

# 1 Administrivia

- There's a bug in HW 2, problem 5. The regularization term should be *added* to the error term, not subtracted (of course). The repaired version should be online now.

# 2 Nonlinear discrimination with Support Vector Machines

- Recall that last time, we finished up by covering the *nonseparable* case for the SVM.
- We ended up with the QP formulation:

$$\begin{aligned} \min_{\mathbf{W}} \quad & \widehat{\mathbf{W}}^\top \widehat{\mathbf{W}} + D \sum_i \xi_i \\ \text{subject to} \quad & y_i \mathbf{W}^\top X_i \geq 1 \quad \forall i \end{aligned}$$

- This is pretty cool, actually, but it only goes so far. It handles a *little* data nonseparability, but it breaks pretty badly if the data is highly nonlinear.
- E.g., if the two classes are concentric or something else nasty.
- Now for the mindbending part...
- We still want to be able to handle *nonlinear* systems. It turns out that if you're a little bit clever, you can get there too.
- I *will not* prove it to you in this class, but it's possible to rewrite the previous system with some variable transformations as:

$$\begin{aligned} \max_{\alpha_i} \quad & \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j X_i^\top X_j \\ \text{Subject to} \quad & 0 \leq \alpha_i \leq D \\ & \sum_i \alpha_i y_i = 0 \end{aligned}$$

(No, it's not obvious, and I told you I wouldn't prove it to you. Go look up the Burges tutorial on SVMs if you really want to understand where that came from...)

- The important thing about that expression is that it's written in terms of a *dot product* between the  $X_i$ . That is, if you know the value of the dot product, *you don't actually need to know what the  $X_i$  themselves are!*
- Why is this useful? Well, suppose that we take our *original*  $X_i$ , which live in  $\mathbb{R}^d$ , and project them into some higher dimensional space,  $\mathbb{R}^k$  via a *nonlinear* transform  $\Phi(X)$  ( $k \gg d$ ).
- It may be that  $k$  is so large that it's painful to manipulate (we have examples of  $k = \infty$ )

- But suppose that we have a convenient way to manipulate the product  $K(X_i, X_j) = \Phi(X_i)\Phi(X_j)$ . Now we could just plug in  $K$  in the place of the dot product, above, and carry through the same solution. It turns out to work just fine.
- Which begs the question of, if you can't handle  $\Phi$ , how can you get  $K$ ?
- Well, often  $K$  is easy to represent, even if  $\Phi$  is hard. E.g.,

$$K(X_i, X_j) = e^{-\frac{(X_i - X_j)^T(X_i - X_j)}{2\sigma^2}}$$

which you should recognize as, essentially, being a Gaussian. That's a  $K$  equivalent to an infinite-dimensional  $\Phi$  (again, I won't prove this to you). Another example is:

$$K(X_i, X_j) = (X_i^T X_j)^p$$

for some integer  $p > 1$ . It turns out that if your original  $X_i \in \mathbb{R}^d$ , then this corresponds to a  $\Phi$  that lives in a space of dimension  $\binom{d+p-1}{p}$ . If you're looking at, say,  $16 \times 16$  images ( $d = 256$ ), and  $p = 4$  (you're looking at a degree-4 polynomial expansion), then  $\Phi$  is a 183,181,376-dimensional space. Wow!

- One little fly in the ointment, of course, is that our classifier still depends on a  $W$ , which now exists in the  $\Phi$  space. I.e.,  $W$  may be infinite dimensional!
- Thankfully, it turns out that by the same bit of magic that is necessary to rewrite our QP as above, it's also possible to rewrite our classifier.
- Recall that our classifier rule was:

$$f(X) = \text{sign}(W^T X)$$

- This can be rewritten as:

$$f(X) = \text{sign} \left( \sum_i y_i \alpha_i K(X_i, X) \right)$$

where  $X$  is the test point,  $X_i$  are the training data points, and  $\alpha_i$  are the auxiliary vars that we introduced in the new QP formulation, above.

- So we can conveniently express  $W$  itself *implicitly* as the sum of the *kernel* dot products of the training data with the test data.
- Furthermore, it turns out that  $\alpha_i > 0$  *only* for the support vectors. This is, in fact, how you identify the support vectors (even for the separable case). Solving the QP will identify them for you, and this sum only has to take place over the support vectors (i.e.,  $\alpha \neq 0$ )
- So far, I've completely represented this in terms of binary classification – labels can be either +1 or -1.
- What do you do when you have multiclass data?

- There exists a full SVM formulation for the multiclass case, but a couple of simple hacks that work pretty well in practice are:
- For  $k$ -class data, build  $k$  different SVMs, where each is trained on the concept “is class  $i$ ” vs. “is not class  $i$ ”
- Or, can make just  $\log k$  classifiers by assigning each class a *bit string* description, and training each classifier on the corresponding bit string.

### 3 Decision Trees

- This material is covered in Ch. 3 of your book. I’ll do it pretty quickly b/c it’s handled fairly in-depth there.
- So now we’re going to shift gears a little bit to a different kind of learner.
- Rather than learning a single, monolithic representation (a hyperplane or a nonlinear hyperplane), this will learn a bunch of very simple local decision surfaces.
- In addition, it can handle *categorical* (i.e., discrete, unordered, a.k.a., *nominal*) data.
- The *Decision Tree* classifier is a tree of tests of the form

```

if (x7 <= -3.5) then
  if (x3 = "cat") then
    class = "furry"
  else if (x3 = "snake") then
    class = "scaly"
  else
    class = "feathery"
  end
else
  class = "leathery"
end

```

- There are a number of important things to note about this representation:
  - The input data features can be either real ( $x_7 < -3.5$ ) or categorical ( $x_3 = \text{“cat”}$ )
  - Categorical tests can yield multi-way splits, while real-valued tests yield binary splits.
  - The class is discrete, but not necessarily binary.
  - The features are tested one at a time, and not necessarily in order.
- You can conceptualize this in a couple of ways. As a tree of decision nodes and leaves (classes)

- Or as a recursive partitioning of the data space. Note that each split produces a single, axis orthogonal split relative to its local region (defined with respect to its parents). Within each region (leaf), the system returns a constant value (the class).
- Ok, that's nice, but where does this set of rules come from?
- The tree itself is learned from our data; the hypothesis space is the set of all trees over our attributes (large).
- It's, in fact, too large to search exhaustively (no surprise).
- Instead, we'll build the tree recursively (also no surprise) and greedily.
- That is, we'll pick the "best" split at the root that we can, without considering lower levels of the tree, fix that split, and then recurse.
- So we'll need a notion of what makes a "good" split. Thoughts?
- We'll need the following definition:
- **Definition** A data set  $S$  is said to be *pure* iff every  $x \in S$  has the same label (i.e.,  $\forall x_i, x_j \in S, y_i = y_j$ )
- So a split that improves the purity of the data set (i.e., whose children are more pure than their parent) is a good split.
- So how do we measure purity? More precisely, how do we measure *partial* purity? I.e., how do we say that one split yields children that are "more pure" than another split?
- There are a number of commonly used metrics. Probably the *most* commonly used is entropy:

$$H(S) = - \sum_{c \in \text{classes}(S)} p_c \log_2 p_c$$

- Now we'll use that to construct a criterion for deciding which attribute to split on at any given point in building a tree.
- What we want is to evaluate which attribute drives the data toward the *most pure* state. I.e., the one that improves purity most w.r.t. the original set.
- Problem: Entropy measures the purity of a *single* set,  $S$ . But after we choose an attribute  $A$  to split on, we have divided  $S$  into a number of different subsets  $S_1, S_2, \dots, S_{A.v}$ . How do we evaluate the total entropy of all of the subsets produced by  $A$ ?
- Could just sum individual entropies for each subset, but that leads to distortions – consider when an attribute pulls out only a *single* datum from the set.

- Better: entropy of a group of sets is the *average* of entropies of individual sets. Let  $S_c$  be the subset of  $S$  defined by attribute  $A = c$  ( $c \in A.v$ ). Let  $H(S|A)$  be the entropy of the split defined by  $A$ , then,

$$H(S|A) = - \sum_{c \in A.v} \frac{|S_c|}{|S|} H(S_c)$$

- Now what we want to ask is, *which attribute improves this the most over the raw entropy of  $S$ ?*
- Well, if raw entropy is  $H(S)$ , then we define the **entropy gain**,  $G(S|A) = H(S) - H(S|A)$ . Now we just pick the attribute that maximizes gain. I.e., for our split, we select  $A$  such that:

$$A = \arg \max_{p \in Attr} G(S|p)$$

- We've treated only categorical splits here, but you can also do splits on continuous values by making tests of the form  $A > 3$  vs.  $A \leq 3$ . Do this by sorting the attribute and picking the best split point within that attribute.