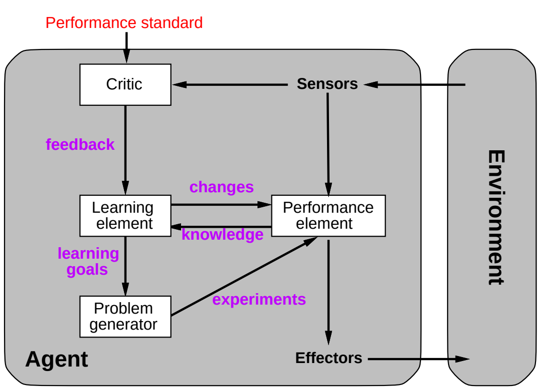
LEARNING FROM EXAMPLES

Overview

- This last section of the course will be on learning.
  - Machine learning
- Lots of different views of what learning is.
  - Already saw some ideas in the guest lecture.
- Today we'll look at another kind of learning
  - Different technique(s), similar scope
- Next week will look at something rather different.

Learning agents



- Key point is that the agent looks at how it performs and modifies this.

- Design of learning element is dictated by
  - what type of performance element is used
  - which functional component is to be learned
  - how that functional component is represented
  - what kind of feedback is available
- Changing components gives different kinds of learning.

- Examples of representations/performance element
  - Lookup table, genetic algorithm, genetic program, neural network.
- Examples of adjustment methods/learning element
  - Evolutionary learning, reinforcement learning, statistical learning
- Methods for evaluating the candidate/feedback/critic
  - Supervised learning, unsupervised learning

What we will look at

- *Supervised learning*
  - Correct answers for each instance.
  - Modify the performance element to give correct answers
- In particular we will look at an approach to classification.
- *Reinforcement learning*
  - Occasional rewards
  - Need to associate actions with the rewardsthey bring.
- We will look at learning in the framework of MDPs.

Inductive learning

- Simplest form: learn a function from examples (*tabula rasa*)
- f is the *target function*
- An *example* is a pair $x, f(x)$:

$$\begin{array}{c|c|c} O & O & X \\ \hline & X & \\ \hline X & & \end{array} \quad , \quad +1$$

- Problem: find a *hypothesis* h such that

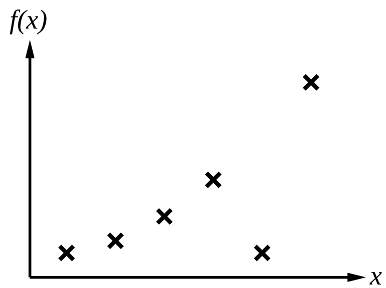
$$h \approx f$$

given a *training set* of examples

Inductive learning method

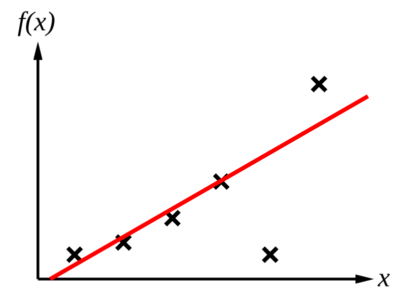
- Construct/adjust h to agree with f on training set
  (h is *consistent* if it agrees with f on all examples)

  E.g., curve fitting:



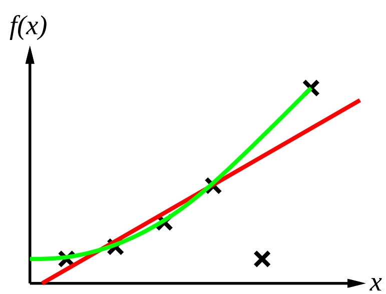
- Construct/adjust h to agree with f on training set
  (h is *consistent* if it agrees with f on all examples)

  E.g., curve fitting:



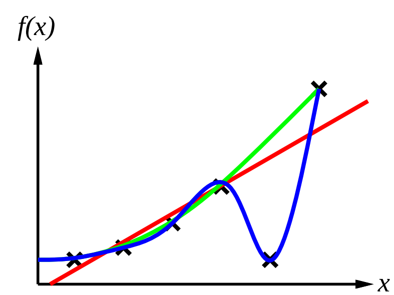
- Construct/adjust h to agree with f on training set
  (h is *consistent* if it agrees with f on all examples)

  E.g., curve fitting:



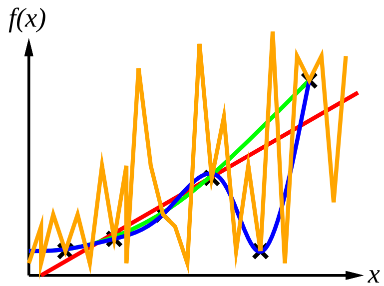
- Construct/adjust h to agree with f on training set
  (h is *consistent* if it agrees with f on all examples)

  E.g., curve fitting:



- Construct/adjust h to agree with f on training set
  (h is *consistent* if it agrees with f on all examples)

  E.g., curve fitting:



- Ockham's razor: maximize a combination of consistency and simplicity

William of Ockham

Attribute-based representations

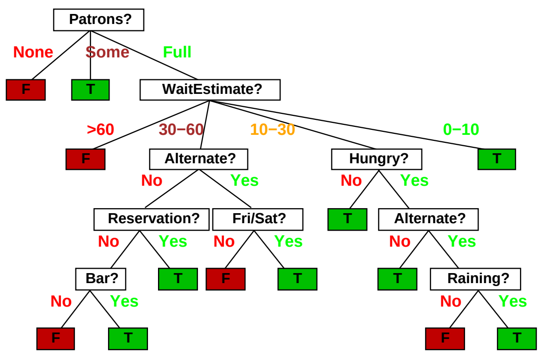
- When will I wait for a table:

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

- Examples described by *attribute values* (Boolean, discrete, continuous, etc.)
- *Classification* of examples is *positive* (T) or *negative* (F)

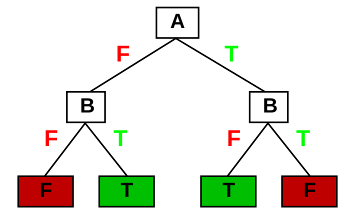
Decision trees

- Here is the “true” tree for deciding whether to wait:



- Decision trees can express any function of the input attributes.
- For Boolean functions, truth table row $\rightarrow$ path to leaf:

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F



(XOR because is hard to capture for some classifiers)

- Trivially, $\exists$ a consistent decision tree for any training set with one path to leaf for each example.
  - unless f nondeterministic in x
- This trivial tree probably won’t generalize to new examples
- Prefer to find more *compact* decision trees

Hypothesis spaces

- How many distinct decision trees with n Boolean attributes?

- How many distinct decision trees with n Boolean attributes?

  = number of Boolean functions

- How many distinct decision trees with n Boolean attributes?

  = number of Boolean functions

  = number of distinct truth tables with 2^n rows

- How many distinct decision trees with n Boolean attributes?

  = number of Boolean functions

  = number of distinct truth tables with 2^n rows = 2^{2^n}

- How many distinct decision trees with n Boolean attributes?

  = number of Boolean functions

  = number of distinct truth tables with 2^n rows = 2^{2^n}

  6 Boolean attributes means 18,446,744,073,709,551,616 trees

- How many distinct decision trees with n Boolean attributes?

  = number of Boolean functions

  = number of distinct truth tables with 2^n rows = 2^{2^n}

  6 Boolean attributes means 18,446,744,073,709,551,616 trees

- How many purely conjunctive hypotheses ($Hungry \wedge \neg Rain$)?

- How many distinct decision trees with n Boolean attributes?

  = number of Boolean functions

  = number of distinct truth tables with 2^n rows = 2^{2^n}

  6 Boolean attributes means 18,446,744,073,709,551,616 trees

- How many purely conjunctive hypotheses ($Hungry \wedge \neg Rain$)?
- Each attribute can be in (positive), in (negative), or out $\Rightarrow 3^n$ distinct conjunctive hypotheses
- More expressive hypothesis space
  - increases chance that target function can be expressed
  - increases number of hypotheses consistent with training set $\Rightarrow$ may get worse predictions

Decision tree learning

- Aim: find a small tree consistent with the training examples.
- Idea: (recursively) choose "most significant" attribute as root of (sub)tree.

Decision tree learning

function DTL(*examples*, *attributes*, *default*) **returns** a decision tree

  if *examples* is empty **then return** *default*

  else if all *examples* have the same classification **then return** the classification

  else if *attributes* is empty **then return** MODE(*examples*)

  else

    best ← CHOOSE-ATTRIBUTE(*attributes*, *examples*)

    tree ← a new decision tree with root test *best*

    for each value v_i of *best* **do**

      $examples_i$ ← {elements of *examples* with *best* $= v_i$}

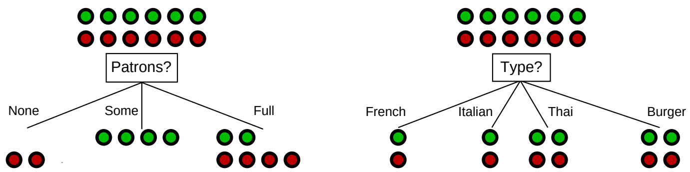
      subtree ← DTL($examples_i$, *attributes* − *best*, MODE(*examples*))

      add a branch to *tree* with label v_i and subtree *subtree*

    return *tree*

Choosing an attribute

- Idea: a good attribute splits the examples into subsets that are (ideally) “all positive” or “all negative”.



- *Patrons*? is a better choice—gives *information* about the classification

Information

- Information answers questions.
- The more clueless I am about the answer initially, the more information is contained in the answer.
- Scale: 1 bit = answer to Boolean question with prior $\langle 0.5, 0.5 \rangle$
- Information in an answer when prior is $\langle P_1, \ldots, P_n \rangle$ is

$$H(\langle P_1, \ldots, P_n \rangle) = \sum_{i=1}^{n} -P_i \log_2 P_i$$

(also called *entropy* of the prior)

- Suppose we have p positive and n negative examples at the root:

$$H(\langle p/(p+n), n/(p+n) \rangle)$$

bits needed to classify a new example.

- For 12 restaurant examples, $p = n = 6$ so we need 1 bit
- An attribute splits the examples E into subsets E_i, each of which (we hope) needs less information to complete the classification

- Let E_i have p_i positive and n_i negative examples.

$$H(\langle p_i/(p_i+n_i), n_i/(p_i+n_i) \rangle)$$

bits needed to classify a new example

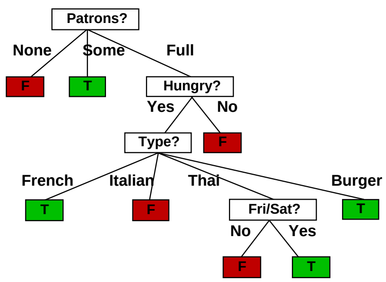
- *Expected* number of bits per example over all branches is

$$\sum_i \frac{p_i + n_i}{p+n} H(\langle p_i/(p_i+n_i), n_i/(p_i+n_i) \rangle)$$

- For *Patrons*?, this is 0.459 bits.
- For *Type* this is (still) 1 bit
- Choose the attribute that minimizes the remaining information needed

Back to the example

- Decision tree learned from the 12 examples:

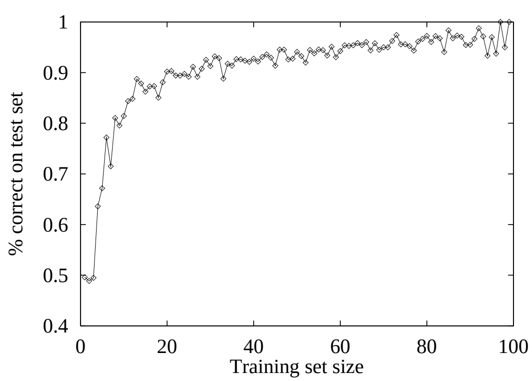


- Substantially simpler than "true" tree—a more complex hypothesis isn't justified by small amount of data

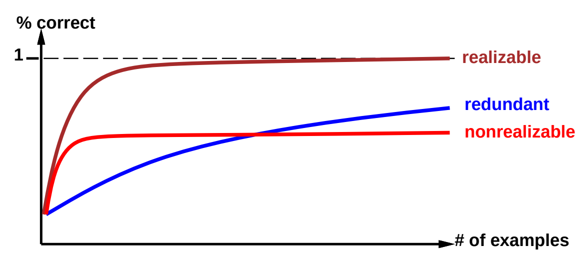
Performance measurement

- How do we know that $h \approx f$?
  1. Use theorems of computational/statistical learning theory
  2. Try h on a new *test set* of examples
     (use *same distribution over example space* as training set)
- *Learning curve* = % correct on test set as a function of training set size

- Learning curve for the restaurant example.



- Learning curve depends on
  – *realizable* (can express target function) vs. *non-realizable*
    non-realizability can be due to missing attributes or restricted hypothesis class
  – redundant expressiveness (e.g., loads of irrelevant attributes)



Validation

- What we just described is *holdout cross-validation*.
  - Disadvantage that it doesn't use all the data.
  - However we split the data we have as training and test sets we can bias the results.
    Not enough training data or bias because the test data is small.
- Better is *k-fold cross validation*.
- Split data into k equal subsets. Learn on all k sets and test each result on the remainder.
- Average test set score is a better estimate of the error rate than a single score.
- Common values of k are 5 and 10, both giving error estimates that are very likely to be accurate.

- The extreme case is when $k = n$, the number of data points.
- *Leave-one-out cross validation*.

Broadening decision tree approach

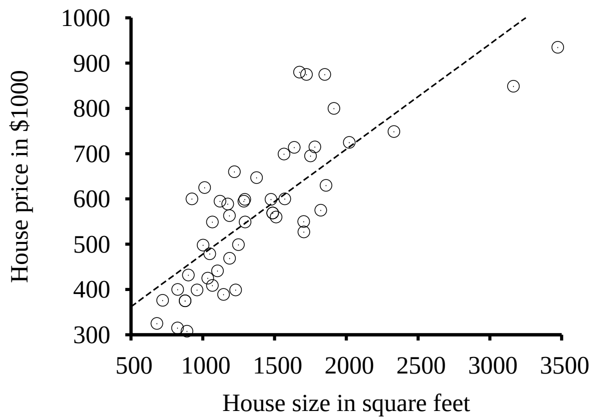
- Multivalued attributes
  - When attributes have many values, information gain gives an inappropriate estimation of the usefulness of the attribute.
    Tend to split examples into small classes (ie. ExactTime)
  - Convert to Boolean tests.
- Continuous/integer input attributes
  - Infinite sets of possible values.
  - Modify approach to identify *split point*s which give highest information gain.
    $Weight > 160$
- Continuous output attributes
  - When trying to predict continuous output values need to create a *regression tree*, which ends with a linear function.

Linear regression

- Learning a linear function of continuous inputs.
- Equation is of the form:

$$h_w(x) = w_1 x + w_0$$

  where the w subscript indicates the vector $[w_0, w_1]$.
- Idea is that we want to estimate the values of w_0 and w_1 from data.
- Textbook gives the example of predicting house prices by floor area.



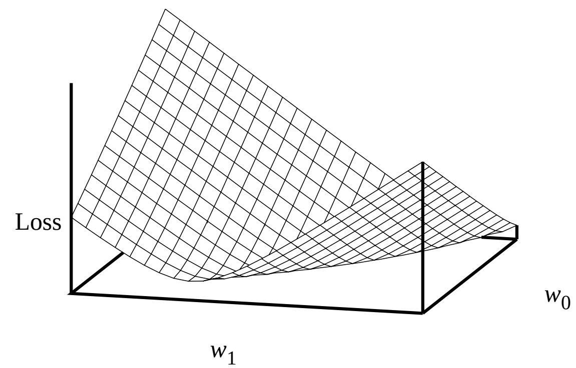
- Finding the h_w that best fits the data is *linear regression*.
- To fit the line we find the $[w_0, w_1]$ that minimize the loss/error.
- Traditionally we use the squared loss function:

$$\begin{aligned} Loss(h_w) &= \sum_{j=1}^{N} L_2(y_j, h_w(x_j)) \\ &= \sum_{j=1}^{N} (y_j - h_w(x_j))^2 \\ &= \sum_{j=1}^{N} (y_j - (w_0 x_j + w_0))^2 \end{aligned}$$

  where the data we have are pairs (x_i, y_i).
- We use the squared loss function because Gauss showed that for normally distributed noise, this gives us the most liklely values of the weights.

- For linear models like this, it is easy enough to solve exactly for w_0 and w_1.
  – See textbook page 719 and any number of statistical packages.
- More interesting is when the model is not linear
  Can use the same kind of ideas.
- What we are doing is trying to minimize the loss.
- Descending the gradient of the loss function.

- For the house price case the loss function looks like:



- More generally, we use a form of hill-climbing.
- Start at any point in the (w_0, w_1) plane and move to a neighboring point that is downhill.
- For each w_i we update with:

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(w)$$

where α is the *learning rate* and controls how fast we move downhill.

- Simple calculus gets us:

$$\begin{aligned} w_0 &\leftarrow w_0 + \alpha(y - h_w(x)) \\ w_1 &\leftarrow w_1 + \alpha(y - h_w(x))x \end{aligned}$$

so if the function is too big, reduce w_0, and adjust w_1 depending on the sign of x.

- This says how to adjust for one example.
- For N examples, we have a choice.
- We can do *batch gradient descent*:

$$\begin{aligned} w_0 &\leftarrow w_0 + \alpha \sum_j (y_j - h_w(x_j)) \\ w_1 &\leftarrow w_1 + \alpha \sum_j (y_j - h_w(x_j))x \end{aligned}$$

which is guaranteed to converge, but can be slow since we need to compute for all N examples at each step.

- We can also adjust separately for each of the N examples at the cost of possibly not converging.

  Quicker though.

  Stochastic gradient descent.

Multivariate linear regression

- Now we have more variables:

$$x_{j,1}, \ldots, x_{j,i}, \ldots x_{j,n}$$

and are interested in a vector of weights w_i.

- Simplify the handling of the weights by creating a dummy attribute to pair with w_0.

  $x_{j,o} = 1$

- Then do gradient descent, as before:

$$w_i \leftarrow w_i + \alpha \sum_j (y_j - h_w(x_j))x_{j,i}$$

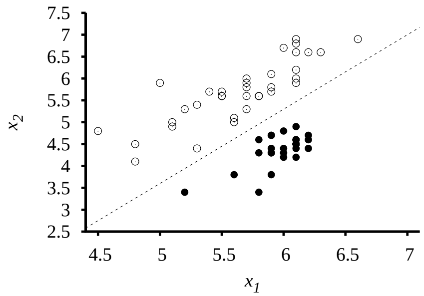
where $h_w(x_j)$ is just the weighted sum of the variable values:

$$h_w(x_j) = \sum_{i=0}^{i=n} w_i x_{j,i}$$

- Not really much harder than the univariate case.
- BUT, have to worry about overfitting.
  - Take the complexity of the model into account in evaluating it.

Linear classifiers

- Can turn a linear function into a classifier:
  - Function defines the boundary between two classes.



- Classify based on where a point lies in relation to the line.

- A linear boundary will separate two *linearly separable* classes.
- In the above example (seismic data due to earthquakes and nuclear explosions)

$$-4.9 + 1.7x_1 - x_2 = 0$$

- Explosions are to the right of the line:

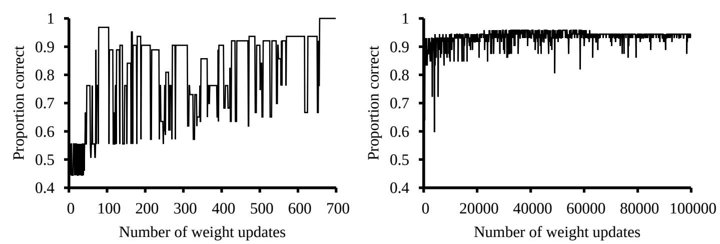
$$-4.9 + 1.7x_1 - x_2 > 0$$

- Thus we classify as follows:

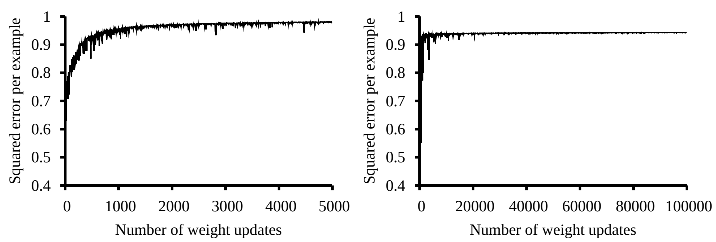
$$h_w(x_j) = 1 \text{ if } \sum_{i=0}^{i=n} w_i x_{j,i} > 0$$

and the classifier returns 0 otherwise.

- Learn the decision boundary just as we learnt the linear function.
- Starting with arbitrary weights
  1. Use the decision rule on a test case.
  2. If it classifies correctly, do not update weights.
  3. Otherwise update weights as above.
- We typically apply one example at a time, i.e. stochastic gradient descent.

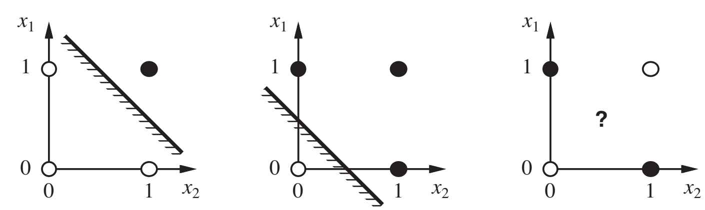


- The curve is not smooth because the boundary is hard, so can misclassify a lot of examples even a long way into learning.



- Using the *logistic function* as a threshold smooths out the errors.
- Logistic function taxes one's calculus, but the update rule is pretty simple.

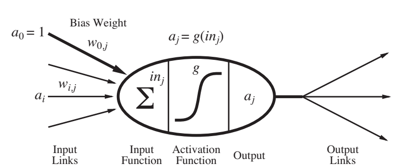
Issues with linear separability



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Neural networks

- We treat a neural network with a single neuron as a simple linear classifier
  - *Perceptron*



- Train it exactly as above.
- Multilayer networks can be trained in a similar fashion, though the derivation of the rules is somewhat nastier.

Nearest neighbor models

- The models we looked at so far are *parametric*
  - We construct them by setting a number of parameters.
  - We effectively search for the right parameter set.
- Work nicely when there is relatively little training data.
- When there is a lot of data, can't the data speak for itself?
  - Rather than filtering it through the small set of parameters.
- *Non-parametric models*.

- Simplest case — could just classify based on all the data we have.
  - If we have the case already, then we know the answer.
  - *Table lookup*
- Clearly this has holes.
- Better is to use *nearest neighbor* approaches.
  - Find the N nearest points.
  - Let the neighbors vote on the classification.
- Can also do regression on the set of neighbors.

- To find "nearest" points we need a notion of distance.
- Common to use the *Minkowski distance*:

$$L_p(x_j, x_q) = \left(\sum_i (|x_{j,i} - x_{q,i}|^p\right)^{1/p}$$

- This is a generalization of Euclidian distance ($p = 2$) to a multidimensional space.
- Have to worry about the differences in scale between dimensions, and correlations between dimensions

  (don't need to use them all).

- Clearly we can find the N nearest neighbors with a single pass through the data.

  $O(N)$
- For large N this may be sub-optimal, so use trees or hash tables to speed the search.
- Naturally you need to build the structures with locality in mind.

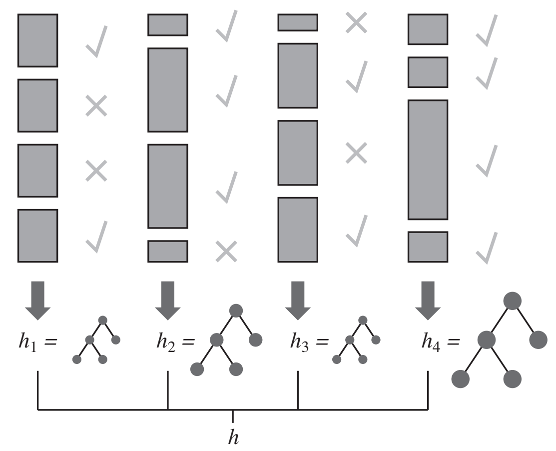
Ensemble learning

- Every classifier has an error rate
  - Will always misclassify some examples.
- Using an *ensemble* is an easy way to improve on this.
- Take N classifiers, use them all on the same example.
- Have them vote on the classification.
- For a binary classification and 5 classifiers, error rate drops from 10% (say) to less than 1%.

  Assuming that the classifiers are independent

  (i.e. different enough).

- *Boosting* extends this idea.
- Builds on the idea of a *weighted training set*
  - Higher weighted examples are counted as more important during training.
    (For example we put more copies into the training set)
- Boosting starts with all examples of equal weight, and learns a classifier h_1.
- Test it.
- Increase the weights of the misclassified examples and learn a new classifier h_2.
- Repeat.
- Final ensemble is the majority combination of all the classifers, weighted by how well they perform on the training set.



- The AdaBoost algorithm is a commonly used approach to boosting.
- Given an initial classifier that is slightly better than random, AdaBoost can generate an ensemble that will perfectly classify the training set.

Summary

- Learning needed for unknown environments, lazy designers
- Learning agent = performance element + learning element
- Learning method depends on type of performance element, available feedback, type of component to be improved, and its representation
- For supervised learning, the aim is to find a simple hypothesis approximately consistent with training examples
- Looked at a number of approaches to this kind of learning.