data dependent features

By the end of this section, you'll be able to
- featurize based on *similarity* and *low-rank approximations* to present data in a form that makes learning complex patterns easy
- use the kernel trick to quickly optimize very-high-dimensional models

LANDMARKS AND SIMILARITY — The 'black-hole' nonlinearity (Figure 20) nicely separates its training data. When domain knowledge says some 'landmark' in $\mathcal{X}$ is salient, we can make a 'black-hole-shaped' feature centered on that landmark. Intuitively, such a feature measures how *similar* a given x is to the landmark. (In general we define similarity using domain knowledge.)

If something works well, it's worth trying it more (:p), so let's now make multiple black-hole-shaped features, one centered around each of *several* landmarks! If we use three representative 'landmarks' $x_\circ, x_\square, x_\star$, say, we get a featurization

$$\varphi(x) = (\text{similarity}(x, x_\circ), \text{similarity}(x, x_\square), \text{similarity}(x, x_\star)) \in \mathbb{R}^3$$

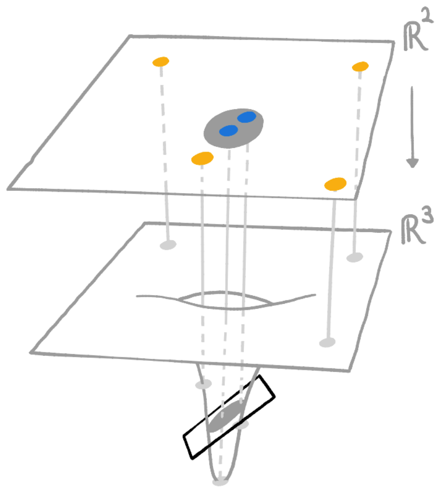
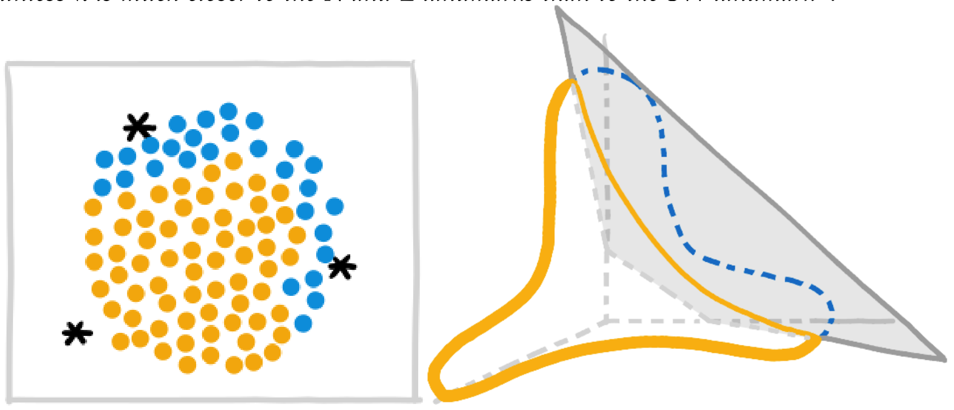


Figure 20: **A nonlinear 'black-hole' feature** (encountered previously). We map raw 2-D inputs x to 3-D feature-vectors $\varphi(x)$s, defined as

$$\varphi(x) = (x[0], x[1], f(x))$$

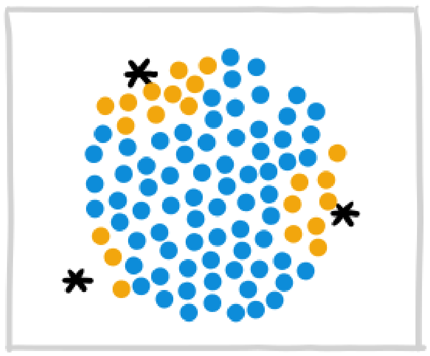
for some complicated function f. For example, maybe $f(x) = \exp(-\|x - (3,4)\|^2)$, a smooth bump centered around the landmark $(3,4)$, which domain knowledge says is salient.

Here's what the decision boundary might look like if $\mathcal{X} = \mathbb{R}^2$ and we define $\text{similarity}(x, x')$ to be big when x, x' are closer than a centimeter in Euclidean distance, but then to rapidly decay as x, x' move apart. Our three **landmarks (black asterisks)** lie to the south-west, north, east. Our nonlinear φ gives us a curvy boundary in raw $\mathcal{X}$ space (left) from each **linear hyperplane** through featurespace (right). Intuitively, the hyperplane's weights say, "*predict orange unless x is much closer to the N and E landmarks than to the SW landmark*":



This looks nice. Very expressive.° *But how do we choose our landmarks?* That is: *what if we lack domain knowledge specific enough to tell us which reference-points in $\mathcal{X}$ are important?* Well, we might guess that some (we don't know which) of our N training points may be useful landmarks. Since we don't know which, let's center 'black-holes' at *every* training point at once! Then we get N new features (and we'll rely on gradient descent to select which of these are actually useful):

← Food For Thought: With the same three landmarks, find a hyperplane in feature-space that produces these predictions:



$$\varphi(x) = (\text{similarity}(x, x_0), \text{similarity}(x, x_1), \cdots \text{similarity}(x, x_{N-1})) \in \mathbb{R}^N$$

This is a *data-dependent* featurization: we adapt φ to training data.

At day's end, we'll use φ to make predictions the same way as before, namely via a hypothesis $(x \mapsto \text{sign}(w \cdot \varphi(x)))$. That is, we'll have a bunch of weights w_i and we'll classify a fresh x according to the sign of

$$\textstyle\sum_i w_i \cdot (\text{ith feature of } x) = \sum_i w_i \cdot \text{similarity}(x, x_i)$$

With N features for N training points, $\mathcal{H}$ will (usually) be very expressive. We (potentially) pay for this improved approximation by suffering worse generalization: expressivity means $\mathcal{H}$ contains many hypotheses that well-fit the training data but do horribly at testing time. On the other hand, if we've done a good job choosing *especially informative* features, then $\mathcal{H}$ will contain a hypothesis that does well on both training and testing data. Thus, regularization is crucial!

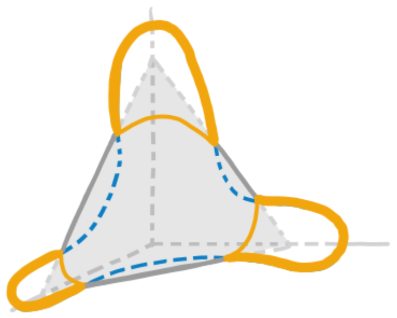


Figure 21: A hyperplane in feature-space that produces the predictions depicted in the margin figure on the previous page.

Below we show how regularization affects learning when we use similarity features. Here we'll define similarity$(x, x') = \exp(-\|x - x'\|^2/\sigma^2)$; this popular choice is called the **Gaussian RBF kernel**. We minimize our familiar friend $\lambda\|w\|^2 + \sum_i \max(0, 1 - y_i w \cdot \varphi(x_i))$ for different settings of λ, σ.

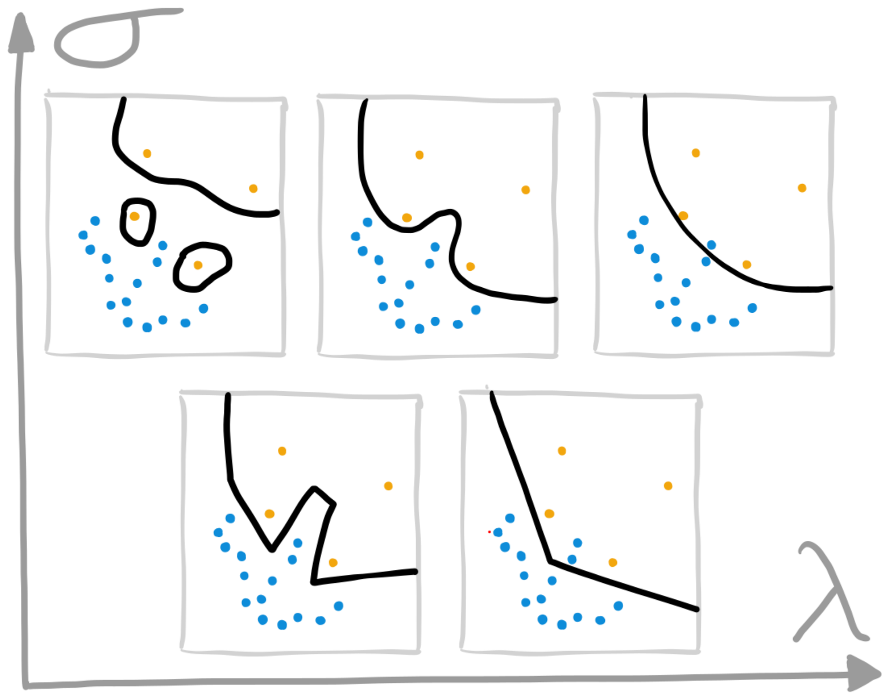


Figure 22: **The RBF kernel's σ controls spatial resolution.** As expected, λ controls model complexity. But so does σ! *Why?* Well, σ is a kind of 'blurring' parameter: the ith feature $\varphi_i(x) = \exp(-\|x - x_i\|^2/\sigma^2)$ is nearly constant once x gets closer than σ to the landmark x_i. *Intuitively, φ 'throws away' spatial information at resolutions < σ.* So for huge σ, the decision function will lack detailed wrinkles (unless w is huge). Conversely, when $\|x - x_i\|$ far exceeds σ, $\varphi_i(x)$ decays very rapidly. So for tiny σ, $\varphi(x)$'s entries will all be small — but in relative terms, the entry for the x_k closest to x will far exceed all other entries. So for tiny σ the hypothesis class easily expresses **nearest neighbors** rules of the form "predict whichever label the closest training point has."

Food For Thought: Is training data featurized with the $\sigma = 1$ Gaussian RBF kernel always linearly separable? That is, can we always achieve perfect training accuracy? Assume, of course, that no training points overlap. *Hint*: use huge ws.

Food For Thought: Design a similarity function to help recognize spam emails. For our purposes, an email is not just body text but also a timestamp, sender address, subject line, set of attachments, etc.

SUPERPOSITIONS AND KERNELS — In this passage we'll discuss° how, once we've featurized our xs by similarities, we'll select a hypothesis from $\mathcal{H}$ based on training data. As usual we can do ordinary gradient descent, the kind we're now used to. But we'll here explore a different method, a special gradient descent. The method is important because it offers a fast way° to solve seemingly different problems:

← This is probably the most challenging passage I'll write. I worked hard to explain kernels as simply as possible but no simpler.

← Frankly, I don't know of modern projects where this 'kernel trick' speedup matters. With GPUs and big data and deep learning, the main computing bottlenecks lie elsewhere. Still, the kernel trick rewards study: it's an idea that continues to inspire modern techniques.

ordinary (slow) gradient descent on data featurized as we please, say by $x \mapsto \varphi(x)$	*is equivalent to*	special (fast) gradient descent on data featurized according to $\text{similarity}(x, x') = \varphi(x) \cdot \varphi(x')$

Here's an analogy for the speedup. We're quick-sorting an array of objects-so-large-that copies are expensive. Instead of directly sorting, we create and sort an array of pointers to the original objects; only as a final step do we arrange the objects based on the sorted pointers. That way we do N large-object-swaps instead of N log N.° Better yet, if the purpose of sorting was to allow us to quickly binary search to count the array elements x_k less than given xs, then we can avoid large-object-swaps *completely* (!) by binary searching the array of pointers.

← Instead of using pointers to implicitly arrange an array of memory-hogging objects into an ordering that helps compute a rank for a fresh x, we'll use numbers to implicitly arrange a training set of high-dimensional featurevectors into a formal linear combination that helps compute a label for a fresh x.

Now for the two methods, ordinary and special. Both methods aim to reduce the loss $\ell(y_k, d_k)$ suffered at the kth training example (here we use the shorthand $d_i = w \cdot (\text{features of } x_i)$ for the decision function value at the ith training input). Whereas the ordinary method subtracts gradients of loss with respect to w, the special method subtracts gradients of loss with respect to d. We can write both in terms of the partial derivative $g_k = \partial\ell(y_k, d_k)/\partial d_k$:°

← We use a fact that looks advanced but that's obvious once you get past the language:

$$\partial\,\ell(y_k, d_k)/\partial\, d = g_k \cdot (\text{kth one-hot vector}) \\ = (0, \cdots, 0, g_k, 0, \cdots, 0)$$

This says that, since $\ell(y_k, d_k)$ depends on d only through the kth coordinate of d, the gradient will be zero in every direction except the kth one. It's a case of the chain rule. So *the special update only changes the kth weight entry!*

ordinary, w-based update

$$w^{\text{new}} = w - \eta\,\frac{\partial\,\ell(y_k, d_k)}{\partial\,w} = w - \eta\, g_k \cdot (\text{features of } x_k)$$

special, d-based update

$$w^{\text{new}} = w - \eta\,\frac{\partial\,\ell(y_k, d_k)}{\partial\,d} = w - \eta\, g_k \cdot (\text{kth one-hot vector})$$

Note that w has as many entries as there are features and d has as many entries as there are training examples; so the special update only makes sense because we've chosen feature-set indexed by training points! Intuitively $d = X \cdot w$,° so w and d are proportional and the ordinary and special updates are just stretched versions of each other. In multiple dimensions, different directions get stretched different amounts; it's because of this that using the two updates to classify similarity-featurized data gives different predictions at day's end.

← Here, X is the $N \times N$ matrix whose kth row is the featurization of the kth training input. So $X_{ki} = \text{similarity}(x_k, x_i)$.

But using the special update to classify similarity-featurized data gives the *same*° predictions as using the ordinary update to classify data featurized as we please, say by $\varphi : \mathcal{X} \to \mathbb{R}^F$, so long as we define $\text{similarity}(x, x') = \varphi(x) \cdot \varphi(x')$ to be a dot product. In each ordinary update we do an F-dimensional dot product of weights with $\varphi(x_k)$. By contrast, in each special update we do an N-dimensional dot product of weights with x_k's similarity-features. We get a speedup° when there are many more features F than training points N.

← Why? We'll see on the next page. For now we briefly discuss the upshot of this equivalence.

← Food For Thought: Say we have in RAM all $N \times F$ entries for the training φ-feature vectors. T ordinary updates would take FT multiplications; T special updates would take $FN^2 + NT$ multiplications. *Where does that* FN^2 *come from?*

Food For Thought: Say $\mathcal{X} = \mathbb{R}$. We define a thousand features $\varphi_s(x) = \sqrt{x^s/s!}$ for $0 \le s < F = 1000$. Compute $\text{similarity}(x, x)$ quickly, up to floating point error. Compute $\text{similarity}(x', x'')$ by comparing $x = x', x ='', x = x' + x''$.

Food For Thought: Say $\mathcal{X} = \mathbb{R}^3$. Let's define $\text{similarity}(x, x') = \exp(-\|x - x'\|^2)$. Explain to a friend why 'similarity' is a fair name for that function.° Find a $\varphi : \mathcal{X} \to \mathbb{R}^{1000}$ so that $\text{similarity}(x, x') = \varphi(x) \cdot \varphi(x')$ up to floating point error.

← This similarity is called the **Gaussian RBF kernel**. We can also scale $\|x - x'\|$ (i.e., measure distance in different units) before exponentiating.

Each ordinary update adds some linear combination of training inputs to the weights, so (if we initialize weights to zero) we can after any number of steps write $w = \sum_i \alpha_i \varphi(x_i)$.° Then the decision function value for any x is

← those αs change as we change the weights

$$d = w \cdot \varphi(x) = \sum_i \alpha_i \varphi(x_i) \cdot \varphi(x) = \sum_i \alpha_i \text{similarity}(x_i, x)$$

We recognize this a sum of similarity-based features, weighted by αs! In other words, the F-dimensional weight vector $w = \sum_i \alpha_i \varphi(x_i)$ *makes the same predictions* on φ-featurized data as the N-dimensional weight vector $(\alpha_0, \cdots, \alpha_{N-1})$ makes on similarity-featurized data!

So our two scenarios have (effectively)° the same hypothesis class. To see why they moreover select from this class *the same* hypothesis, we compare the ordinary update for weight w against the special update for weight α:

← I say 'effectively' because the ordinary-with-φ scenario could have many more hypotheses corresponding to weight vectors not of the form $\sum_i \alpha_i \varphi(x_i)$. But gradient descent in that scenario wouldn't select those hypotheses anyway.

ordinary update on w	special update on α
$w^{\text{new}} = w - \eta\, g_k \cdot \varphi(x_k)$	$\alpha_k^{\text{new}} = \alpha_k - \eta\, g_k$

and then substitute $w = \sum_i \alpha_i \varphi(x_i)$. This whole idea is the **KERNEL TRICK**.

Okay, that's a lot to absorb. Let's see how it works for perceptron loss. Here $g_k = \partial\ell(y_k, d_k)/\partial d_k$ is $-y_k \cdot \text{step}(-d_k)$: zero if $d_k > 0$ and $-y_k$ otherwise. In

ordinary perceptron	kernelized perceptron
$w^{\text{new}} = w + \eta y_k \text{step}(-d_k) \cdot \varphi(x_k)$	$\alpha_k^{\text{new}} = \alpha_k + \eta y_k \text{step}(-d_k)$
$d_k = w \cdot \varphi(x_k)$	$d_k = \sum_i \alpha_i \text{similarity}(x_i, x_k)$

code, the kernelized perceptron looks like this:°

← We call `similarity_features(x[42])` again each time we update based on the $(k = 42)$nd example. This wastes time. It's faster instead to precompute the training examples' similarity features once and for all.

```
similarity_features = lambda x: np.array([similarity(x,x[k]) for k in range(N)])
predict = lambda alpha, x: np.sign(np.dot(alpha, similarity_features(x)))
alpha = np.zeros(N)
for k in IDXS: # sequence of training examples indices to update based on
  alpha[k] = alpha[k] + ETA * y[k] * (1 if predict(alpha, x[k]) != y[k] else 0)
```

The program never refers explicitly to φ; it only calls `similarity`.°

← We call such a program **kernelized**.

Food For Thought: Write a kernelized program to minimize hinge° loss.

← i.e., $\ell(y, d) = \max(0, 1 - yd)$, remember?

Food For Thought: That $w = \sum_i \alpha_i \varphi(x_i)$ suggests two different L2 regularizers: $\lambda\|w\|^2$ or $\lambda\|\alpha\|^2$. Write kernelized programs to minimize (hinge loss plus either regularizer). The $\lambda\|w\|^2$ is trickier.

We have flexibility in designing our function similarity : $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$. But for the function to be worthy of the name, it should at least satisfy these two rules: (**symmetry**) x is as similar to x' as x' is to x and (**positivity**) x is similar to itself.°

← In symbols, symmetry says $\text{similarity}(x, x') = \text{similarity}(x', x)$ and positivity says $\text{similarity}(x, x) \geq 0$.

Food For Thought: *Show that, if our similarity function disobeys the positivity rule, then a kernelized perceptron update on x_k can* worsen *the prediction for x_k!*

So the positivity rule helps learning. That rule generalizes to a yet more helpful one ("**Mercer's**"): if we expand our L2 regularizer $\lambda\|w\|^2 = \lambda\|\sum_i \alpha_i \varphi(x_i)\|^2$ into a weighted sum of similarities, that sum should never be negative.°

← You might say: the answer is $\lambda\|w\|^2$, so it can't be negative! That's true when we derive similarity from φ (for example, $\text{similarity}(x, x') = \exp(-\|x - x'\|^2)$ obeys Mercer). But my point is that we can invent similarity functions in other ways; those functions might or might not obey Mercer. Optimization works better when they obey Mercer.

Food For Thought: Say $\mathcal{X} = \mathbb{R}^2$. Does $\text{similarity}(x, x') = 1 + \|x - x'\|$ obey Mercer's rule? Our $N = 2$ training points are $\{(x_0, y_0 = +1), (x_1, y_1 = -1)\}$ with x_0 far from x_1. Manually work out one gradient step on the *total* perceptron loss. Sketch the decision boundary. Notice that it goes 'the wrong way'!

QUANTILES AND DECISION TREES — There are many other good ideas for choosing featurizations based on data. Here's one: *rescale a feature based on the distributions of its values in the training data*.

← VERY OPTIONAL PASSAGE

We won't discuss them in lecture, but **decision trees** can be very practical: at their best they offer fast learning, fast prediction, interpretable models, and robust generalization. Trees are discrete so we can't use plain gradient descent; instead, we train decision trees by greedily growing branches from a stump. We typically make predictions by averaging over ensembles — "forests" — of several decision trees each trained on the training data using different random seeds.

LINEAR DIMENSION-REDUCTION — There are many other good ideas for choosing featurizations based on data. Here's one: *if some raw features are (on the training data) highly correlated*, collapse them into a single feature. Beyond saving computation time, this can improve generalization by reducing the number of parameters to learn. We lose information in the collapse — the small deviations of those raw features from their average° — so to warrant this collapse we'd want justification from domain knowledge that those small deviations are mostly irrelevant noise.

← or more precisely, from a properly scaled average

One way of understanding such linear dimension-reduction is matrix factorization. I mean that we want to approximate our $N \times D$ matrix X of raw features as $X \approx FC$, a product of an $N \times R$ matrix F of processed features with an $R \times D$ matrix C that defines each processed feature as a combination of the raw features.

There's **principal component analysis**.

As a fun application, we can fix a corrupted row (i.e., vector of raw features for some data point) of X by replacing it with the corresponding row of FC. We expect this to help when the character of the corruption fits our notion of "$\approx$". For example, if the corruption is small in an L2 sense then PCA is appropriate.