goodness of fit

By the end of this section, you'll be able to
- define (in both math and code) a class of linear hypotheses for given featurized data
- compute the perceptron and hinge losses a given hypothesis suffers on given data

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. We'll 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 concept of threeness (and what plays the same role in other ML projects) is called the **decision function**. The decision function takes an input x and returns some kind of score for what the final output should be. The decision function value is what we feed into the readout.

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. These arrows summarize the situation:°

← This triple of arrows is standard in ML. But the name 'readout' is not standard jargon. I don't know standard jargon for this arrow.

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

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

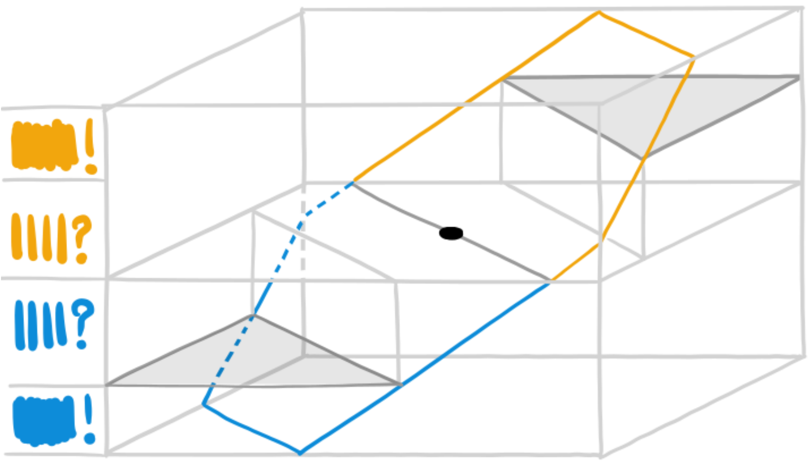

Figure 7: **The *slope* of the decision function's graph is meaningful, even though it doesn't affect the decision boundary.** We plot the decision function (vertical axis) over weight-space (horizontal axes) (**black dot**). The graph is an infinite plane, a hexagonal portion of which we show; it attains zero altitude at the decision boundary. We interpret high (low) altitude as 'confident the label is orange (blue)'; intermediate altitudes, as leaning but without confidence. We'll introduce a notion of 'goodness-of-fit-to-data' that seeks not just to correctly classify all the training points BUT ALSO to do so with confidence. That is, we want orange (blue training points to all be above (below) the gray 'shelf' that delimits high-confidence. I like to imagine a shoreline: an interface between water and beach. We want to adjust the topography so that all blue (orange) training points are deep underwater (high and dry).

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

```
prediction = 3 if threeness[0]>0. else 1          # for a digit-classifying project
prediction = 'cow' if bovinity[0]>0. else 'dog'   # for an animal-classifying project
```

← Intuitively, threeness is the machine's **confidence** that the answer is 3. Our current readout discards all information except for the sign: we forget whether the machine was very confident or slightly confident that the answer is 3, so long as it prefers 3 over 1. However, we will care about confidence levels when assessing goodness-of-fit.

That's how we make predictions from those "**decision function values**" threeness or bovinity. But to compute those values we need to determine some weights. *How shall we determine our weights and hence our hypothesis?*

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 $(-10, +40)$ 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 can be a glass half-empty kind of subject!

← We'll refer to the "leeway before a mistake" throughout this section; it's a standard concept but **not** a concept with a standard name afaik.

```
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, hence the '1−'

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)
```

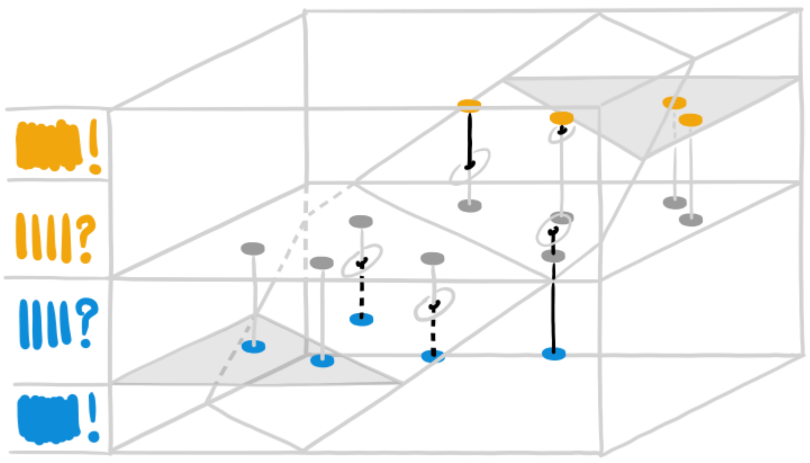

Figure 8: **Geometry of leeways, margins, and hinge loss**. As before, we graph the decision function (vertical axis) against weight-space. The $y = -1$ ($y = +1$) training points (x, y) sit one unit below (above) the zero-altitude plane. We draw regions where the graph surpasses ± 1 altitude as shaded 'shelves'. **Leeway** is the vertical distance between a gray x and the graph (and it's positive when the point is correctly classified). **Linear loss** is the vertical distance between an (x, y) point and the graph (positive when the point is on the 'wrong' side of the graph, i.e., when the graph lies between the point and the zero-altitude plane). **Hinge loss** is linear loss, except no reward is given for any distance beyond the 'shelves' at ± 1 altitude. We draw hinge loss in **black vertical segments**. A hypothesis's **margin** is the horizontal distance the edge of those 'shelves' and the decision boundary.

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:

```
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))
```

← That is, we'd rather have `leeway`s $+.1, +.1, +.1, +.1$ 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 optimizing `leeway`s instead of just accuracies might help improve testing accuracy.

Proportional weights have the same accuracies but different hinge losses.

Food For Thought: Can we say the same of perceptron loss?

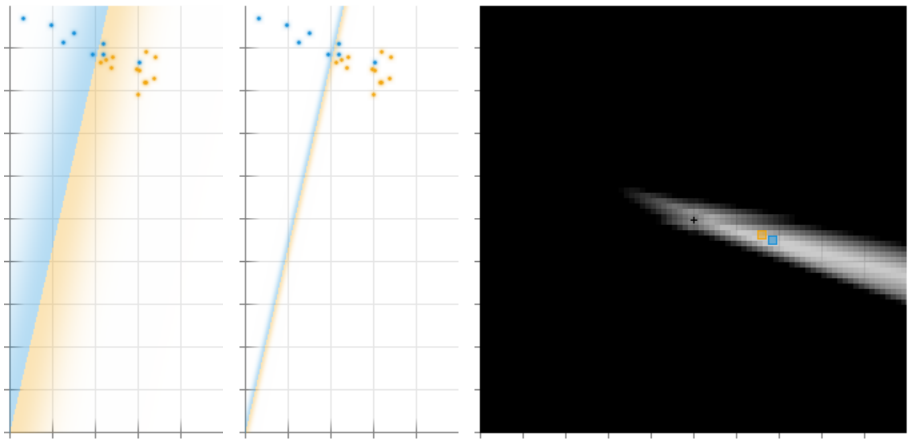

Figure 9: **Hinge loss's optimization landscape reflects confidence, unlike training accuracy's.** — **Left rectangular panes**. An under-confident and over-confident hypothesis. These have weights $(a/3, b/3)$ and $(3a, 3b)$, where $(a, b) = (-3.75, 16.25)$ minimizes training hinge loss. The glowing 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 *over-confident* 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.