HOW GOOD IS A HYPOTHESIS? FIT — We instruct our machine to find within our menu $\mathcal{H}$ a hypothesis that's as "good" as possible. That is, the hypothesis should both fit our training data and seem intrinsically plausible. We want to quantify these notions of goodness-of-fit and intrinsic-plausibility. As with $\mathcal{H}$, how we quantify these notions is an engineering art informed by domain knowledge. Still, there are patterns and principles — we will study two specific quantitative notions, the **perceptron loss** and **SVM loss**, to study these principles. Later, once we understand these notions as quantifying uncertainty (i.e., as probabilistic notions), we'll appreciate their logic. But for now we'll bravely venture forth, ad hoc!

We start with goodness-of-fit. Hypotheses correspond° to weights. For example, the weight vector $(-1, +4)$ determines the hypothesis listed above.

← A very careful reader might ask: *can't multiple choices of weights determine the same hypothesis?* E.g. $(-1, +4)$ and classify every input the same way, since they either both make `threeness` positive or both make `threeness` negative. This is a very good point, dear reader, but at this stage in the course, much too pedantic! Ask again later.

One way to quantify h's goodness-of-fit to a training example (x, y) is to see whether or not h correctly predicts y from x. That is, we could quantify goodness-of-fit by *training accuracy*, like we did in the previous digits example:

```
def is_correct(x,y,a,b):
  threeness = a*brightness(x) + b*width(x)
  prediction = 3 if threeness>0. else 1
  return 1. if prediction==y else 0.
```

By historical convention we actually like° to minimize badness (jargon: **loss**) rather than maximize goodness. So we'll rewrite the above in terms of mistakes:

← ML is sometimes a glass half-empty kind of subject!

```
def leeway_before_mistake(x,y,a,b):
  threeness = a*brightness(x) + b*width(x)
  return +threeness if y==3 else -threeness
def is_mistake(x,y,a,b):
  return 0. leeway_before_mistake(x,y,a,b)>0. else 1.
```

We *could* define goodness-of-fit as training accuracy. But we'll enjoy better generalization and easier optimization by allowing "partial credit" for borderline predictions. E.g. we could use `leeway_before_mistake` as goodness-of-fit:°

← to define *loss*, we flip signs

Food For Thought: For incentives to point the right way, loss should *decrease as* `threeness` *increases when* `y==3` but should *increase as* `threeness` *increases when* `y==1`. Verify these relations for the several loss functions we define.

```
def linear_loss(x,y,a,b):
  return 1 - leeway_before_mistake(x,y,a,b)
```

But, continuing the theme of pessimism, we usually feel that a "very safely classified" point (very positive `leeway`) shouldn't make up for a bunch of "slightly misclassified" points (slightly negative `leeway`).° But total linear loss doesn't capture this asymmetry; to address this, let's impose a floor on `linear_loss` so that it can't get too negative (i.e., so that positive `leeway` doesn't count arbitrarily much). We get **perceptron loss** if we set a floor of 1; **SVM loss** (also known as **hinge loss**) if we set a floor of 0:

← That is, we'd rather have `leeways` $+.1, +.1, +.1, +.1$ than $+10, -1, -1, -1$ on four training examples. A very positive `leeway` feels mildly pleasant to us while a very negative one feels urgently alarming.

Food For Thought: compute and compare the training accuracies in these two situations. As an open-ended followup, suggest reasons why considering training `leeways` instead of just accuracies might help improve testing accuracy.

```
def perceptron_loss(x,y,a,b):
  return max(1, 1 - leeway_before_mistake(x,y,a,b))
def svm_loss(x,y,a,b):
  return max(0, 1 - leeway_before_mistake(x,y,a,b))
```

Proportional weights have the same accuracies but different hinge losses.

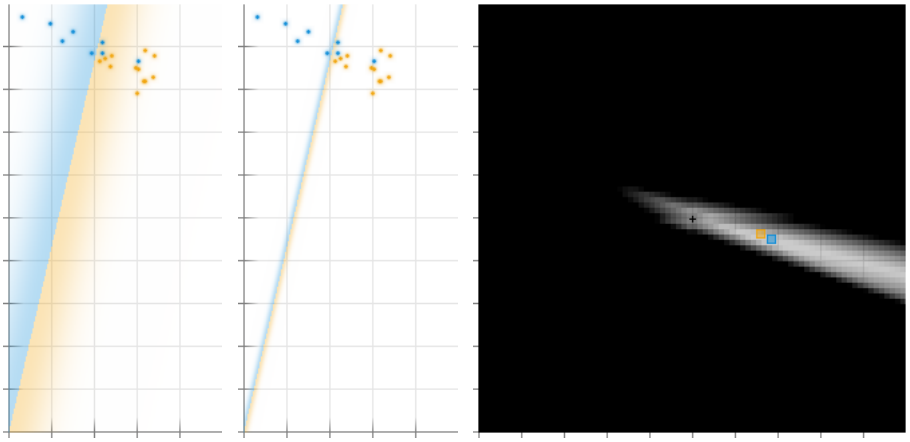

Figure 7: **Hinge loss's optimization landscape reflects confidence, unlike training accuracy's. — Left rectangular panes**. An under-confident and over-confident hypothesis. These have weights $(a/3, b/3)$ and $(3a, 3b)$, where $(a, b) = (-3.75, 16.25)$ minimizes training hinge loss. The glowing colors' width indicates how rapidly `leeway` changes as we move farther from the boundary. — **Right shaded box**. The (a, b) plane, centered at the origin and shaded by hinge loss, with training optimum blue. Indecisive hs (e.g. if `threeness`≈ 0) suffer, since $\max(0, 1-\ell) \approx 1$, (not 0) when $\ell \approx 0$. Hinge loss penalizes *overconfident* mistakes severely (e.g. when $y = 1$ yet `threeness` is huge): $\max(0, 1-\ell)$ is unbounded in ℓ. If we start at the origin $(a, b) = (0, 0)$ and shoot (to less underconfidence) toward the optimal hypothesis, loss will decrease; but once we pass that optimum (to overconfidence), loss will (slowly) start increasing again.

HOW GOOD IS A HYPOTHESIS? PLAUSIBILITY — Now to define intrinsic plausiblity, also known as a **regularizer** term. There are many intuitions° but we'll focus for now on capturing this particular intution: *a hypothesis that depends very much on very many features is less plausible*. That is, we find a hypothesis more plausible when its "total amount of dependence" on the features is small. We may conveniently quantify this as proportional to a sum of squared weights (jargon: L2):° implausibility of $h = (a, b) = \lambda(a^2 + b^2 + \cdots)$. In code:

```
LAMBDA = 1.
def implausibility(a,b):
  return LAMBDA * np.sum(np.square([a,b]))
```

Intuitively, the constant λ=LAMBDA tells us how much we care about plausibility relative to goodness-of-fit-to-data.

Here's what the formula means. Imagine each of three friends has a theory° about which birds sing. Which hypothesis do we prefer? Well, AJ seems way too confident. Maybe they're right that wingspan matters, but it seems implausible that wingspan is so decisive. Pat, meanwhile, doesn't make black-and-white claims, but Pat's predictions depend substantively on many features: flipping any one quality flips their prediction. This, too, seems implausible. By contrast, Sandy's hypothesis doesn't depend too strongly on too many features. To me, a bird non-expert, Sandy's seems most plausible.

Now we can define the overall undesirability of a hypothesis:°

```
def objective_function(examples,a,b):
  data_term = np.sum([svm_loss(x,y,a,b) for x,y in examples])
  regularizer = implausibility(a, b)
  return data_term + regularizer
```

To build intuition about which hypotheses are most desirable according to that metric, let's suppose λ is a tiny positive number. Then minimizing the objective function is the same as minimizing the data term, the total SVM loss: our notion of implausibility only becomes important as a tiebreaker.

Now, how does it break ties? Momentarily ignore the Figure's rightmost orange point and consider the black hypothesis; its predictions depend only on an input's first (vertical) coordinate, so it comes from weights of the form $(a, b) = (a, 0)$. The $(a, 0)$ pairs differ in SVM loss. If $a \approx 0$, each point has leeway close to 0 and thus SVM loss close to 1; conversely, if a is huge, each point has leeway very positive and thus SVM loss equal to the imposed floor: 0. So SVM loss is 0 as long as a is so big that each leeway to exceed 1.

Imagine sliding a point through the plane. Its leeway is 0 at the black line and changes by a for every unit we slide vertically. So the farther the point is from the black line, the less a must be before leeway exceeds 1 — and the happier is the regularizer, which wants a small. So *minimizing SVM loss with an* L2 *regularizer favors decision boundaries far from even the closest correctly classified points!* The black line's margins exceed the gray's, so we favor black.

For large λ, then this margin-maximization tendency can be so strong that it overrides the data term. Thus, even when we bring back the rightmost orange point we ignored, we might prefer the black hypothesis to the gray one.

← these come from domain knowledge. symmetry especially comes to mind.

← We could just as well use $6.86a^2 + b^2$ instead of $a^2 + b^2$.

Food For Thought: When (a, b) represent weights for brightness-width digits features, how how do the hypotheses with small $6.86a^2 + b^2$ visually differ from those with small $a^2 + b^2$?

← **AJ** says with wingspan. A bird with a wings shorter than 1ft can't fly far, so it's *sure* to sing instead. Conversely, birds with longer wings never sing. **Pat** says to check whether the bird grows red feathers, eats shrimp, lives near ice, wakes in the night, and has a bill. If, of these 5 qualities, an even number are true, then the bird probably sings. Otherwise, it probably doesn't. **Sandy** uses both wingspan and nocturnality: shorter wings and nocturnality both make a bird somewhat more likely to sing.

← We'll use SVM loss but feel free to plug in other losses to get different learning behaviors!

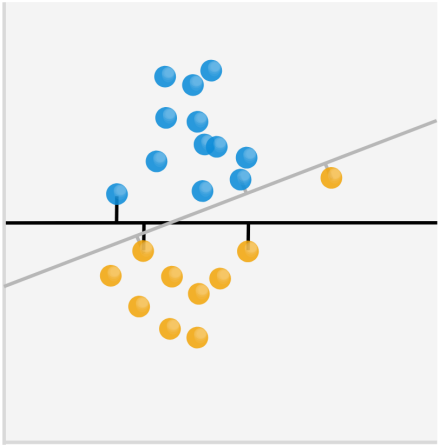

Figure 8: **Balancing goodness-of-fit against intrinsic plausibility leads to hypotheses with large margins.** A hypothesis's **margin** is its distance to the closest correctly classified training point. Short stems depict these distances for two hypotheses (black and gray). If not for the rightmost orange point, we'd prefer black over gray since it has larger margins. With large λ (i.e., strong regularization), we might prefer black over gray even with that rightmost orange point included, since expanding the margin is worth the single misclassification.

WHICH HYPOTHESIS IS BEST? OPTIMIZATION BY GRADIENT DESCENT — Now that we've defined our objective function, we want to find a hypothesis $h = (a, b)$ that minimizes it. We *could* try brute force, as follows:

```
def best_hypothesis():
  # returns a pair (objective-function value, hypothesis)
  return min((objective_function(training_data, a, b), (a,b))
             for a in np.arange(-50,+50,.25)
             for b in np.arange(-50,+50,.25)                )
```

But this is slow! Here we're searching a 2D grid at resolution ≈ 400, so we call the objective function 400^2 times. That exponent counts the number of parameters we're finding — here, we seek 2 weights a and b, but if we had 10 features and 10 weights, we'd make 400^{10} calls. Yikes!

Let's instead use more of the information available to direct our search. Suppose at some point in our search the best h we've found so far is (a, b). The objective function is an implausibility plus, for each of N training points (x_i, y_i), a badness-of-fit:[◦]

$$+(\lambda/N)(a^2 + b^2) + \max(0, 1 - y_0(a \cdot \mathrm{br}(x_0) + b \cdot \mathrm{wi}(x_0)) + \cdots$$
$$+(\lambda/N)(a^2 + b^2) + \max(0, 1 - y_{42}(a \cdot \mathrm{br}(x_{42}) + b \cdot \mathrm{wi}(x_{42})) + \cdots$$

Let's try to decrease this sum by reducing one row at a time. Well, we can reduce a row's λ/N term by moving a, b closer to 0. If the row's $\max(0, 1 - \ell)$ term is 0, then[◦] any small change in (a, b) won't change that term. But $\max(0, 1 - \ell)$ term isn't 0, then we can decrease it by increasing ℓ, i.e., by increasing (say):

$$\underbrace{+1}_{y_{42}}(a \cdot \underbrace{0.9}_{\mathrm{br}(x_{42})} + b \cdot \underbrace{0.1}_{\mathrm{wi}(x_{42})})$$

We can increase ℓ by increasing a or b; but increasing a gives us more bang for our buck ($0.9 > 0.1$), so we'd probably nudge a more than b, say, by adding a multiple of $(+0.9, +0.1)$ to (a, b). Conversely, if $y_i = -1$ then we'd add a multiple of $(-0.9, -0.1)$ to (a, b). Therefore, to reduce the ith row, we want to move a, b like this: Add a multiple of $(-a, -b)$ to (a, b); and, unless the max term is 0, add a multiple of $y_i(\mathrm{br}(x_i), \mathrm{wi}(x_i))$ to (a, b).

Now, what if improving the ith row messes up other rows? Because of this danger we want to take small steps — i.e., we scale the multiples we mention above by some small η. That way, even if the rows all pull (a, b) in different directions, the dance will buzz close to some average (a, b) that maximizes the average row's happiness. So let's initialize (a, b) arbitrarily and take a bunch of small steps![◦]

```
ETA = 0.01
ab = initialize()
for t in range(10000):
  xfeatures, y = fetch_datapoint_from(training_examples)
  ab = ab + ETA * ( - L * ab
                    + y * xfeats * (0 if max(0., y*ab.dot(xfeatures))==0 else 1) )
```

This is the **pegasos algorithm** we'll see in the project. Soon we'll formalize and generalize this algorithm using calculus.

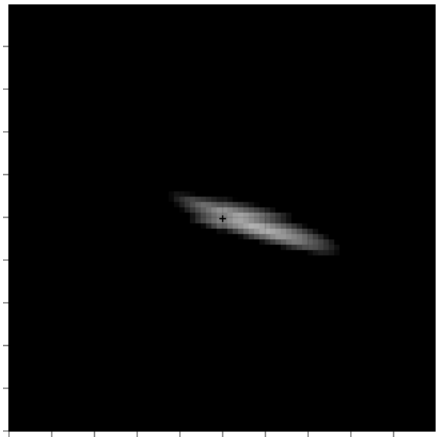

Figure 9: With $\lambda = 0.02$ the objective visibly prefers weights near 0. We develop an algorithm to take steps in this plane toward the minimum, 'rolling down' the hill so to speak.

← Here, $\mathrm{br}(x)$ and $\mathrm{wi}(x)$ stand for the features of x, say the brightness and width. Also, we'll use take y's values to be ± 1 (rather than cow vs dog or 1 vs 3), for notational convenience.

← except in the negligible case where $\ell = 0$ exactly

← Food For Thought: We could have used perceptron loss instead of hinge loss. Mimicking the reasoning above, derive the familiar perceptron update. For simplicity let $\lambda = 0, \eta = 1$.