6.034 Exam 2 Cheat Sheet

Identification Trees

Use training data to determine series of tests that determine classification

How to Build an ID tree - Steps

1. Partition data into groups based on the value of a particular feature
2. Choose test with the lowest disorder

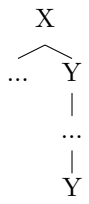
$$\text{Average Disorder} = \sum_{b \in \text{ branches}} \frac{\#\text{ of training points in } b}{\text{total } \#\text{ of training points}} \times \text{Disorder}(b)$$

$$\text{Disorder}(b) = -\sum_{y \in \text{ classifications}} \frac{\#\text{ of training points in } y}{\#\text{ of training points in } b} \log_2 \left(\frac{\#\text{ of training points in } y}{\#\text{ of training points in } b}\right)$$

3. Keep doing this until all the training points are correctly classified

Notes

- more complex trees could be a sign of over fitting
  - or, tests rely too much on outliers to classify data
- within the same subtree, don't use the same test twice



Y doesn't give you any new information

- however, the following is perfectly fine

- (one branch) - this is the same basically as not running a test (should never occur)

K Nearest Neighbors

Training – Store all feature vectors in the training set, along with each class label.

Prediction – Given a query feature vector, find "nearest" stored feature vector and return the associated class.

1-NN: Given an unknown point, pick the closest 1 neighbor by some distance measure. Class of the unknown is the 1-nearest neighbor's label.

k-NN: Given an unknown, pick the k closest neighbors by some distance function. Class of unknown is the mode of the k-nearest neighbor's labels. *k is usually an odd number to facilitate tie breaking.*

Normalization: To separate values clustered close together, divide by the standard deviation

Relevant features: All features are used; to find relevant ones, have to cross-validate, dropping features out.

What's the k?: Can find best value using cross-validation Voting for vectors? k-Nearest Neighbors votes on class for query feature vector; reduces sensitivity to noise

k-NN fixes a set of decision boundaries for whether a point is/is not in a given class.

Distance Metrics

Euclidean Distance (common)

$$D(\vec{w}, \vec{v}) = \sqrt{\sum_{i}^{n} (w_i - v_i)^2}$$

Manhattan Distance (Block distance) - Sum of distances in each dimension

$$D(\vec{w}, \vec{v}) = \sqrt{\sum_{i}^{n} \mid w_i - v_i \mid}$$

Hamming Distance (for non-numerical data) - Sum of differences in each dimension

$$D(\vec{w}, \vec{v}) = \sqrt{\sum_{i}^{n} I(w_i, v_i)}$$

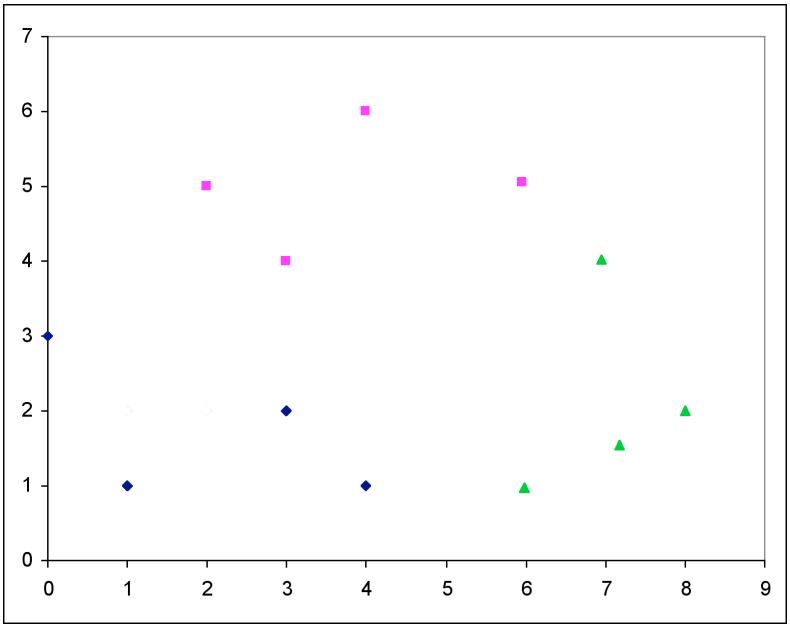
$$I(x, y) = \begin{cases} 1 & \text{if different} \\ 0 & \text{if identifical} \end{cases}$$

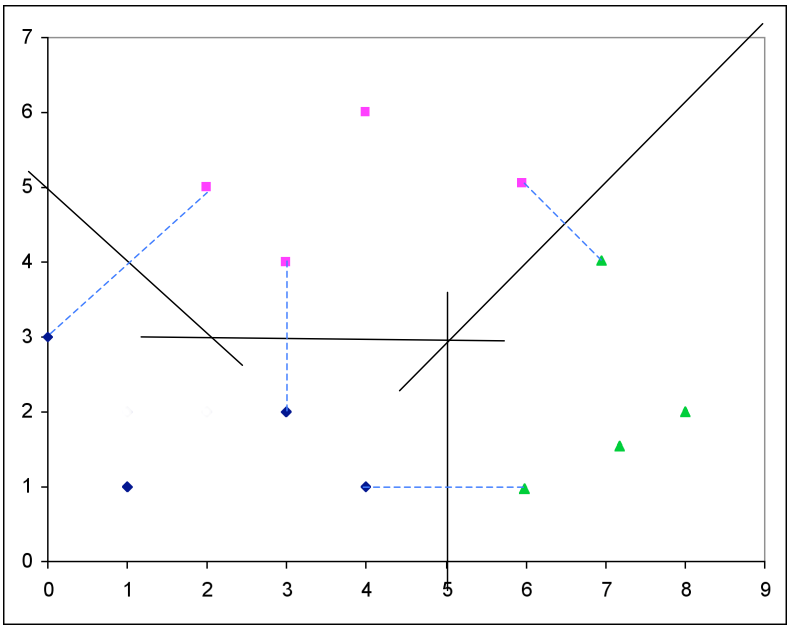
Cosine Similarity - Used in Text classification; words are dimensions; documents are vectors of words; vector component is 1 if word i exists.

$$D(\vec{w}, \vec{v}) = \frac{\vec{w} \cdot \vec{v}}{\mid \vec{w} \mid\mid \vec{v} \mid} = \cos(\theta)$$

6.034 Recitation October 20: Nearest Neighbors, Drawing decision boundaries

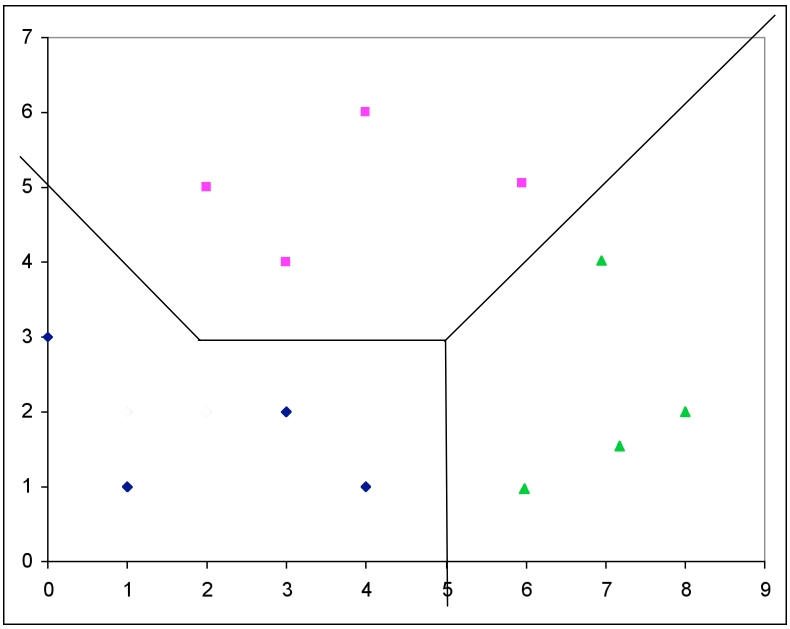
Boundary lines are formed by the intersection of perpendicular bisectors of every pair of points. Using pairs of closest points in different classes gives a good enough approximation. (To be absolutely sure about the boundaries, one would draw perpendicular bisectors between each pair of neighboring points to create a region for each point, then consolidate regions belonging to the same class, i.e., remove the boundaries separating points in the same class. This technique is unnecessary for our purposes.)





Construct lines between closest pairs of points in different classes.

Draw perpendicular bisectors.



End bisectors at intersections; extend beyond axes (to infinity).

These notes are available at http://web.mit.edu/dxh/www/6034-constraint.odt

6.034 Notes: Four strategies for Constraint Propagation

We typically solve constraint satisfaction problems by drawing a search tree whose nodes contain partial solutions. That way, the task of solving the problem becomes the task of searching a tree —each branch represents a value that we tentatively assign to a variable; we later backtrack if we find that our chosen assignment cannot work.

To make the search more intelligent, we try to prune as many branches as we can from the tree, so that we avoid wasting time with dead ends.

In 6.034, we study four different strategies for pruning branches while you're searching for a solution. In **increasing order of power**, (but also increasing order of the amount of extra work you have to do), the strategies are:

1. Depth first search only.
2. Depth first search + forward checking
3. Depth first search + forward checking + propagation through singleton domains
4. Depth first search + forward checking + propagation through reduced domains

To elaborate:

1. Depth first search only.
This strategy is the most basic approach; **it doesn't prune any branches at all**. The only check it makes is that all the assigned values so far are consistent with each other.

To perform depth first search only:

1. [DFS] After you assign a value to a variable, examine all of the variables you've assigned values so far, and make sure those values are consistent with the constraints. If they aren't, backtrack.

2. Depth first search + forward checking
This strategy eliminates impossible options from neighboring variables.

To perform dfs+forward checking:

1. [DFS] After you assign a value to a variable, examine all of the variables you've assigned values so far, and make sure those values are consistent with the constraints. If they aren't, backtrack.
2. [FC] After you assign a value to a variable, consider all of its neighbors. Eliminate any options from its neighbors that are incompatible with the value you just assigned.

3. Depth first search + forward checking + propagation through singleton domains

This strategy eliminates impossible options from neighboring variables. Then, if any of those neighboring variables have only one option left, it looks ahead to see what else it can eliminate.

1. [DFS] After you assign a value to a variable, examine all of the variables you've assigned values so far, and make sure those values are consistent with the constraints. If they aren't, backtrack.

2. [FC] After you assign a value to a variable, consider all of its neighbors. Eliminate any options from its neighbors that are incompatible with the value you just assigned.

3. [PROP-1] If you eliminate options from a neighbor in the previous step, and that neighbor has only one option left, add that neighbor to the list of variables to propagate. Propagate all the variables in the list.

To *propagate a singleton variable*, take note of its one remaining option and consider each of its neighbors. You want to cross off values in the neighboring variables that are incompatible with the one remaining option. Then, if you cross off options in a neighboring variable, and that neighbor has only one option left, add that neighbor to the list of variables to propagate.

4. Depth first search + forward checking + propagation through reduced domains

This strategy eliminates impossible options from neighboring variables. If it successfully eliminates any options from those variables, it looks ahead to see what else it can eliminate.

1. [DFS] After you assign a value to a variable, examine all of the variables you've assigned values so far, and make sure those values are consistent with the constraints. If they aren't, backtrack.

2. [FC] After you assign a value to a variable, consider all of its neighbors. Eliminate any options from its neighbors that are incompatible with the value you just assigned.

3. [PROP-ANY] If you eliminate any options from a neighbor in the previous step, add that neighbor to the list of variables to propagate. Propagate all the variables in the list.

To *propagate a reduced variable*, take note of its remaining options and consider each of its neighbors. You want to cross off values in the neighboring variables that are incompatible with every one of the remaining options. If you cross off any values in a neighboring variable, add that neighbor to the list of variables to propagate.

Note 1: Notice that none of these pruning options ever assign values to a variable; they only remove values from consideration.

Note 2: There's always an implicit "escape clause" in a constraint satisfaction problem, namely:

> If you ever eliminate *all* the options from a variable, then you can't solve the problem with the assignments you've made so far. Backtrack.

*** Domain reduction before search begins**

This procedure eliminates impossible options from domains before you even begin searching and assigning values to variables. Like strategy (4), it uses the subroutine for *propagating reduced variables.*

1. Add all the variables in the problem to the list of variables to propagate.

2. Until the list is empty, propagate each variable in the list.

3. [PROP-ANY] If you eliminate any options from a neighbor in the previous step, add that neighbor to the list of variables to propagate. Propagate all the variables in the list.

As before, to *propagate a reduced variable*, take note of its remaining options and consider each of its neighbors. You want to cross off values in the neighboring variables that are incompatible with every one of the remaining options. If you cross off any values in a neighboring variable, add that neighbor to the list of variables to propagate.

###Various Notes

- backtracking in all 4 domains

- usually it's a decent idea to try to start with the most constrained variable first, or the variable with the smallest domain
- amount of backtracking in Prop-Any $\leq$ amount of backtracking in DFS
- amount of variable assignments in Prop-Any $\leq$ amount of variable assignments in DFS