

# CSE 6740 Lecture 8

## *How Do I Predict a Discrete Variable? II (Classification)*

Alexander Gray

agray@cc.gatech.edu

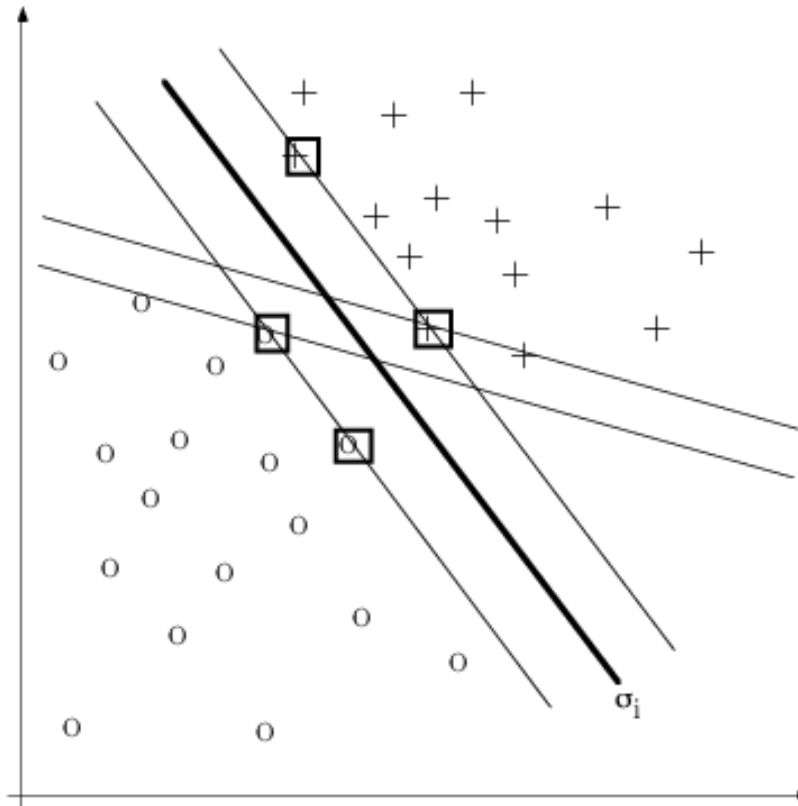
Georgia Institute of Technology

# Today

1. More classification methods (*How can I predict a discrete variable?*)

# Support Vector Machine

Now let's choose a different criterion. Let's find the hyperplane which maximizes the distance of the closest point from either class. We call this distance the *margin*. Points on the margin are called *support vectors*. Let's begin by assuming the classes are linearly separable.



# Support Vector Machine

The hyperplane which maximizes the margin is given by finding

$$\max_{\beta_0, \beta} m \quad \text{subject to} \quad \frac{1}{\|\beta\|} y_i (\beta_0 + \beta^T x_i) \geq m, \forall i. \quad (1)$$

Equivalently the constraints can be written as

$y_i (\beta_0 + \beta^T x_i) \geq m \|\beta\|$ . Since for any  $\beta_0$  and  $\beta$  satisfying these inequalities, any positively scaled multiple satisfies them too, we can arbitrarily set  $\|\beta\| = 1/m$ .

# Support Vector Machine

Thus the optimization problem is equivalent to minimizing

$$\frac{1}{2} \|\beta\|^2 \quad \text{subject to} \quad y_i(\beta_0 + \beta^T x_i) \geq 1, \forall i. \quad (2)$$

It turns out this optimization problem is a *quadratic programming* problem (quadratic objective function with linear constraints), a standard type of optimization problem for which methods exist for finding the global optimum. The theory of convex optimization tells us there is an equivalent way to write this optimization problem (its *dual formulation*).

# Support Vector Machine

Let  $g^*(x)$  denote the optimal (maximum margin) hyperplane. Let  $\langle x_i x_{i'} \rangle$  denote the inner product of  $x_i$  and  $x_{i'}$ . Then

$$\beta_j^* = \sum_{i=1}^N \alpha_i y_i x_{ij} \quad (3)$$

where  $\alpha$  is the vector of weights that maximizes

$$\sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \langle x_i x_{i'} \rangle \quad (4)$$

$$\text{subject to } \alpha_i \geq 0 \quad \text{and} \quad \sum_i \alpha_i y_i = 0. \quad (5)$$

# Support Vector Machine

However, for realistic problems we must relax the assumption that the classes are linearly separable. In the primal formulation, instead of minimizing

$$\frac{1}{2} \|\beta\| \quad \text{subject to} \quad y_i(\beta_0 + \beta^T x_i) \geq 1, \forall i \quad (6)$$

we'll now minimize

$$\frac{1}{2} \|\beta\| \quad \text{subject to} \quad y_i(\beta_0 + \beta^T x_i) \geq 1 - \xi_i, \forall i, \quad (7)$$

where the  $\xi_i$  are called *slack variables* and we limit the amount of slack by adding the constraints

$$\xi_i \geq 0 \quad \text{and} \quad \sum_i \xi_i \leq C. \quad (8)$$

# Support Vector Machine

This effectively bounds the total number of misclassifications at  $C$ , which becomes a tuning parameter of the support vector machine.

The points  $x_i$  for which  $\alpha_i \neq 0$  are the support vectors.

The discriminant function can be rewritten as

$$g(x) = \beta_0 + \sum_{i=1}^N \alpha_i y_i \langle x, x_i \rangle \quad (9)$$

and the final classification rule is  $\hat{c}(x) = \text{sgn}g(x)$ .

# Support Vector Machine

It turns out the SVM optimization is equivalent to minimizing

$$\sum_{i=1}^N (1 - y_i g(x_i))_+ + \lambda \|\beta\|^2, \quad (10)$$

where  $\lambda$  plays the role of  $C$ . This is a fairly robust loss function, called the *hinge loss*, and we have a regularization parameter allowing us to control complexity. We can also use an  $L_1$  regularization.

# Support Vector Machine

**Task:** classification

**Model class:** all possible linear classifiers (parametric)

**Loss:** Hinge loss, with regularization term

**Optimizer:** Constrained optimization (quadratic program):  
SMO, interior-point

**Generalization mechanism:** cross-validation

# $\kappa$ -Nearest-Neighbor Classifier

Now let's turn to a completely nonparametric classification method, perhaps the simplest possible one: the *nearest-neighbor rule*:

$$\hat{c}(x) = c(\arg \min_i \|x - x_i\|), \quad (11)$$

*i.e.* use the class label of the nearest point. The intuitive justification is that  $\mathbb{E}(Y|X = x) \approx \mathbb{E}(Y|X = x')$  if  $x'$  is very close to  $x$ .

We can add the equivalent of a smoothing parameter,  $\kappa$ , which regulates complexity, by using the majority label among the  $\kappa$  nearest neighbors.

# $\kappa$ -Nearest-Neighbor Classifier

Let  $E^* = \mathbb{E}[L(Y, c(X))]$  where  $c(X)$  is the (optimal) Bayes rule, and let  $k^*$  be the true class for  $x$ . Then if  $\hat{c}(x)$  is the 1-NN rule,

$$E^* = 1 - \mathbb{P}(Y = k^* | X = x) \quad \text{and} \quad (12)$$

$$\mathbb{E}[L(Y, \hat{c}(X))] = \sum_k^K \mathbb{P}(Y = k | X = x) (1 - \mathbb{P}(Y = k | X = x)) \quad (13)$$

$$\geq 1 - \mathbb{P}(Y = k^* | X = x) \quad (14)$$

$$\leq 2(1 - \mathbb{P}(Y = k^* | X = x)) \quad \text{for } K = 2. \quad (15)$$

In other words, asymptotically the error of the 1-NN rule is no more than twice the optimal error rate.

# $\kappa$ -Nearest-Neighbor Classifier

There is a  $\kappa$ -NN method for density estimation, and one for regression. However, these are both problematic.

$\kappa$ -NN rules can be seen as approximations to kernel estimators.

Often very easy to apply to new problems – just define a distance metric.

**Task:** classification

**Model class:** nonparametric

**Loss:** 0-1 loss

**Optimizer:** exhaustive or gradient descent

**Generalization mechanism:** cross-validation

# Decision Trees

Here's another nonparametric method, which has variations for different tasks (*classification trees, regression trees, density trees*).

Let's consider the regression case first, using squared error  $\sum_i (y_i - \hat{f}(x_i))^2$ .

# Decision Trees

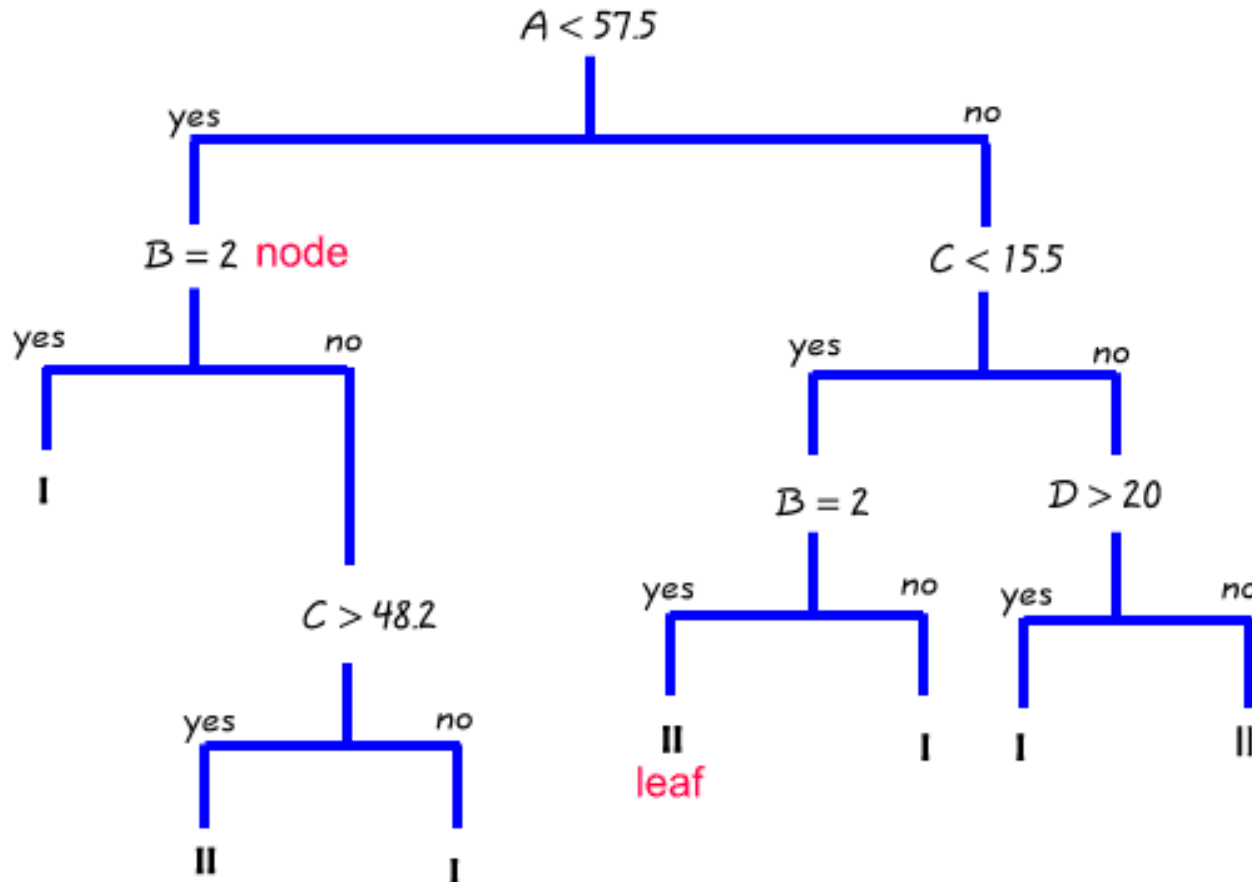
Each node is a hyperrectangular subset of the feature space, so that the overall model is of the form

$$\hat{f}(x) = \sum_m c_m I(x \in R_m). \quad (16)$$

We want hyperrectangles which try to contain points all having similar  $Y$  values. Given hyperrectangles, it's easy to see that the best  $\hat{c}_m$  is the average in the node,  $\hat{c}_m = 1/N_m \sum_i y_i I(x_i \in R_m)$ , or  $\hat{c}_m = 1/N_m \sum_{x_i \in R_m} y_i$ .

# Decision Trees

Finding the optimal set of hyperrectangles is generally intractable computationally, so we'll proceed greedily, chopping up the input space one dimension at a time:



# Decision Trees

Starting with all the data, consider a splitting variable  $d$  and split point  $s$ , and define the pair of half-planes

$$R_1(d, s) = \{X | X_d \leq s\} \quad \text{and} \quad R_2(d, s) = \{X | X_d > s\}. \quad (17)$$

Then we seek the splitting variable  $d$  and split point  $s$  corresponding to

$$\min_{d,s} \left( \min_{c_1} \sum_{x_i \in R_1(d,s)} (y_i - c_1)^2 \right. \quad (18)$$

$$\left. + \min_{c_2} \sum_{x_i \in R_2(d,s)} (y_i - c_2)^2 \right). \quad (19)$$

# Decision Trees

For any choice of  $d$  and  $s$ , the inner minimization is solved by

$$\hat{c}_1 = 1/N_1 \sum_i y_i I(x_i \in R_1) \quad (20)$$

$$\hat{c}_2 = 1/N_2 \sum_i y_i I(x_i \in R_2). \quad (21)$$

To minimize over  $d$  and  $s$ , we can evaluate the inner expression over all possible choices. This can be done relatively efficiently by running over the sorted values of  $Y$ .

# Decision Trees

We can keep doing this recursively in a top-down fashion. Then we need to determine when to stop splitting nodes.

Equivalently, we can build the tree to completion (the tree  $T_0$  where every data point is a leaf, say), then decide which nodes to collapse to obtain a pruned tree  $T \subset T_0$ .

# Decision Trees

Since the number of leaves of the tree  $|T|$  can be thought of as a complexity parameter, we can use to obtain a regularization of the training set error:

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - c_m)^2 \quad (22)$$

$$C_\lambda(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \lambda |T|. \quad (23)$$

We can then cross-validate over  $\lambda$  to minimize the *cost-complexity* criterion  $C_\lambda(T)$ .

If more than one subtree minimizes this criterion, we use the smallest one.

# Decision Tree

For classification, we simply minimize a classification loss instead, such as the cross-entropy (recall that this corresponds to maximum likelihood with a certain error model). First define

$$\hat{p}_{mc} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = c), \quad (24)$$

the proportion of class  $c$  observations in node  $m$ . We classify the observations in node  $m$  as  $\hat{c}(m) = \arg \max_c \hat{p}_{mc}$ , the majority class in node  $m$ .  $Q_m(T)$  is now

$$Q_m(T) = - \sum_c \hat{p}_{mc} \log \hat{p}_{mc}. \quad (25)$$

# Decision Tree

Note that the actual form of the model is a set of *rules*, which are conjunctions of univariate tests, resulting in a piecewise constant model. The way of obtaining the rules happens to be a computationally efficient (but suboptimal) algorithm.

**Task:** classification

**Model class:** nonparametric

**Loss:** likelihood, with regularization term

**Optimizer:** top-down greedy split selection

**Generalization mechanism:** cross-validation

# Main Things You Should Know

- What a support vector machine is
- What a nearest-neighbor classifier is
- What a decision tree is

# Sample Final Questions

1. (T/F) What is the loss function for a support vector machine?
2. (T/F) A learned decision tree represents the optimal tree structure under a certain loss function.
3. (T/F) The nearest-neighbor rule is a nonparametric classifier.