MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034—Artificial Intelligence
Recitation 8, November 2, 2001

Agenda

1. Learning: Types of problems; classification vs. regression problems
2. Representation issues
3. Features and Nearest Neighbors: Note the way the plane gets tiled
4. Learning: ID trees and information; from trees to rules
5. Learning: information theory
6. ID tree example

Learning: general issues

One general view of learning is this: finding a *function* based on past inputs or *examples* that can be used to *predict* future inputs. We are given a bunch of data with some features, and we want to predict some feature of interest:

1. Learning how to pronounce words
2. Learning how to throw a ball, ride a bicyle, play piano
3. Learning how to diagnose diseases, predict bankruptcy, etc.
4. Learning how to solve math problems
5. Learning how to predict movie choices

One way to carve up types of learning problems depends on the following general factors (applicable to all learning situations):

1. Representational adequacy: you can't learn something if you can't represent it.
2. Complexity control: how to avoid overfitting (predict 5 points with 5 parameters)
3. Feature choice

There are also ways to classify learning problems themselves, depending on:

1. The *kind* of data available for learning — *positive* examples or positive and *negative* examples (target is KNOWN).

2. The representation of the function itself — as points, a net, and so forth.
3. The availability of prior information
4. The type of prediction to be made — *discrete* (like a classification), or *continuous*.

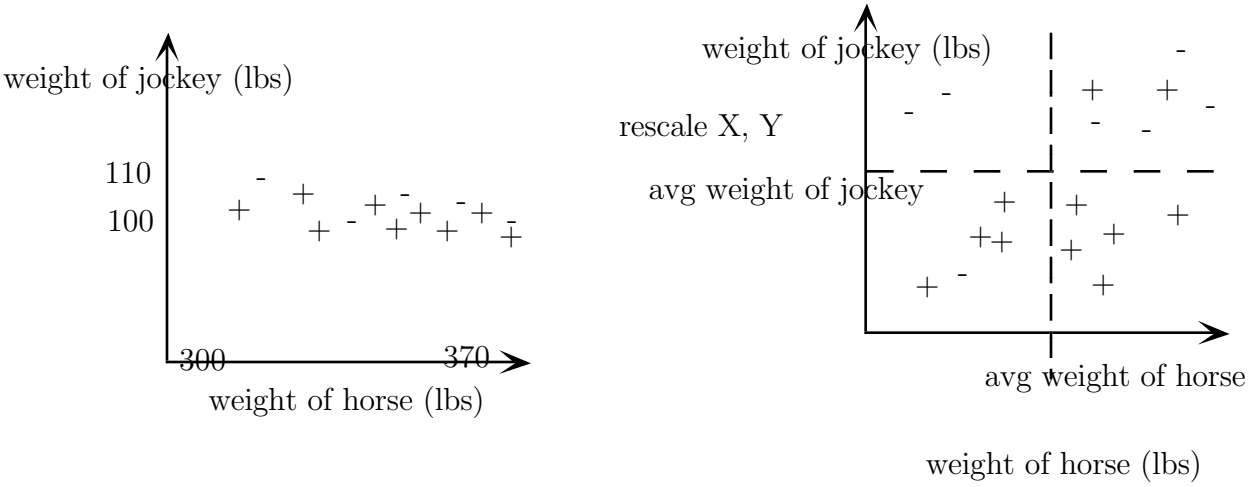
KEY QUESTION: HOW TO REPRESENT WHAT IS LEARNED

Learning: features and nearest neighbors

The *choice* of features determines whether a function may be learned by a representation. If the input is not correlated with the desired output, nothing can be learned. Even so, if a correlation exists, it might not be captured by the particular *feature choice*, or, even with a good set of features, *scaling* can matter.

Example of feature choice: polar vs. (x, y) coordinates.

Example of *scaling*: Predict outcome of horse race based on weight of horse and weight of jockey.



Nearest neighbors

KEY ASSUMPTION: The **stereotype** principle — THINGS THAT ARE ALIKE IN A FEW WAYS ARE ALIKE IN ALL OTHER WAYS.

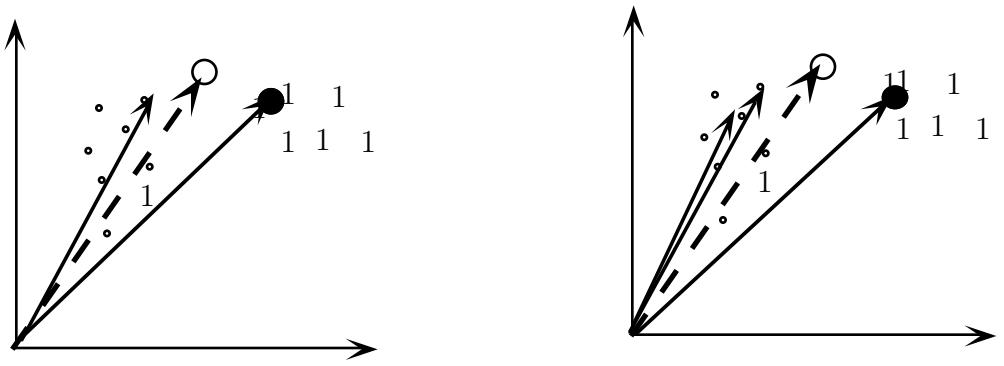
Application: Supervised learning/classification

Training: Store *all* feature vector points in the training set, each labeled with its class.

Prediction: Given a *new* feature vector, find the "nearest" stored feature vector and return the result. Note the way it "tiles" the plane.

Advantages: Fastest possible training. Little bias — little preconception about the function that works. All features assumed equal — what does this imply about scaling?

Disadvantages: How to pick the distance metric (how to define *near*). No feature selection or extrapolation (doesn't do generalization). Prediction is costly when there are many features (but we can speed this up). Also sensitive to errors.



Extensions:

1. To reduce sensitivity to noise: pick k neighbors and have them vote, e.g., 3 nearest neighbors. This is k-nearest neighbors method. (see figure).
2. To speed up time for prediction, use a different type of representation (see next lecture: $K-D$ trees, splitting data along selected feature values.
3. To choose k and the metric, can do *cross-validation*: use one bunch of data for training, used to choose the free parameters, and another set is held back, and used for *validation*, used to estimate predictive performance.

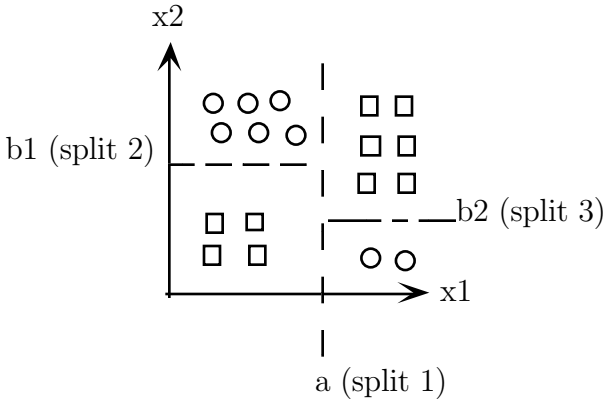
   Cross-validation: divide training data into m subsets. Cycle through these subsets using each subset for the validation set, and the remaining subsets for the training set. The validation set is thus $1/m$ of the data, training set is $(m-1)/m$ of the data. Then we average the performance on each of the m subsets and use this as the estimated prediction performance.

   Bootstrap: Treat the training data as representing the actual input distribution, so sample from this distribution, with replacement, to obtain training and validation sets. Use average and deviation to characterize performance, along with confidence bounds.

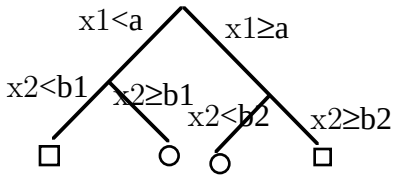
Classification or ID Trees

Strategy: Divide feature space into boxes that are uniform with respect to labels. Do splitting along each axis, recursively, to define a *tree*.

Example:



And the resulting tree is:



More "practical" example: predicting bankruptcy based on late payments, L, expenses/income, R, and bankruptcy within the past year, B (the variable we want to predict).

The simplest tree has a better chance of *generalizing*, i.e., it correctly classifies *unseen* cases.

Problem: Finding the simplest tree is computationally intractable.

Solution: Do a *greedy* (local hill-climbing) search for the simplest tree.

1. Greedy-Tree-Builder (DATA)

a. av= Pick-best-attribute/value-to-split (DATA)

b. split-list= Split-data(DATA, av)

c. For each subset in split-list, Greedy-Tree-Builder(DATA)

2. Pick-best-attribute/value-to-split (DATA)

a. Try every attribute/value combination

b. Pick one that produces **least average disorder** in split.

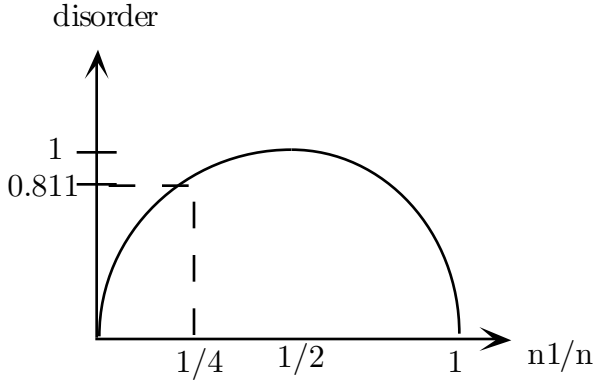
(So we have to have a way of measuring disorder). Intuition is to split as uniformly as possible.

How can we make that intuition precise?

If we split a group of m objects into two bins of size m_1 and m_2 then what is the *disorder* of the resulting split?

1. If all objects wind up in one bin, then the disorder is 0.
2. If half the objects end up in one bin and the other half in the other bin, then the disorder is maximal, call this 1.
3. The disorder measure should be symmetrical — that is, it doesn't matter whether all the objects wind up in bin 1 or bin 2, the disorder should still be 0.

So we want a function that looks like this:



A measure that yields this behavior is:

$$-\frac{m_1}{m}\log_2\frac{m_1}{m} - \frac{m_2}{m}\log_2\frac{m_2}{m} \Rightarrow$$
$$\Sigma - \frac{m_i}{m}\log_2\frac{m_i}{m}$$

Define P_i= the probability of being in bin i, as m_i/m. Then disorder or *entropy* is simply

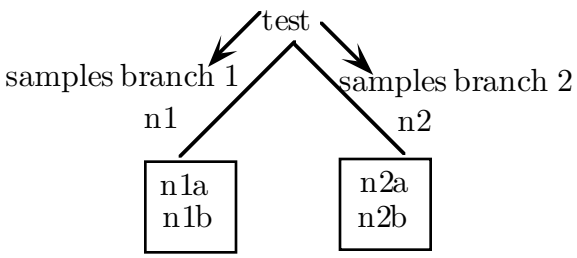
$$\Sigma - p_i \log_2 p_i$$

where $\Sigma p_i = 1$ and we must define $0 \log 0 = 0$.

Table:

p_1	p_2	result
1	0	$-1\log 1 - 0\log 0 = 0 + 0 = 0$
1/2	1/2	$-\frac{1}{2}\log\frac{1}{2} - \frac{1}{2}\log\frac{1}{2} = 1$
1/4	3/4	$-\frac{1}{4}\log\frac{1}{4} - \frac{3}{4}\log\frac{3}{4} = 0.5 + 0.311 = 0.811$

Suppose are data are labeled as either a or b (say, good or bad). Suppose further that n_t = the total number of data points (samples). If we apply a particular (binary) test to all of these data, then some fraction of the samples, n_1, will fall down one side of the pachinko machine branch, and some fraction n_2 down the second branch. Of those that go down the first branch, some fraction, say n_{1a} will be actually of type a, while the rest, n_{1b}, will be of type b. (In general, let us say there will be k branches).



Then we can measure *average disorder* as the product of two factors:

(1) the fraction of samples that go down a particular branch × disorder of the class distribution on that branch, or,

$$\Sigma_{l=1}^{k}(\frac{n_l}{n_t}) \times (\Sigma_{c\in\text{class}} - \frac{n_{lc}}{n_b}\log_2\frac{n_{lc}}{n_b})$$

The resulting trees are called...

Classification Trees

Some classification of classification trees, and then an example.

Discrete Attributes – with k values

This is simply a k-way split in a tree. Watch out: for attributes that yield large k values, their disorder measure will be artificially low (why?).

Continuous attributes

How to handle?

- Option 1: Quantized by user
- Option 2: Binary split between each attribute value in the training set — $O(n)$ splits for n samples. (For example, if some sample has temperature 90, then we need both less than 90 and ge 90.)
- Option 3: Three-way split, that is, consider pair of attribute values at a time, so $O(n^2)$ splits to consider. This can lead to fewer intervals.

Missing values

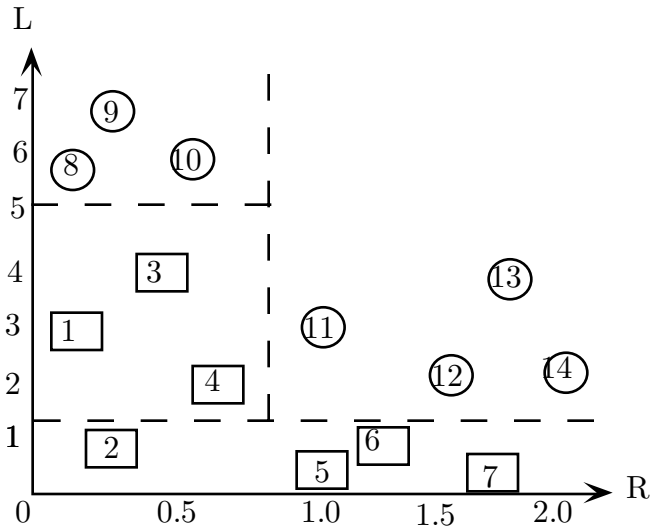
Most data sets of have samples with missing attributes. How to handle? One way: treat such a sample as splitting into fractional samples that go down *all* branches of a test in proportion to the frequency of the different values of the attribute data set being considered. (E.g., if just two values, then 50-50 split — like assuming uniform distribution.)

Classification trees - example

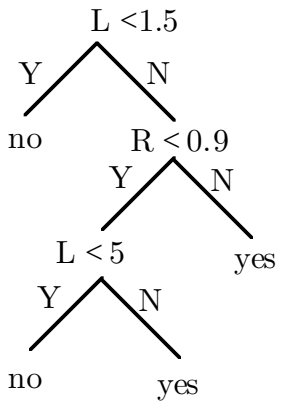
Predicting bankruptcy, B, based on L, number of late payments/year, and R= expenses/income.

Sample	L	R	B
1	3	0.2	No
2	1	0.3	No
3	4	0.5	No
4	2	0.7	No
5	0	1.0	No
6	1	1.2	No
7	0	1.7	No
8	6	0.2	Yes
9	7	0.3	Yes
10	6	0.7	Yes
11	3	1.1	Yes
12	2	1.5	Yes
13	4	1.7	Yes
14	2	1.9	Yes

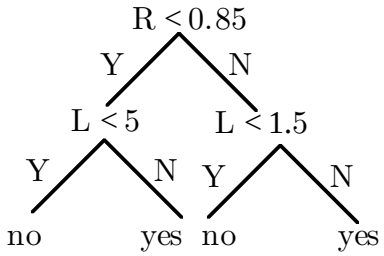
Picture:



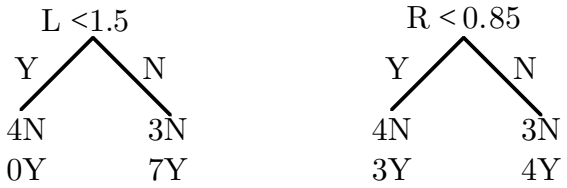
Tree found by greedy algorithm:



A better tree, or at least equally good:



Note that the split $R < 0.85$ has much higher disorder than $L < 1.5$. The R split has an average disorder of 1.0, while the L split has an average disorder of 0.436:



Classification trees — overfitting issues

If one keeps subdividing data until every leaf is uniform in its own class, we will *overfit* — we will characterize the noise in the data rather than any underlying regularity. (Every data point winds up in its own bin.)

As an example, consider building a decision tree to predict the outcome of the role of a die based on the following attributes:

1. Month
2. Day
3. Minute
4. Dow Jones average
5. Windspeed
6. Temperature

We can build a tree where every roll of the die winds up in a unique leaf — but this is worthless.

If you think this is easy to watch out for, consider the example of a medical database with Social Security numbers — somebody once trained a system using this "feature" — it is very informative, but useless for medical diagnosis.

We want to measure the ability of a tree to *predict unseen data* — not just fit the training data.

So we want to either:

- Include a stopping rule from growing the number of classes
- Prune the fully formed trees

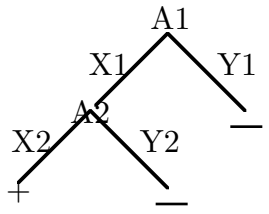
(More generally, we want to penalize for the SIZE of the the "theory" we produce — Occam's razor.

In both cases, we want to test the increase in predictive performance of having an extra test in the tree. One way to do this is use a *hold-out* set for such tests (data that we didn't train on), or to do *cross-validation* on the training set to test the predictive performance. Here's a repeat of cross-validation:

Cross-validation: divide training data into m subsets. Cycle through these subsets using each subset for the validation set, and the remaining subsets for the training set. The validation set is thus $1/m$ of the data, training set is $(m-1)/m$ of the data. Then we average the performance on each of the m subsets and use this as the estimated prediction performance.

From trees to rules

Given a tree, we can construct if-then rules that perform the equivalent classification:



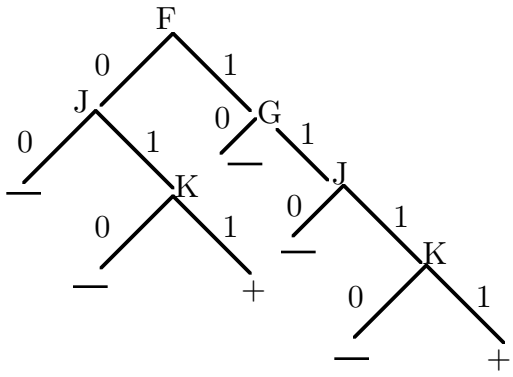

```
R1: IF A1=X1          R2: IF A1=X1          R3: IF A1=Y1
       A2=X2                 A2=Y2                THEN -
    THEN +                THEN -
```

HOWEVER, rules are a more general framework than trees; there are rules that cannot be modeled as trees.

Consider a data set where every *positive* sample has value 1 for features F and G **or** J and K. So a set of rules to classify this data would be:

```
R1: IF F=1          R2: IF J=1          R3: IF F=0        R4:  IF F=0   etc.
       G=1                 K=1                J=0                K=0
    THEN +              THEN +              THEN -             THEN -
```

BUT we cannot capture this in a tree. The best one can do is a tree where the tests are ordered:



However, one can construct rules from this tree:

```
IF  F=1
    G=0
    J=1
    K=1
THEN +
```

We can drop tests that don't make any difference, to get the simplified rule:

```
IF  J=1
    K=1
THEN +
```