

# Homework

- Project: CoNLL 2004 shared task
- Homework for this week
- Homework for two weeks from today:
  - ★ Write a program which calculates the probability of each verb sense from the training data
  - ★ Don't rely on the verb sense tags, though you can use them for debugging
  - ★ Turn in program, plus sense probabilities for *exchange*,

# Committee machines

- AdaBoost is a form of forward stagewise additive regression, minimizing the the exponential loss of a model
- AdaBoost also increases the margin of the training data, reducing generalization error even after training error is zero
- Self-training and co-training use ensembles of learners to take advantage of under-annotated training data
- Committee machines generally outperform all but the best single learners, but by any measure are more complex – so what about Occam's Razor?

# Generalization

- What happened to the curse of dimensionality?
- Dimensionality (as number of features) has no clear interpretation for complex modeling procedures
- And what about simplicity? Didn't we say simpler = lower variance?
- How is an ensemble of trees simpler than a single tree?
- How is MaxEnt simpler than Naive Bayes?
- How is MaxEnt+prior simpler than MaxEnt?

# Generalization

- Notions like dimensionality and simplicity aren't really what we're interested in
- Dimensionality reduction and simplification are meant to improve *generalization* – the ability to abstract away from accidental details of a training sample
- A better way to get at that is by restricting *capacity*
- Deleting features or simplifying models may (or may not) reduce the representational capacity of a model

# Generalization

- Take MaxEnt models with a Gaussian prior:

$$\lambda^* = \operatorname{argmax}_{\lambda} L(\lambda) - \frac{1}{2\sigma^2} \sum_i \lambda_i^2$$

- This is equivalent to:

$$\lambda^* = \operatorname{argmax}_{\lambda} L(\lambda)$$

where

$$\sum_i \lambda_i^2 \leq s^2$$

- The solution is the point which maximizes  $L$  and falls inside the hypersphere with radius  $s$
- The prior reduces the capacity of the model, which (empirically) improves generalization

# Generalization

- The VC dimension of a class of functions  $\{f(x, \alpha)\}$  is the largest number of points which can be shattered by members of  $\{f(x, \alpha)\}$
- VC dimension is another way of measuring capacity, and we've already seen how this bounds generalization error.
- Given a classifier with a VC dimension of  $d$  and  $N$  training examples, then if  $d \leq N$  we have with probability  $1 - \delta$  the error rate is bounded by:

$$\frac{2}{N} \left( d \log \frac{2eN}{d} + \log \frac{2}{\delta} \right)$$

# Generalization

- What about complex adaptive procedures?
- The winning entry for the 2001 KDD Cup uses a 2–20 features, out of 140,000 (and 1,900 training examples): what's the dimensionality?
- Generalized Degrees of Freedom (Ye 1998) – randomly perturb target values  $y$  by  $\delta$ , and see how the fitted values vary
- Covariance Inflation Criterion (Tibshirani and Knight 1999) – randomly scramble target values  $y$  and see how the fitted values vary
- Both of these methods measure the flexibility of a modeling procedure

# Perceptron

- A non-linear model of a neuron used in artificial neural networks (ANNs) combines a set of weighted inputs through a biased *summing junction* and an *activation function*.
- The *McCulloch-Pitts neuron* (McCulloch and Pitts 1943) or *perceptron* (Rosenblatt 1962) is a particular neural model which uses a linear combination of its inputs:

$$v = w_0 + \sum w_i x_i$$

and whose activation function is the *threshold function* (aka: hard limiter, signum):

$$y = \begin{cases} +1 & \text{if } v \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

# Perceptron

- Rosenblatt's perceptron algorithm iteratively updates the weights  $w$  to fit a training sample
- Start with  $w = 0$
- Take each training example  $x_i$ , and compute:

$$\hat{y}_i = \text{sign}\left(\sum_j w^j x_i^j\right) = \text{sign}(w \cdot x_i)$$

- Next update weights (where  $0 < \eta \leq 1$  is the *learning rate*):

$$w \leftarrow w + \eta(y_i - \hat{y}_i)x_i$$

- Repeat until  $w$  stops changing

# Perceptron

- Small  $\eta$  gives stable weight estimates, large  $\eta$  gives fast adaptation
- If the training data is linearly separable, then this is guaranteed to converge
- If the training data is *not* linearly separable, then it may cycle infinitely among sub-optimal solutions!
- The perceptron convergence algorithm is a kind of gradient descent to maximize the *functional margin*:

$$\gamma = \min_i y_i(x_i \cdot w)$$

# Perceptron

- The functional margin  $\gamma$  is non-negative if the data is separated by the hyperplane  $w$ , and the larger  $\gamma$  is, the greater the separation
- Novikoff (1962): Suppose some weight vector  $w_0$  (where  $\|w_0\| = 1$ ) correctly classifies all examples in the training set with margin  $\gamma$ , and  $R = \max_i \|x_i\|$ . Then the number of corrections made by the perceptron algorithm is at most:

$$\left(\frac{2R}{\gamma}\right)^2$$

- The difficulty of learning a concepts depends on the the pattern length divided by the margin

# Perceptron

- What if the data is not linearly separable? We define a margin *slack variable* with respect to an example  $x_i$  and a target margin  $\gamma$ :

$$\xi_i = \max(0, \gamma - y_i(x_i \cdot w))$$

- The slack variable reflects by how much  $x_i$  fails to have a margin of  $\gamma$
- Freund and Schapire (1998): If  $D = \|\xi\|$ , then the number of mistakes made on one pass through the training data is at most:

$$\left( \frac{2(R + D)}{\gamma} \right)^2$$

# Perceptron

- The perceptron algorithm learns linear boundaries, and so can't represent many real-world concepts
- In the 1960's this led to general discouragement, followed by multi-layer perceptrons and more complex ANNs (but these have their own problems)
- Alternatively, we could use more powerful learning methods (decision trees,  $k$ -nearest neighbor)
- Or, we could just not worry about it (naive Bayes, MaxEnt)
- Or, we could find a way of converting a non-linearly separable problem into a linearly separable one by mapping between feature spaces

# Perceptron

- Suppose the concept we want to learn depends not on individual features but on combinations of features (e.g., XOR, 8)
- We could map our original feature space into a larger one which captures these relationships (*monomial features*):

$$(x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1 x_2)$$

- Alas, for an  $n$  dimensional input vector, the number of monomials of degree  $d$  is:

$$\binom{d+n-1}{d} = \frac{(d+N-1)!}{d!(N-1)!}$$

- But, there is a trick that lets us use these large derived feature spaces without actually computing them

## Perceptron (dual form)

- First we need the *dual* form of the perceptron algorithm
- The perceptron algorithm works by adding or subtracting misclassified examples from an arbitrary initial weight vector
- When it converges, the weight vector will be:

$$w = \sum_i \alpha_i y_i x_i$$

where  $\alpha_i > 0$  is the *embedding strength* of  $x_i$ , proportional to the number of times misclassification of  $x_i$  caused the weights to be updated

- Given a fixed training set, the solution can be represented either by  $w$  or by  $\alpha$

# Perceptron (dual form)

- We rewrite the decision rule as:

$$\begin{aligned}\hat{y} &= \text{sign}(w \cdot x) \\ &= \text{sign}\left(\sum \alpha_i y_i x_i\right) \cdot x \\ &= \text{sign}\left(\sum \alpha_i y_i (x_i \cdot x)\right)\end{aligned}$$

- And the update of the perceptron algorithm becomes:

If

$$y_i \left( \sum_j \alpha_j y_j (x_j \cdot x_i) \right) \leq 0$$

then

$$\alpha_i \leftarrow \alpha_i + \eta$$

# Kernel functions

- Notice now that the training data only comes into play via the *Gram matrix*  $G$ :

$$G_{ij} = x_i \cdot x_j$$

- The dot product is one measure of the similarity between  $x_i$  and  $x_j$ , but there are others
- In general, we want to define a mapping  $\Phi$  from our feature space into  $\mathbb{R}^N$ , and a *kernel function*:

$$k(x, x') = \Phi(x) \cdot \Phi(x')$$

# Kernel functions

- In the case of an ordered *polynomial kernel*, we have:

$$\Phi : (x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1 x_2, x_2 x_1)$$

and

$$\Phi(x) \cdot \Phi(x') = x_1^2 (x'_1)^2 + x_2^2 (x'_2)^2 + 2 x_1 x_2 x'_1 x'_2 = (x \cdot x')^2$$

- For polynomial kernels in general:

$$\Phi_d(x) \cdot \Phi_d(x') = (x \cdot x')^d$$

# Kernel functions

- Thus, we can work with a very high dimensional ‘virtual’ space with essentially the same amount of effort as in the original feature space
- Mercer’s Theorem states that any kernel function which meets certain general conditions can be represented as a dot product in a high dimensional space
- This leads to the *kernel trick*: any algorithm which uses a dot product can be rewritten to use a Mercer kernel instead
- A ‘kernelized’ perceptron algorithm can fit a hyperplane to a non-linear mapping of a feature space with little or no extra computational cost
- But, we still pay a statistical cost for those extra features (there’s no escaping the curse)