COMP 6321
Machine Learning

Instructor: Adam Krzyżak
Email: krzyzak@cs.concordia.ca

Lecture 4: More on neural nets. More in instance-based learning. Decision Trees

- Automatic methods for constructing the network structure
- Other types of networks:
  - Autoencoders
  - Recurrent neural networks
- Instance-based (nearest neighbor) learning
- Brief interlude on information theory
- Decision tree construction
- Overfitting avoidance

Destructive methods

- Simple solution: consider removing each weight in turn (by setting it to 0), and examine the effect on the error
- Weight decay: give each weight a chance to go to 0, unless it is needed to decrease error:

$$\Delta w_j = -\alpha_j \frac{\partial J}{\partial w_j} - \lambda w_j$$

  where λ is a decay rate
- Optimal brain damage:
  - Train the network to a local optimum
  - Approximate the saliency of each link or unit (i.e., its impact on the performance of the network), using the Hessian matrix
  - Greedily prune the element with the lowest saliency

  This loop is repeated until the error starts to deteriorate

Constructive methods

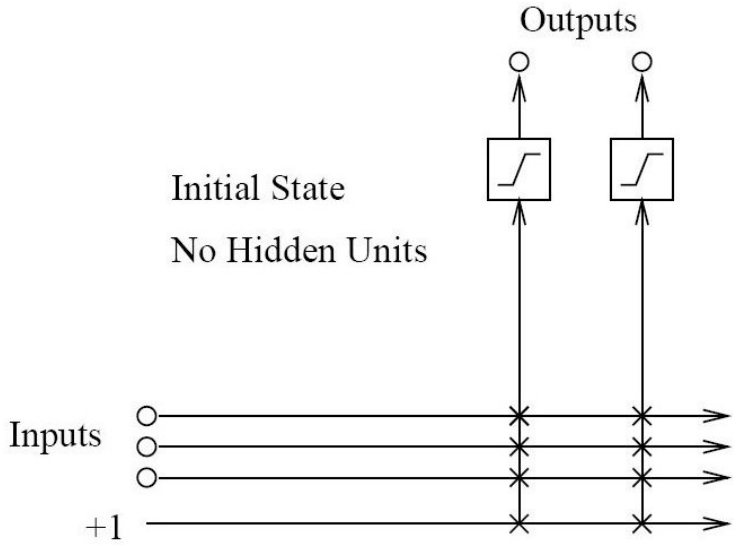
- Dynamic node creation (Ash):
  - Start with just one hidden unit, train using backprop
  - If the error is still high, add another unit in the same layer and repeat

  Only one layer is constructed
- Meiosis networks (Hanson):
  - Start with just one hidden unit, train using backprop
  - Compute the variance of each weight during training
  - If a unit has one or more weights of high variance, it is split into two units, and the weights are perturbed

  The intuition is that the new units will specialize to different functions.
- Cascade correlation: Add units to correlate with the residual error

Cascade correlation (Fahlman & Lebiere)

- Start with no hidden units, just the inputs connected to the outputs



- Train using backpropagation until the error is stable
- If the error is small enough, stop, otherwise consider adding a new hidden unit

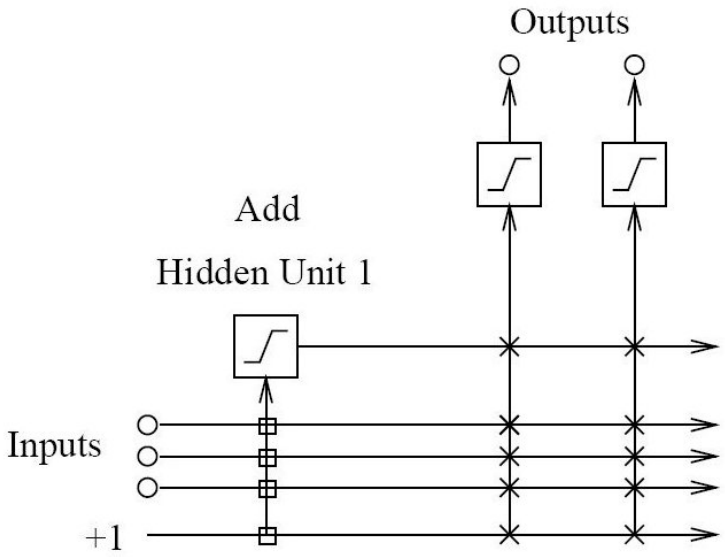
Cascade correlation (II)

- Consider a pool of candidate hidden units, each linked with the inputs, but not yet connected to the output
- Initialize each candidate randomly and train it to maximize the *correlation to the error of the previous network* (using gradient descent):

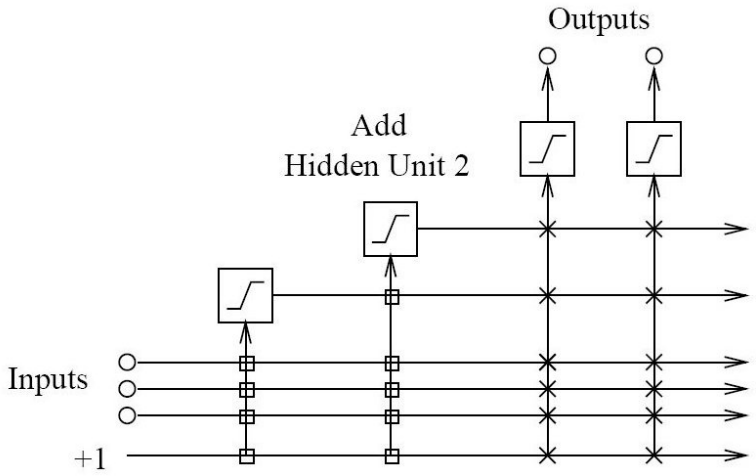
$$\sum_i (J_i - \bar{J_i})(o_i - \bar{o})$$

  where o_i is the output of the neuron for pattern i and J_i is the error on pattern i, $\bar{o}$ and $\bar{J}$ are averages over all patterns.
- Pick the best candidate and insert it in the network, by connecting it to the output unit
- Freeze the weights coming into the unit and *train only the weights going into the output neuron*

Network after adding one hidden unit



Network after adding two hidden units



Cascade correlation (III)

- The algorithm alternates between two phases:
  1. Input phase: Training candidate units
  2. Output phase: Training the weights that connect the units to the output

  until the error is below a desired threshold
- Because only a small set of weights is trained at one time, training is very fast despite the depth of the network
- Variations allow for the new neurons to go either in the same layer or in a new layer.

More application-specific tricks

- If there is too little data, it can be perturbed by random noise; this helps escape local minima and gives more robust results
- In image classification and pattern recognition tasks, extra data can be generated, e.g., by applying transformations that make sense
- *Weight sharing* can be used to indicate parameters that should have the same value based on prior knowledge
- In this case, each update is computed separately using backprop, then the tied parameters are updated with an average

Other kinds of networks: Auto-encoders

- When doing regression or classification, typically each successive hidden layer has fewer units than the previous one
- Hence, the hidden layer can be viewed as performing *dimensionality reduction* on the data
- The task is performed in a way targeted towards prediction of a class label or real-valued output
- If we just want to perform dimensionality reduction on the data, we can use *autoencoders*
- An autoencoder has identical input and output, and typically a smaller hidden layer
- If learning is successful, the hidden layer will contain enough information to re-construct the output.
- If linear activations or only a single sigmoid hidden layer are used, then the optimal solution to an auto-encoder is strongly related to principal component analysis (PCA)

Example: Learning an autoencoder function

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned? How many hidden units should we use?

Learned hidden layer representations

Input		Hidden Layer				Output
10000000	$\rightarrow$	.89	.04	.08	$\rightarrow$	10000000
01000000	$\rightarrow$	.15	.99	.99	$\rightarrow$	01000000
00100000	$\rightarrow$	.01	.97	.27	$\rightarrow$	00100000
00010000	$\rightarrow$	.99	.97	.71	$\rightarrow$	00010000
00001000	$\rightarrow$	.03	.05	.02	$\rightarrow$	00001000
00000100	$\rightarrow$	.01	.11	.88	$\rightarrow$	00000100
00000010	$\rightarrow$	.80	.01	.98	$\rightarrow$	00000010
00000001	$\rightarrow$	.60	.94	.01	$\rightarrow$	00000001

Applications of autoencoder

- Hinton, Salakhutdinov, Science (2006)

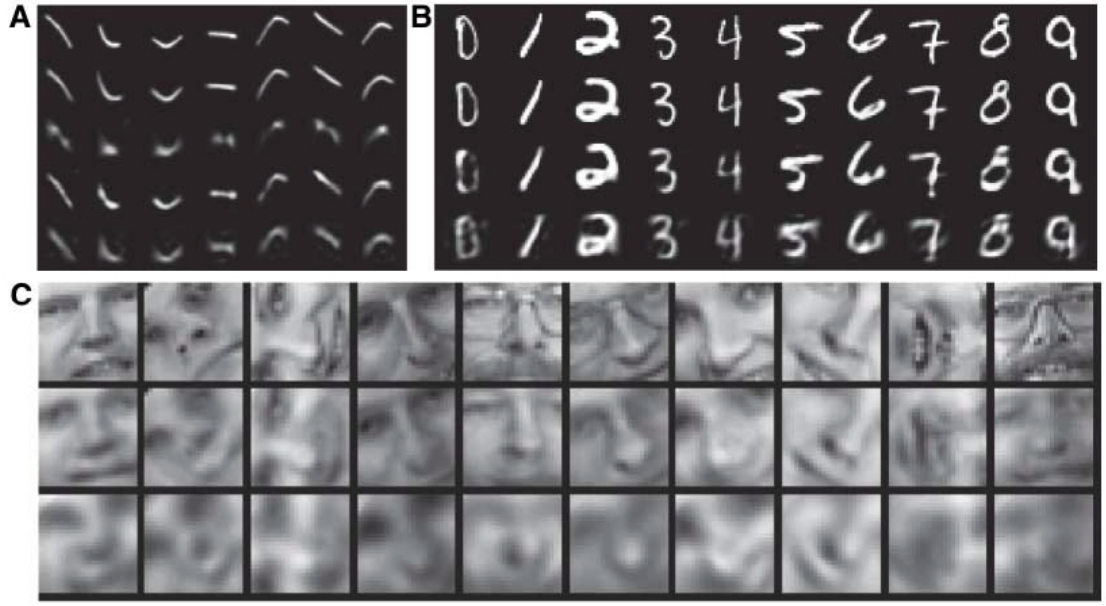


Fig. 2. (**A**) Top to bottom: Random samples of curves from the test data set; reconstructions produced by the six-dimensional deep autoencoder; reconstructions by "logistic PCA" (*8*) using six components; reconstructions by logistic PCA and standard PCA using 18 components. The average squared error per image for the last four rows is 1.44, 7.64, 2.45, 5.90. (**B**) Top to bottom: A random test image from each class; reconstructions by the 30-dimensional autoencoder; reconstructions by 30-dimensional logistic PCA and standard PCA. The average squared errors for the last three rows are 3.00, 8.01, and 13.87. (**C**) Top to bottom: Random samples from the test data set; reconstructions by the 30-dimensional autoencoder; reconstructions by 30-dimensional PCA. The average squared errors are 126 and 135.

Neural networks for temporal data

- In temporal data, we have a succession of inputs at different times $\mathbf{x}(t)$
- There can be several tasks:
  - Predicting a class label
    E.g., Speech recognition
  - Predicting the next data point, i.e., time-series prediction
    E.g., Financial prediction
- Note that the number of inputs is no longer fixed and constant!
- Prediction may need to be performed as the data is received

Time-delay neural networks (Waibel)

- Fix a time window T
- Collect the inputs $\mathbf{x}(t), \ldots \mathbf{x}(t-T)$ and feed them together to the network
- Shift the window and repeat
- Train using standard backpropagation

Recurrent neural networks

- Connections are allowed to go to units in the same layer (including self-connections) or previous layers
- These recurrent connections are a form of *short-term memory*
- If the maximum length of the desired sequences is small, the network can be trained using *backpropagation through time* (Rumelhart, Hinton and Williams)
  - Create a copy for each unit and connection at different points in time
  - As in weight sharing, copies of the same unit are forced to be identical, by training each of the weights using the average weight update
- For more: tutorial by Pearlmutter

Example applications

- Speech phoneme recognition [Waibel] and synthesis [Nettalk]
- Image classification [Kanade, Baluja, Rowley]
- Digit recognition [Bengio, Bouttou, LeCun et al - LeNet]
- Financial prediction
- Learning control [Pomerleau et al]

When to consider using neural networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued, or a vector of values
- Possibly noisy data
- Training time is not important
- Form of target function is unknown
- Human readability of result is not important
- The computation of the output based on the input has to be fast

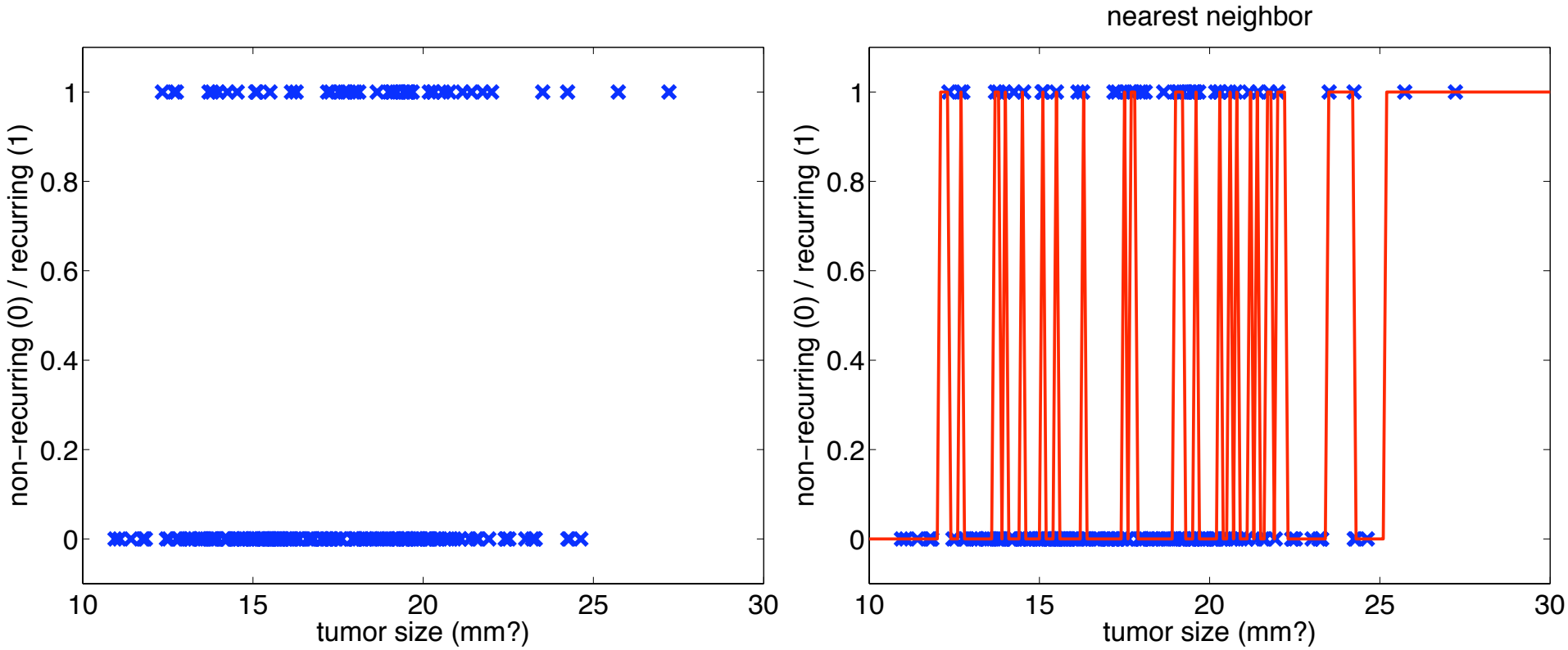
Parametric supervised learning

- So far, we have assumed that we have a data set D of labeled examples
- From this, we learn a *parameter vector* of a *fixed size* such that some error measure based on the training data is minimized
- These methods are called *parametric*, and their main goal is to summarize the data using the parameters
- Parametric methods are typically global, i.e. have one set of parameters for the entire data space
- But what if we just remembered the data?
- When new instances arrive, we will compare them with what we know, and determine the answer

Non-parametric (memory-based) learning methods

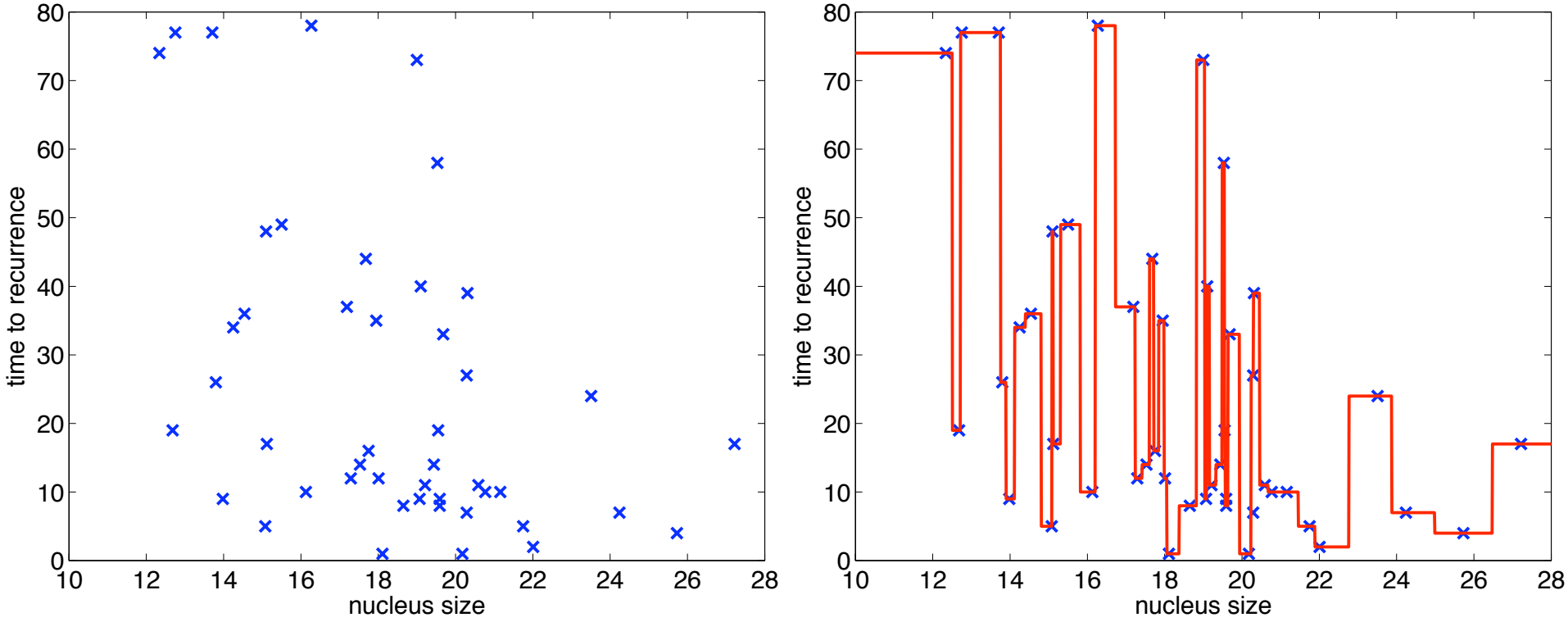
- Key idea: just store all training examples $\langle \mathbf{x_i}, y_i \rangle$
- When a query is made, compute the value of the new instance based on the values of the *closest (most similar) points*
- Requirements:
  - A distance function
  - How many closest points (neighbors) to look at?
  - How do we compute the value of the new point based on the existing values?

Simple idea: Connect the dots!



Wisconsin data set, classification

Simple idea: Connect the dots!



Wisconsin data set, regression

One-nearest neighbor

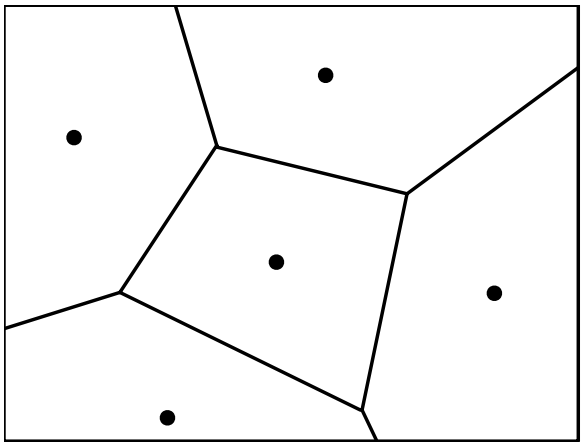
- Given: Training data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$, distance metric d on $\mathcal{X}$.
- Learning: Nothing to do! (just store data)
- Prediction: for $\mathbf{x} \in \mathcal{X}$
  - Find nearest training sample to $\mathbf{x}$.

$$i \in \arg\min_i d(\mathbf{x}_i, \mathbf{x})$$

  - Predict $\mathbf{y} = \mathbf{y}_i$.

What does the approximator look like?

- Nearest-neighbor does not explicitly compute decision boundaries
- But the effective decision boundaries are a subset of the *Voronoi diagram* for the training data



Each line segment is equidistant between two points of opposite classes.

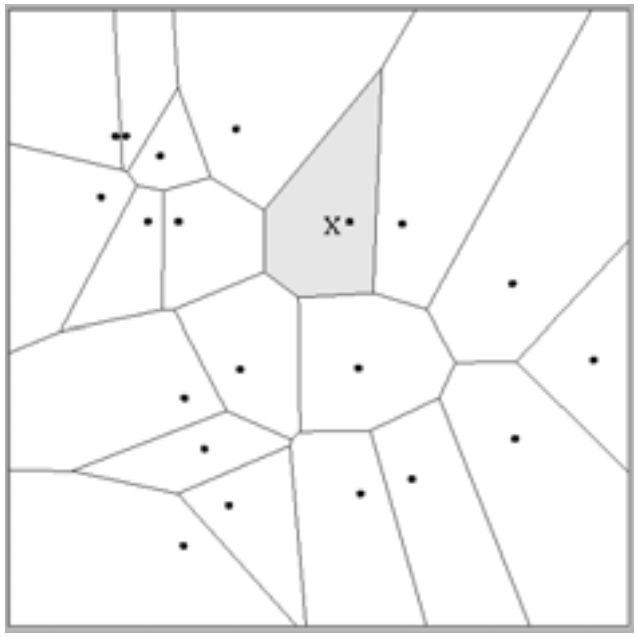
What kind of distance metric?

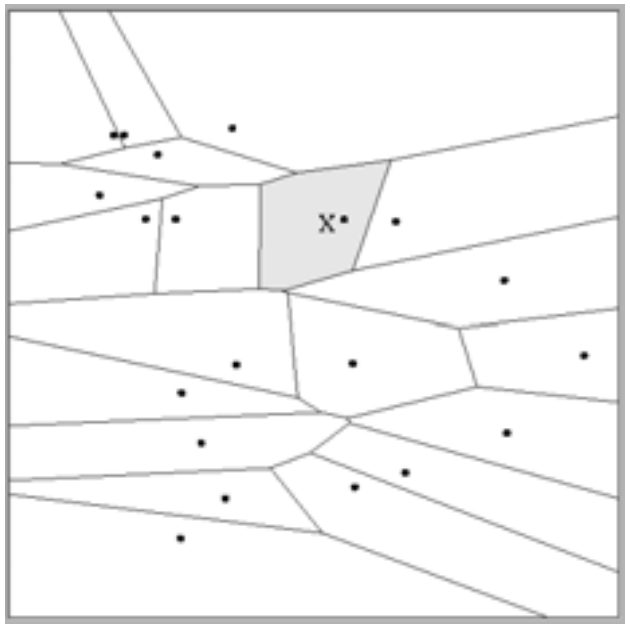
- Euclidian distance
- Maximum/minimum difference along any axis
- Weighted Euclidian distance (with weights based on domain knowledge)

$$\sum_i w_i (x_{q,i} - x_{t,i})^2$$

- An arbitrary distance or similarity function d, specific for the application at hand (works best, if you have one)

Distance metric is really important!





Left: both attributes weighted equally; Right: second attributes weighted more

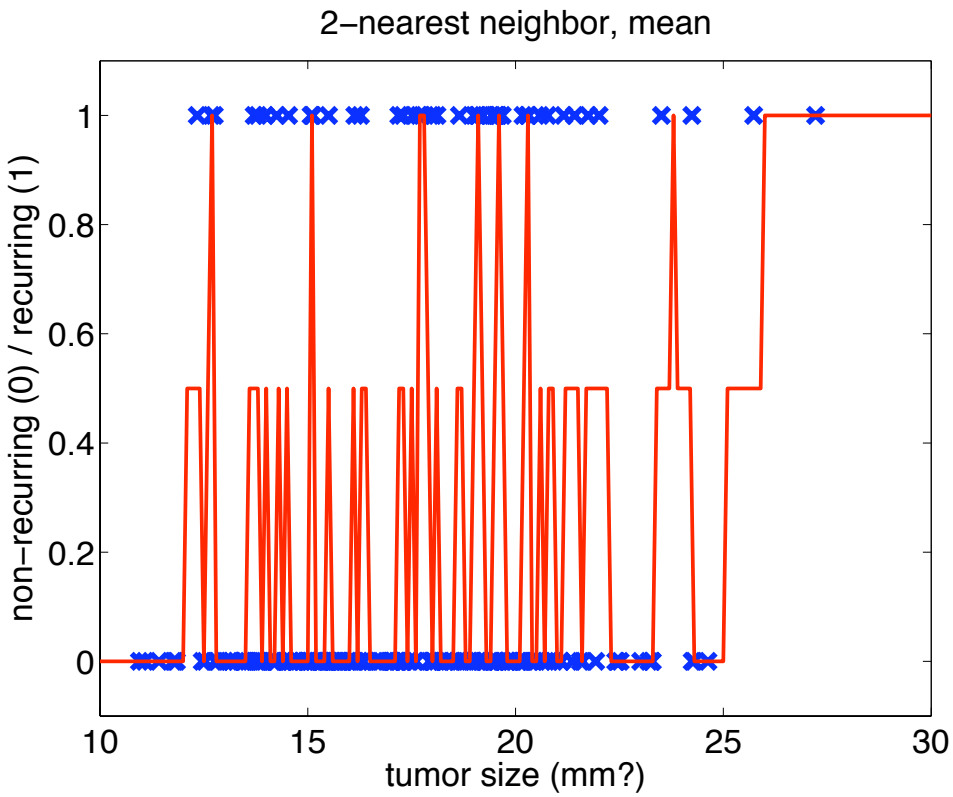
Distance metric tricks

- You may need to do preprocessing:
  - *Scale* the input dimensions (or normalize them)
  - Remove noisy inputs
  - Determine weights for attributes based on cross-validation (or information-theoretic methods)
- Distance metric is often domain-specific
  - E.g. string edit distance in bioinformatics
  - E.g. trajectory distance in time series models for walking data
- Distance metric can be learned sometimes (more on this later)

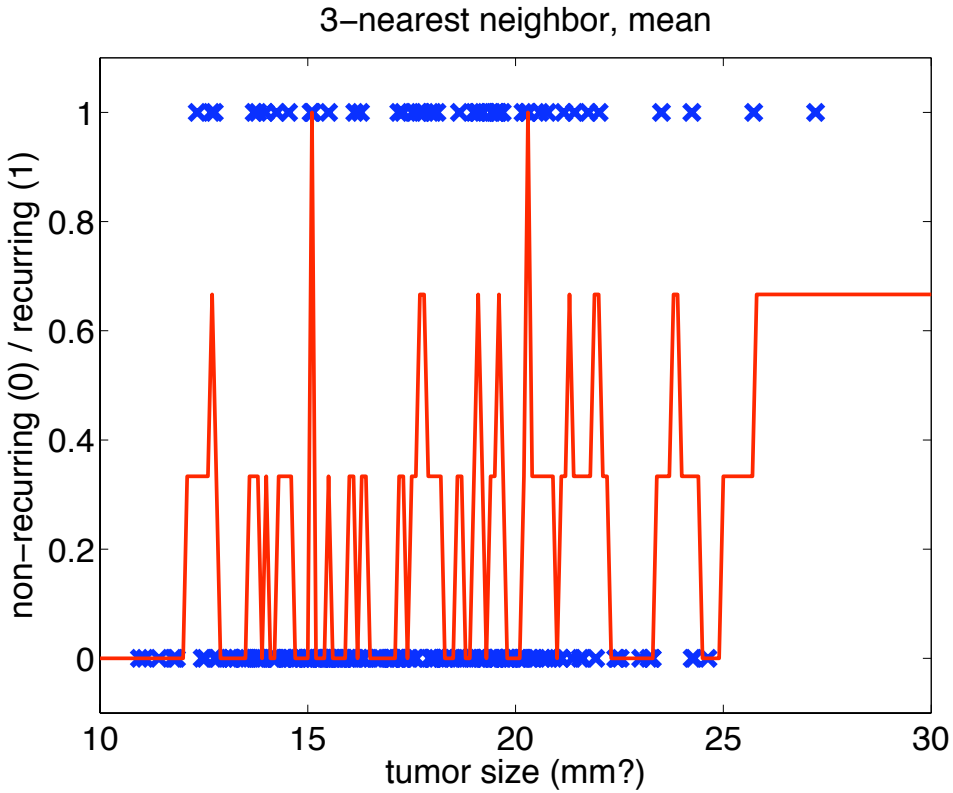
k-nearest neighbor

- Given: Training data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$, distance metric d on $\mathcal{X}$.
- Learning: Nothing to do!
- Prediction: for $\mathbf{x} \in \mathcal{X}$
  - Find the k nearest training samples to $\mathbf{x}$.
    Let their indeces be $i_1, i_2, \ldots, i_k$.
  - Predict
    - $\mathbf{y} =$ mean/median of $\{\mathbf{y}_{i_1}, \mathbf{y}_{i_2}, \ldots, \mathbf{y}_{i_k}\}$ for regression
    - $\mathbf{y} =$ majority of $\{\mathbf{y}_{i_1}, \mathbf{y}_{i_2}, \ldots, \mathbf{y}_{i_k}\}$ for classification, or empirical probability of each class

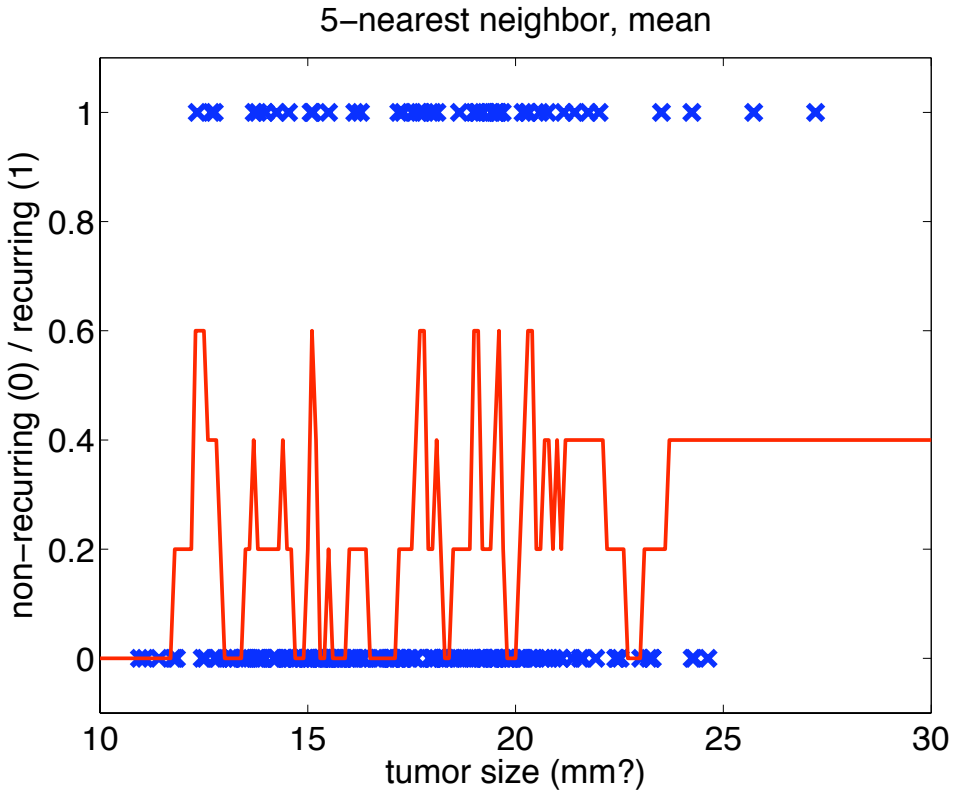
Classification, 2-nearest neighbor, empirical distribution



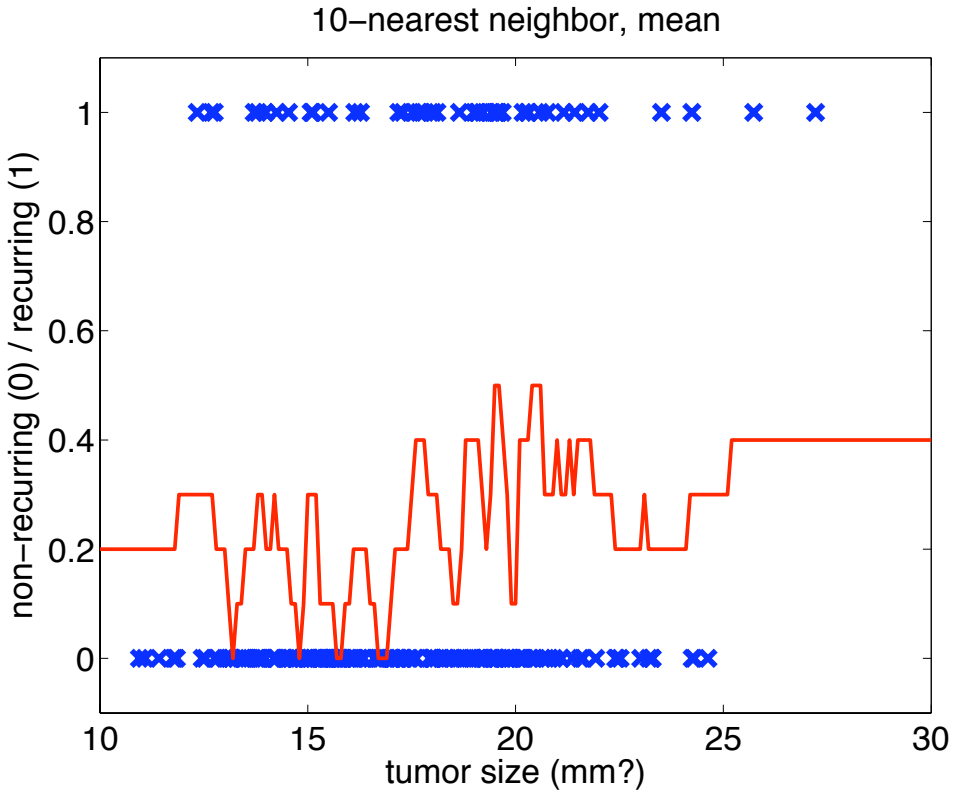
Classification, 3-nearest neighbor



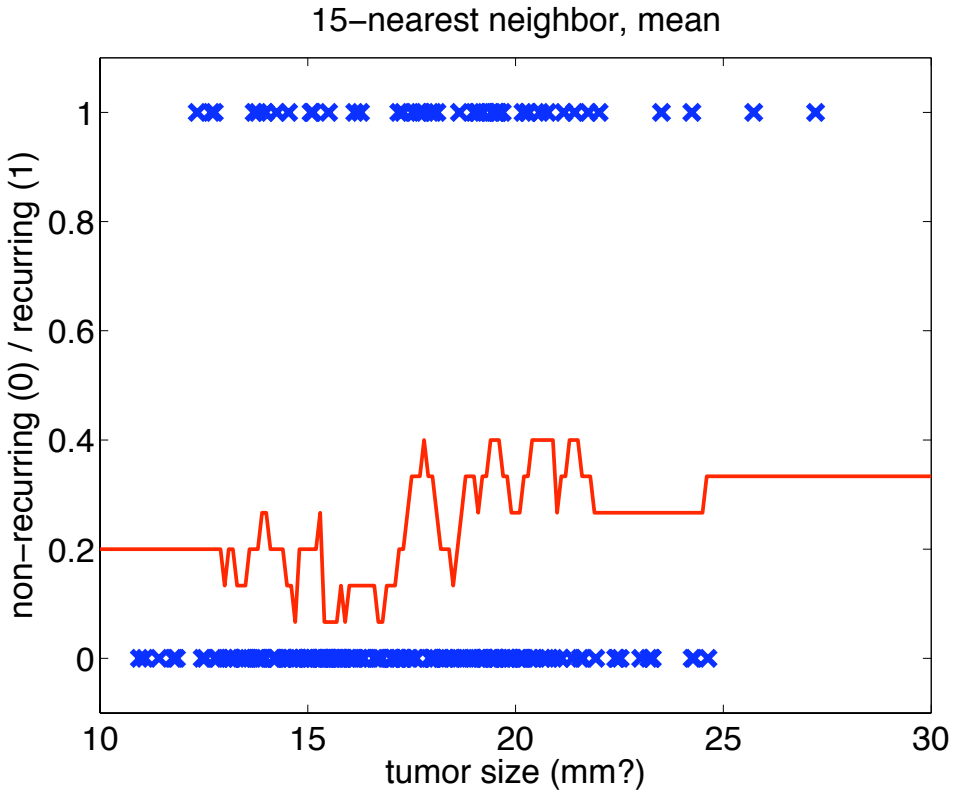
Classification, 5-nearest neighbor



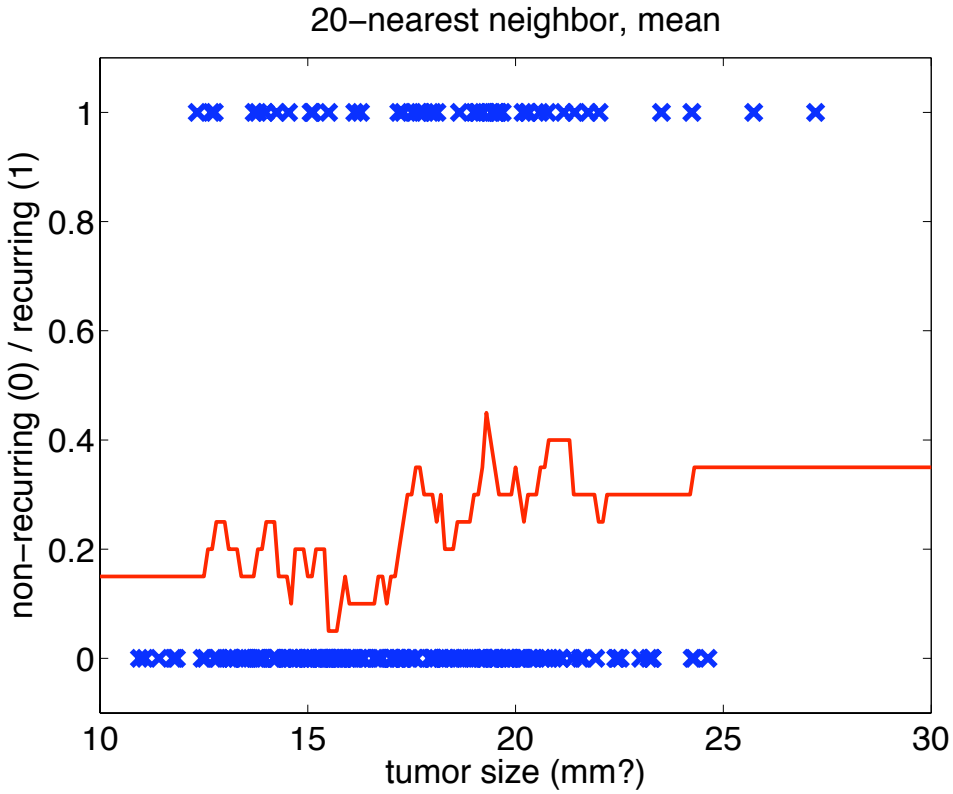
Classification, 10-nearest neighbor



Classification, 15-nearest neighbor



Classification, 20-nearest neighbor



Bias-variance trade-off

- Bias of k-NN is approximately

$$\frac{1}{24f^3(x)}[(m''f + 2m'f')(x)](\frac{k}{n})^2$$

  where m is regression curve and f is density of the input
- Variance of k-NN is approximately

$$\frac{\sigma^2(x)}{k}$$

  (Härdle, Applied Nonparametric Regression)

Bias-variance trade-off

- If k is low, very non-linear functions can be approximated, but we also capture the noise in the data
  Bias is low, variance is high
- If k is high, the output is much smoother, less sensitive to data variation
  High bias, low variance
- A validation set can be used to pick the best k

- Bias-variance trade-off
- Methods for improving efficiency of nearest-neighbor
- What are decision trees?

Recall: Parametric and non-parametric learning

- Parametric methods summarize the existing data set into a set of parameters

  E.g. Logistic and linear regression, feed-forward neural nets

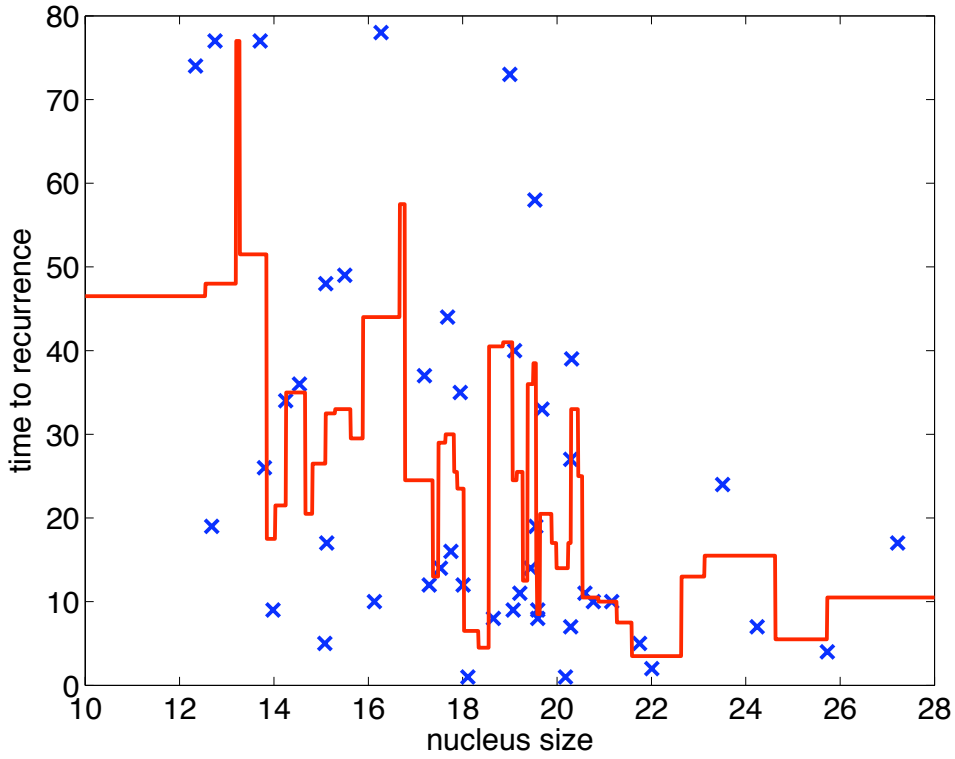
- Nonparametric methods “memorize” the data, then use a distance metric to measure the similarity of new query points to existing examples

  E.g. Nearest neighbor

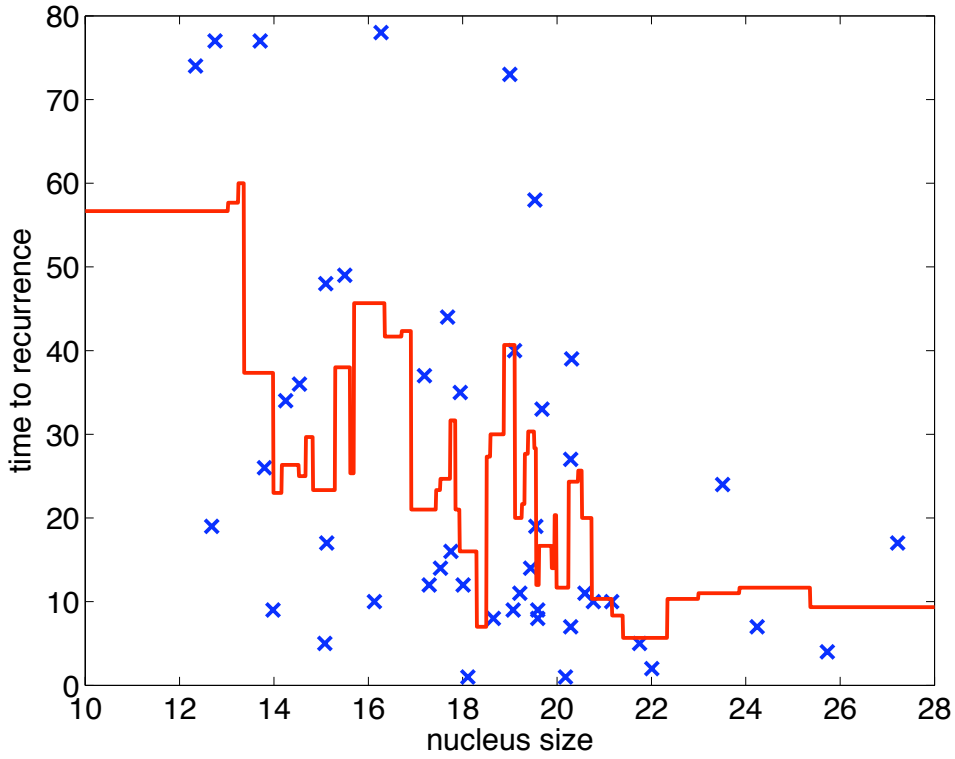
Recall: k-nearest neighbor

- Given: Training data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$, distance metric d on $\mathcal{X}$.
- Learning: Nothing to do!
- Prediction: for $\mathbf{x} \in \mathcal{X}$
  - Find the k nearest training samples to $\mathbf{x}$.
    Let their indeces be $i_1, i_2, \ldots, i_k$.
  - Predict
    - $\mathbf{y} =$ mean/median of $\{\mathbf{y}_{i_1}, \mathbf{y}_{i_2}, \ldots, \mathbf{y}_{i_k}\}$ for regression
    - $\mathbf{y} =$ majority of $\{\mathbf{y}_{i_1}, \mathbf{y}_{i_2}, \ldots, \mathbf{y}_{i_k}\}$ for classification, or empirical probability of each class

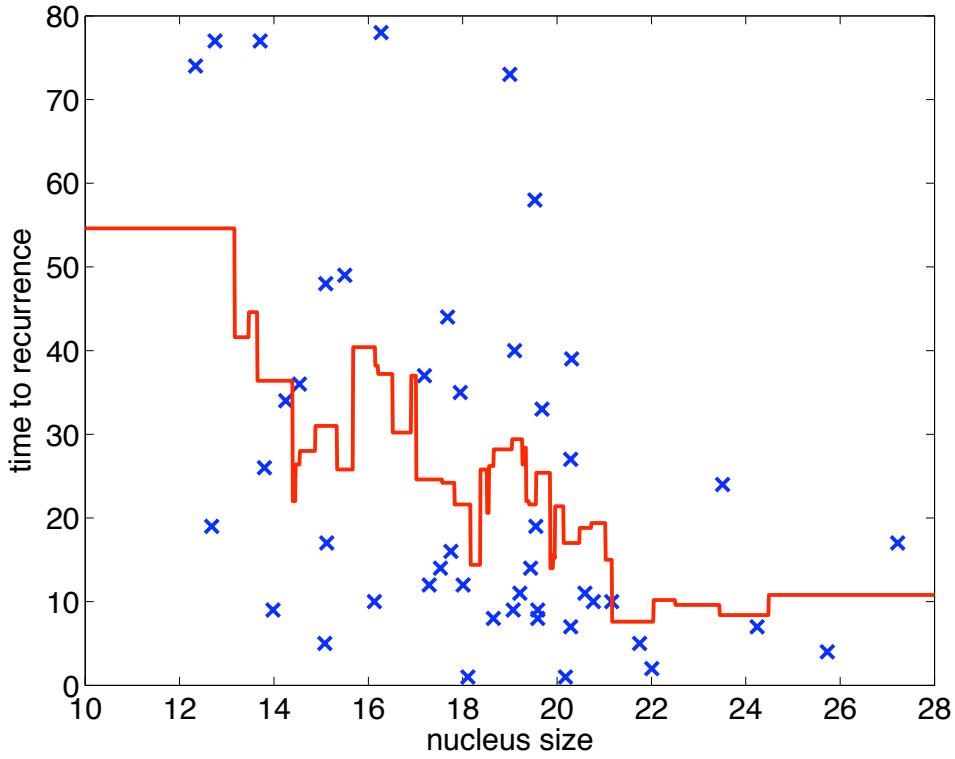
Regression, 2-nearest neighbor, mean prediction



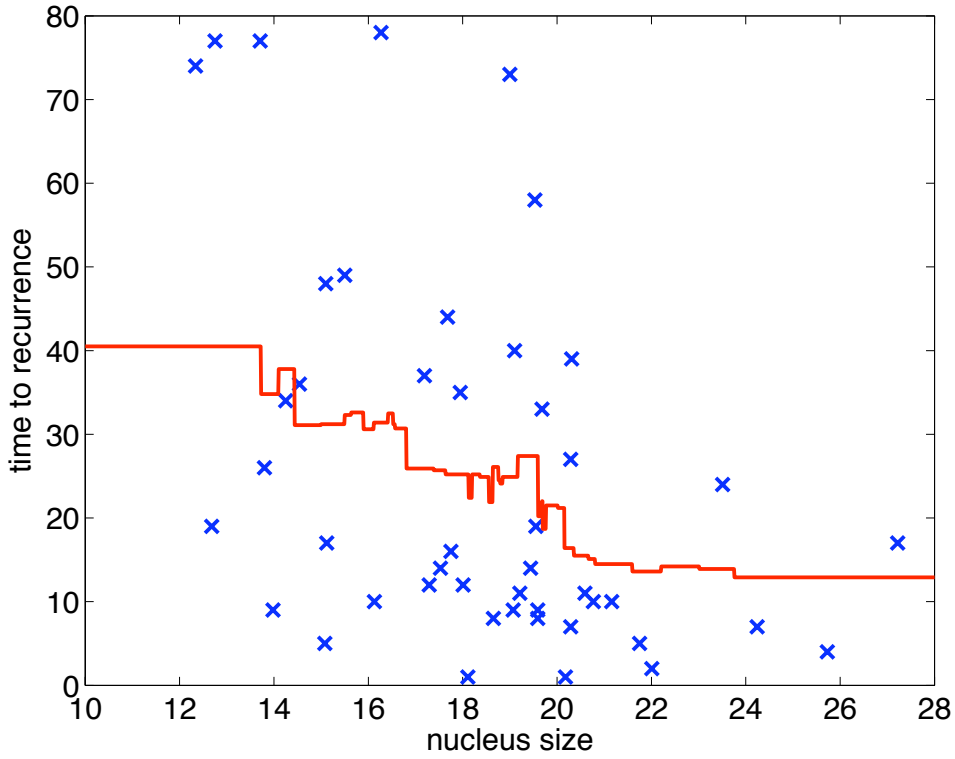
Regression, 3-nearest neighbor



Regression, 5-nearest neighbor



Regression, 10-nearest neighbor



Recall: Bias-variance trade-off

- If k is low, very non-linear functions can be approximated, but we also capture the noise in the data
  Bias is low, variance is high
- If k is high, the output is much smoother, less sensitive to data variation
  High bias, low variance
- A validation set can be used to pick the best k

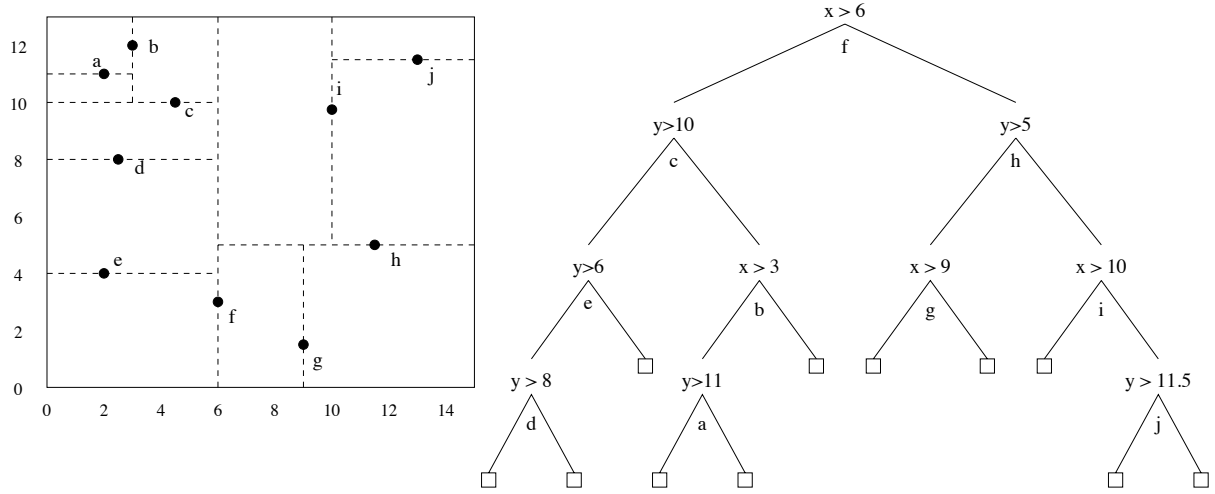
Improving query efficiency

- If the data set is large, searching through all points to compute the set of nearest neighbors is very slow
- Possible solutions:
  - Condensation of the data set
  - Hash tables in which the hashing function is based on the distance metric
  - kd-trees

Condensation: Main idea

- Only the points that support the decision boundary are needed to compute the classification
- Unfortunately, finding the minimal set of such points is NP complete.
- Heuristic; go through the data set, if a point is classified correctly do nothing, otherwise add it to the "condensed" set
- More generally: dictionary methods try to determine a subset of data that is worth keeping

kd-trees



- Split the examples using the median value of the feature with the highest variance
- Points corresponding to the splitting value are stored in the internal nodes
- We can control the depth of the tree (stop splitting)
- In this case, we will have a pool of points at the leaves, and we still need to go through all of them

kd-tree search

- Go down the appropriate branches until we find a match - this gives a candidate best distance
- At every node along the path, check if a better distance could have been obtained on a different branch
  Compute intersection of a hypersphere of the candidate distance, centered at the point, with the hyperplane at that node.
- If a better solution is possible, recurse down other branches

Features of kd-trees

- Makes it easy to do 1-nearest neighbor
- To compute weighted nearest-neighbor efficiently, we can leave out some neighbors, if their influence on the prediction will be small
- But the tree needs to be restructured periodically if we acquire more data, to keep it balanced

Problems with k-nearest neighbor

- A lot of discontinuities!
- Sensitive to small variations in the input data
- Can we fix this but still keep it (fairly) local?

Distance-weighted (kernel-based) nearest neighbor

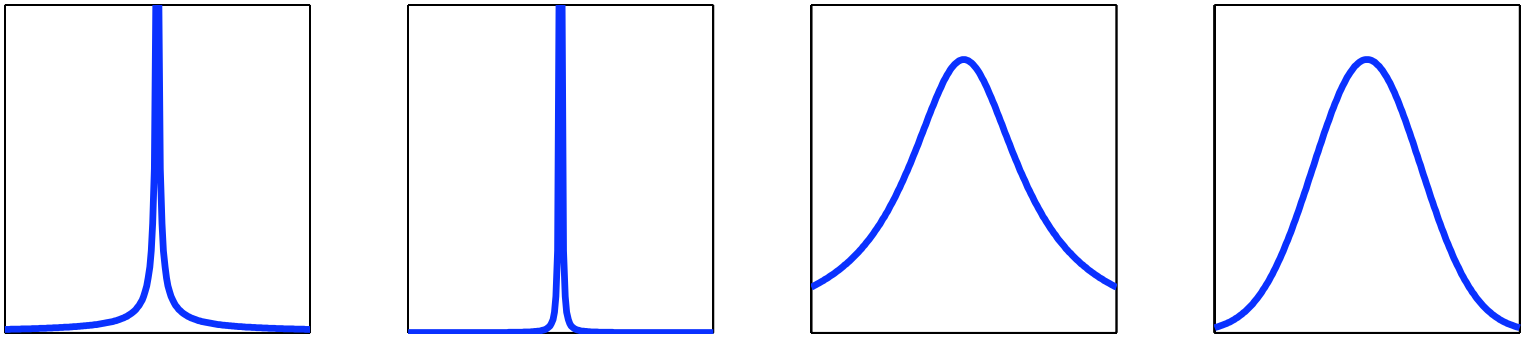
- Inputs: Training data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$, distance metric d on $\mathcal{X}$, weighting function $w : \Re \mapsto \Re$.
- Learning: Nothing to do!
- Prediction: On input $\mathbf{x}$,
  - For each i compute $w_i = w(d(\mathbf{x}_i, \mathbf{x}))$.
  - Predict weighted majority or mean. For example,

$$\mathbf{y} = \frac{\sum_i w_i \mathbf{y}_i}{\sum_i w_i}$$

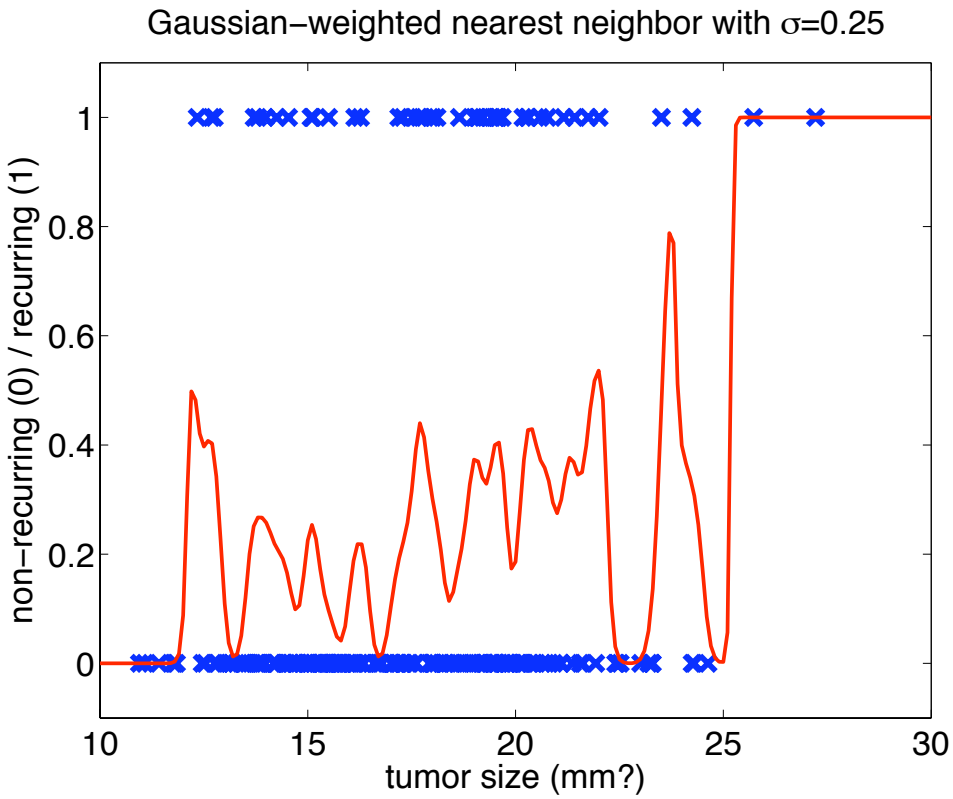
- How to weight distances?

Some weighting functions

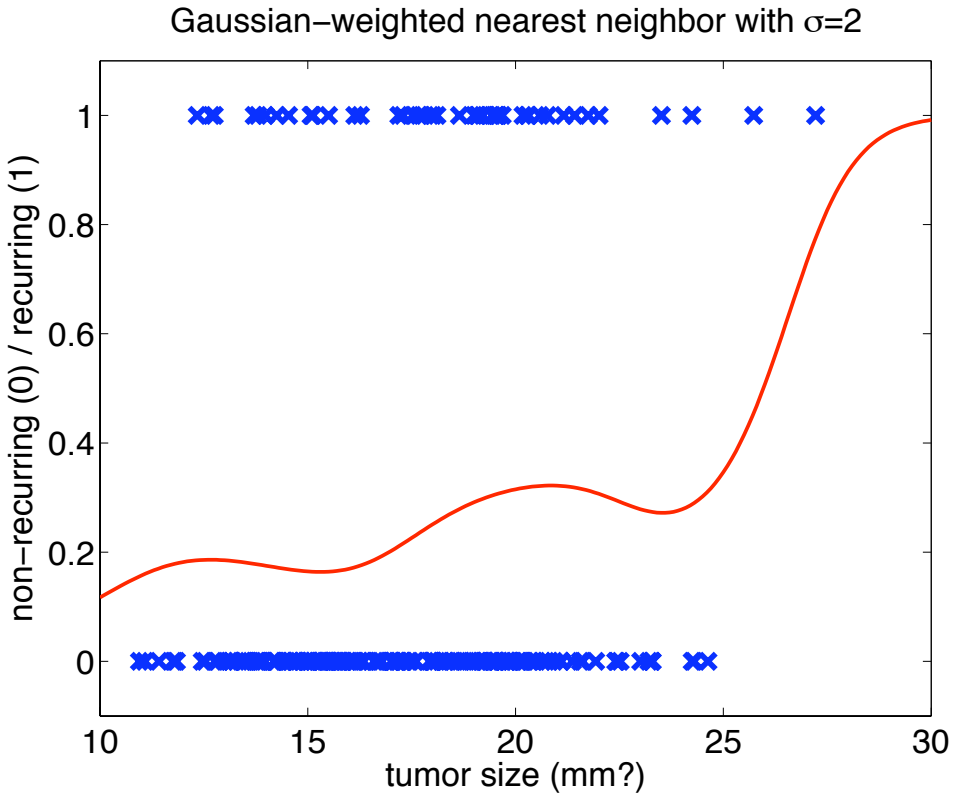
$$\frac{1}{d(\mathbf{x}_i, \mathbf{x})} \qquad \frac{1}{d(\mathbf{x}_i, \mathbf{x})^2} \qquad \frac{1}{c + d(\mathbf{x}_i, \mathbf{x})^2} \qquad e^{-\frac{d(\mathbf{x}_i, \mathbf{x})^2}{\sigma^2}}$$



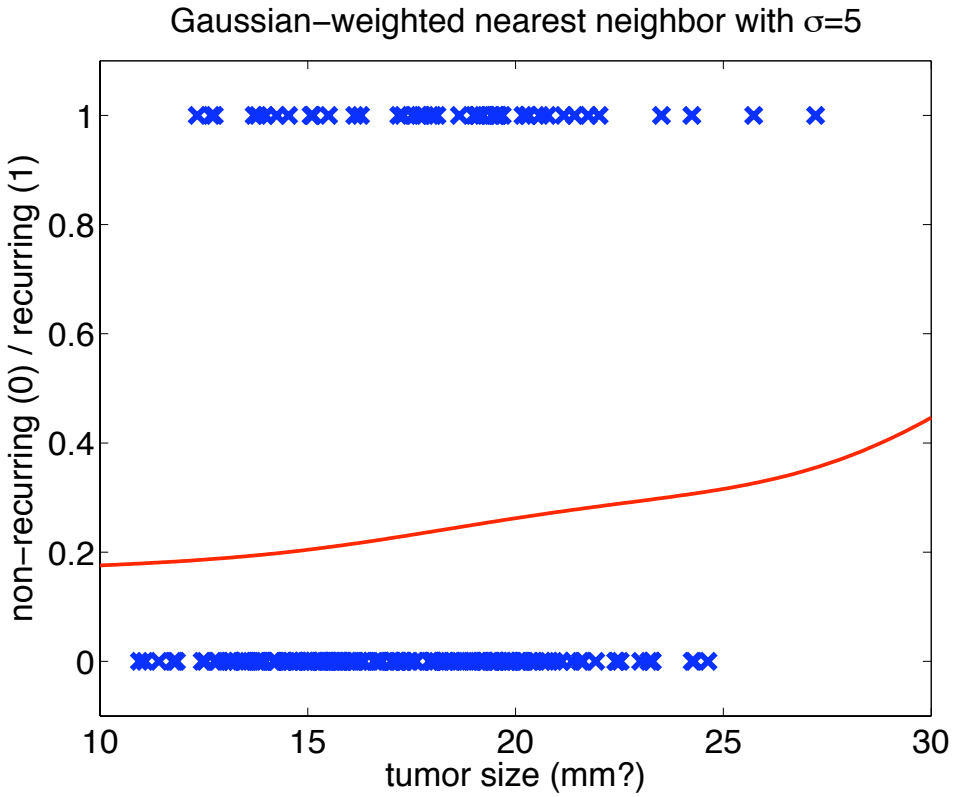
Example: Gaussian weighting, small σ



Gaussian weighting, medium σ



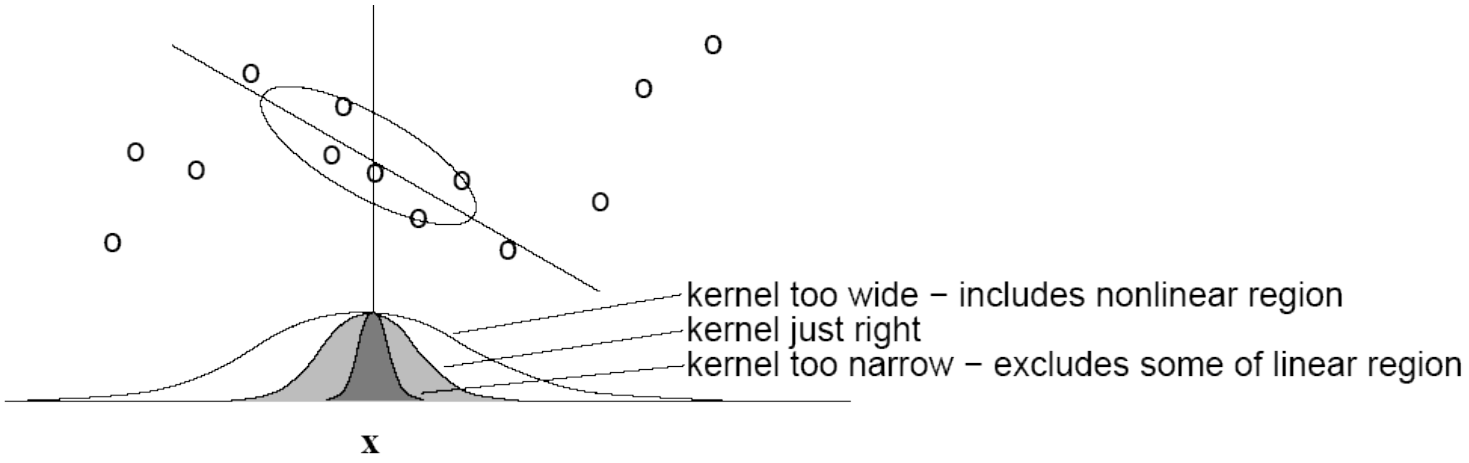
Gaussian weighting, large σ



All examples get to vote! Curve is smoother, but perhaps too smooth.

Locally-weighted linear regression

- Weighted linear regression: different weights in the error function for different points (see homework 1)
- Locally weighted linear regression: weights *depend on the distance* to the query point
- Compared to kernel-based regression: use a linear fit rather than just an average



Lazy and eager learning

- *Lazy*: wait for query before generalizing

  E.g. Nearest Neighbor
- *Eager*: generalize before seeing query

  E.g. Backpropagation, Linear regression,

Does it matter?

Pros and cons of lazy and eager learning

- Eager learner must create global approximation
- Lazy learner can create many local approximations
- If they use same hypothesis space H, a lazy learner can represent more complex functions (e.g., consider H = linear functions)
- Eager learner does the work off-line, summarizes lots of data with few parameters
- Lazy learner has to do lots of work sifting through the data at query time
- Typically lazy learners take longer time to answer queries and require more space

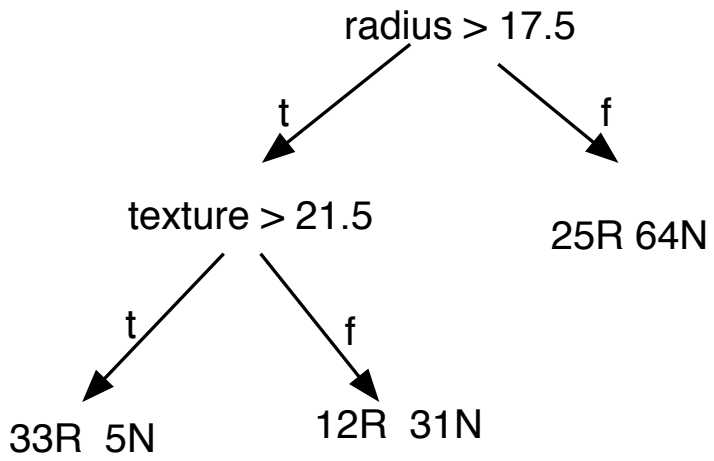
When to consider instance-based learning

- Instances map to points in $\mathbb{R}^n$
- Not too many attributes per instace (< 20)
- Advantages:
  - Training is very fast
  - Easy to learn complex functions over few variables
  - Can give back confidence intervals in addition to the prediction
  - Variable resolution (depends on the data)
  - Does not lose any information
- Disadvantages:
  - Slow at query time
  - *Easily fooled by irrelevant attributes*
  - Cannot be used directly for problems with lots of inputs

Non-metric learning

- The result of learning is not a set of parameters, but there is no distance metric to assess similarity of different instances
- Typical examples:
  - Decision trees
  - Rule-based systems

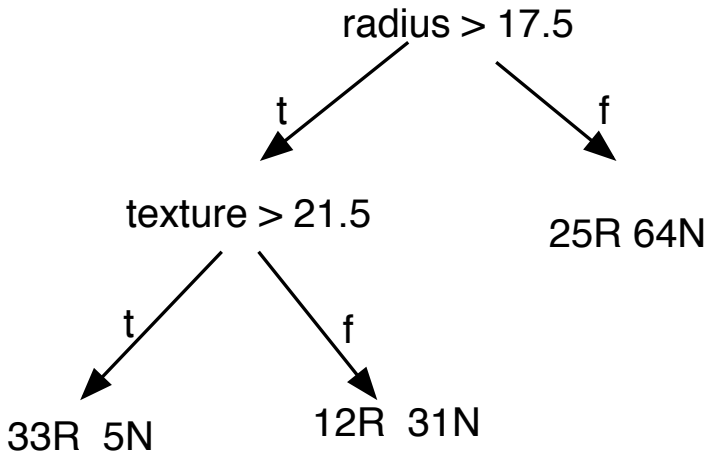
Example: Decision tree for Wisconsin data



- Internal nodes are tests on the values of different attributes
- Tests can be binary or multi-valued
- Each training example $(\mathbf{x}_i, y_i)$ falls in precisely one leaf.

Using decision trees for classification

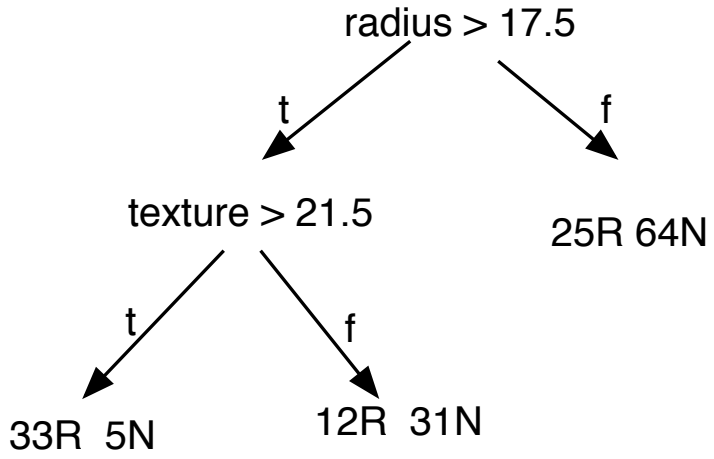
How do we classify a new a new instance, e.g.: *radius=18, texture=12, . . .*



- At every node, test the corresponding attribute
- Follow the appropriate branch of the tree
- At a leaf, one can predict the class of the majority of the examples for the corresponding leaf, or the probabilities of the two classes.

Decision trees as logical representations

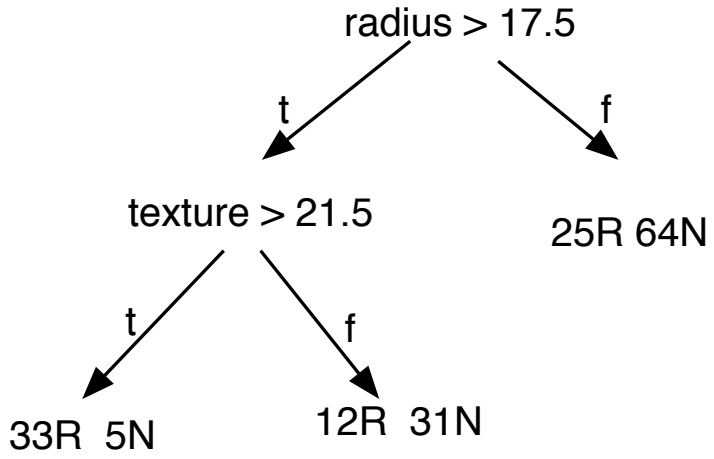
A decision tree can be converted an equivalent set of if-then rules.



IF	THEN most likely class is
radius$>$17.5 AND texture$>$21.5	R
radius$>$17.5 AND texture$\leq$21.5	N
radius$\leq$17.5	N

Decision trees as logical representations

A decision tree can be converted an equivalent set of if-then rules.



IF	THEN P(R) is
radius$>$17.5 AND texture$>$21.5	$\frac{33}{33+5}$
radius$>$17.5 AND texture$\leq$21.5	$\frac{12}{12+31}$
radius$\leq$17.5	$\frac{25}{25+64}$

Decision trees, more formally

- Each internal node contains a *test*, on the value of one (typically) or more feature values
- A test produces discrete outcomes, e.g.,
  - radius > 17.5
  - radius $\in [12, 18]$
  - grade is $\{A, B, C, D, F\}$
  - grade is $\geq B$
  - color is RED
  - $2*$radius$-3*$texture > 16
- For discrete features, typically branch on some, or all, possible values
- For real features, typically branch based on a threshold value

$\Rightarrow$ *A finite set of possible tests* is usually decided before learning the tree; learning comprises choosing the shape of the tree and the tests at every node.

More on tests for real-valued features

- Suppose feature j is real-valued,
- How do we choose a finite set of possible thresholds, for tests of the form $x_j > \tau$?
  - Regression: choose midpoints of the observed data values, $x_{1,j}, x_{2,j}, \ldots, x_{m,j}$

  - Classification: choose midpoints of data values with different y values

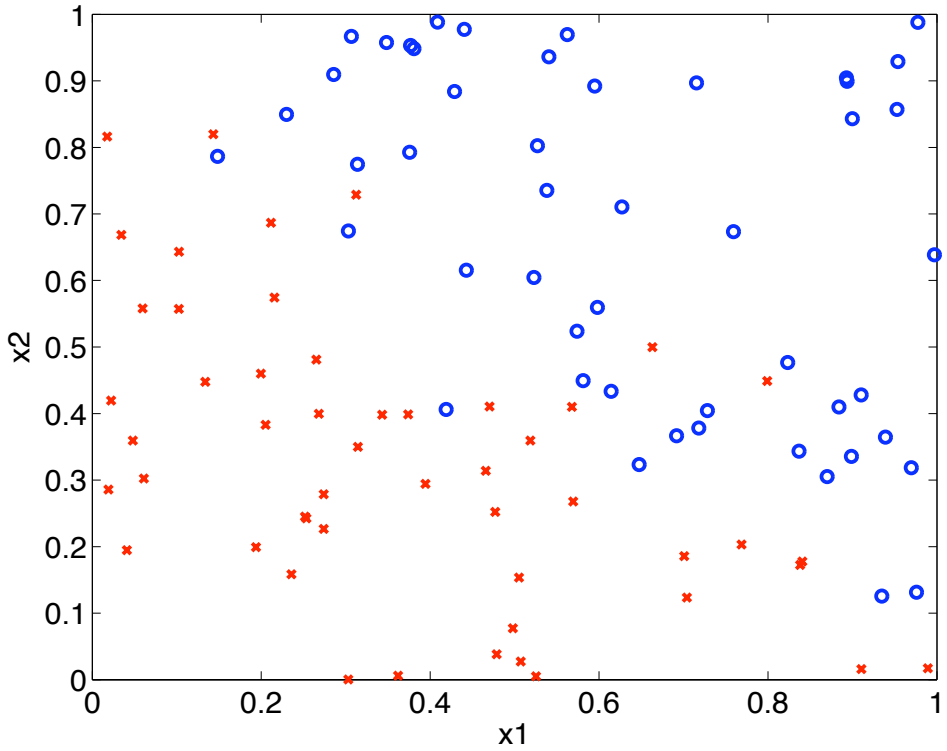
Representational power and efficiency of decision trees

- Suppose the input $\mathbf{x}$ consists of n binary features
- How can a decision tree represent:
  - $y = x_1$ AND x_2 AND ... AND x_n
  - $y = x_1$ OR x_2 OR ... OR x_n
  - $y = x_1$ XOR x_2 XOR ... XOR x_n

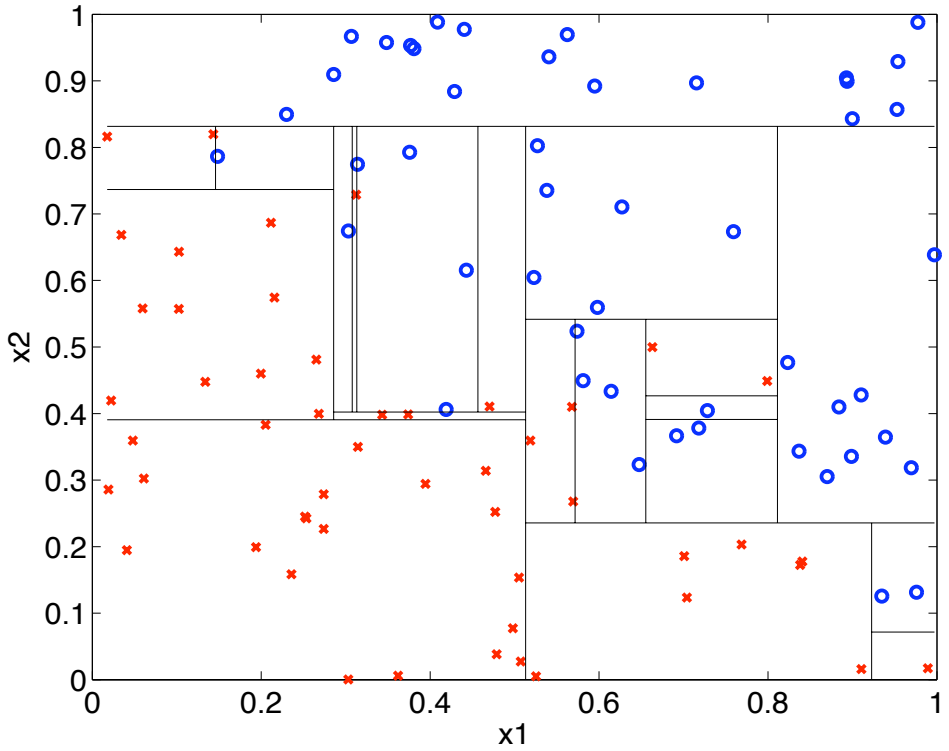
Representational power and efficiency of decision trees

- With typical univariate tests, AND and OR are easy, taking $O(n)$ tests
- Parity/XOR type problems are hard, taking $O(2^n)$ tests
- With real-valued features, decision trees are good at problems in which the class label is constant in large, connected, axis-orthogonal regions of the input space.

An artificial example



Example: Decision tree decision surface



How do we learn decision trees?

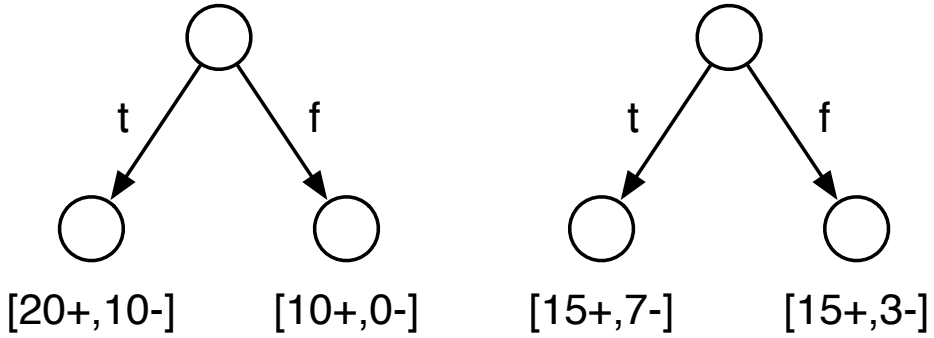
- We could enumerate all possible trees (assuming number of possible tests is finite),
  - Each tree could be evaluated using the training set or, better yet, a validation set
  - But there are many possible trees! Combinatorial problem...
  - We'd probably overfit the data anyway
- Usually, decision trees are constructed in two phases:
  1. An recursive, top-down procedure "grows" a tree (possibly until the training data is completely fit)
  2. The tree is "pruned" back to avoid overfitting
- Both typically use *greedy heuristics*

Top-down induction of decision trees

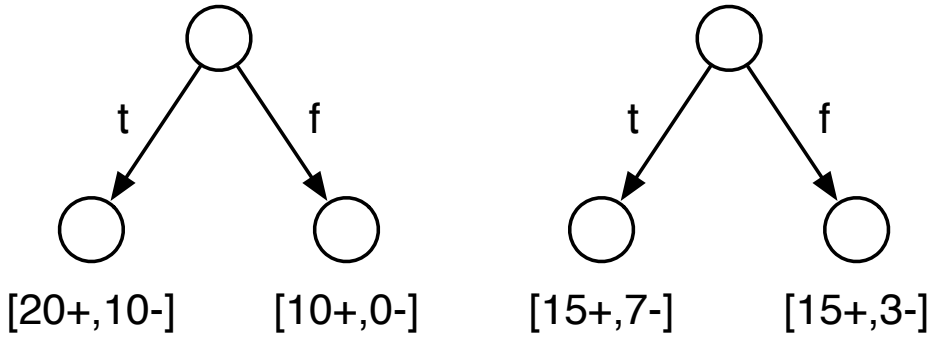
- For a classification problem:
  1. If all the training instances have the same class, create a leaf with that class label and exit.
  2. Pick the *best test* to split the data on
  3. Split the training set according to the value of the outcome of the test
  4. Recurse on each subset of the training data
- For a regression problem - same as above, except:
  - The decision on when to stop splitting has to be made earlier
  - At a leaf, either predict the mean value, or do a linear fit

Which test is best?

- The test should provide *information* about the class label.
- Suppose we have 30 positive examples, 10 negative ones, and we are considering two tests that would give the following splits of instances:



Which test is best?



- Intuitively, we would like an attribute that *separates* the training instances as well as possible
- If each leaf was pure, the attribute would provide maximal information about the label at the leaf
- We need a mathematical measure for the purity of a set of instances

What is information?

- Imagine:
  1. You are about to observe the outcome of a dice roll
  2. You are about to observe the outcome of a coin flip
  3. You are about to observe the outcome of a biased coin flip
  4. Someone is about to tell you your own name
- Intuitively, in each situation you have a different amount of uncertainty as to what outcome / message you will observe.

Information=Reduction in uncertainty

Let E be an event that occurs with probability $P(E)$. If we are told that E has occurred with certainty, then we received

$$I(E) = \log_2 \frac{1}{P(E)}$$

bits of *information*.

- You can also think of information as the amount of "surprise" in the outcome (e.g., consider $P(E) = 1$, $P(E) \approx 0$)
- E.g., result of a fair coin flip provides $\log_2 2 = 1$ bit of information
- E.g., result of a fair dice roll provides $\log_2 6 \approx 2.58$ bits of information.
- E.g., result of being told your own name (or any other deterministic event) produces 0 bits of information

Information is additive

Suppose you have k independent fair coin tosses. How much information do they give?

$$I(k \text{ fair coin tosses }) = \log_2 \frac{1}{1/2^k} = k \text{ bits}$$

A cute example:

- Consider a random word drawn from a vocabulary of 100,000 words: $I(\text{word}) = \log_2 100,000 \approx 16.61$ bits
- Now consider a 1000 word document drawn from the same source: $I(\text{document}) \approx 16610$ bits
- Now consider a 480×640 gray-scale image with 16 grey levels: $I(\text{picture}) = 307,200 \cdot \log_2 16 = 1,228,800$ bits!

$\Longrightarrow$ A picture is worth (more than) a thousand words!

Entropy

- Suppose we have an information source S which emits symbols from an alphabet $\{s_1, \ldots s_k\}$ with probabilities $\{p_1, \ldots p_k\}$. Each emission is independent of the others.
- What is the *average amount of information* when observing the output of S?

$$H(S) = \sum_i p_i I(s_i) = \sum_i p_i \log \frac{1}{p_i} = -\sum_i p_i \log p_i$$

  This is called the *entropy* of S.
- Note that this depends *only on the probability distribution* and not on the actual alphabet (so we can really write $H(P)$).

Interpretations of entropy

$$H(P) = \sum_i p_i \log \frac{1}{p_i}$$

- Average amount of information per symbol
- Average amount of surprise when observing the symbol
- Uncertainty the observer has before seeing the symbol
- Average number of bits needed to communicate the symbol

Entropy and coding theory

- Suppose I will get data from a 4-value alphabet $\mathbf{z}_j$ and I want to send it over a channel. I know that the probability of item $\mathbf{z}_j$ is p_j.
- Suppose all values are equally likely. Then I can encode them in two bits each, so on every transmission I need 2 bits
- Suppose now $p_0 = 0.5$, $p_1 = 0.25$, $p_2 = p_3 = 0.125$. What is the best encoding? What is the expected length of the message over time?

Entropy and coding theory

- Suppose I will get data from a 4-value alphabet $\mathbf{z}_j$ and I want to send it over a channel. I know that the probability of item $\mathbf{z}_j$ is p_j.
- Suppose all values are equally likely. Then I can encode them in two bits each, so on every transmission I need 2 bits
- Suppose now $p_0 = 0.5$, $p_1 = 0.25$, $p_2 = p_3 = 0.125$. What is the best encoding?
  Huffman coding:
  $\mathbf{z}_0 = 0$, $\mathbf{z}_1 = 10$, $\mathbf{z}_2 = 110$, $\mathbf{z}_3 = 111$.
  What is the expected length of the message over time?

$$L = 1 \cdot 0.5 + 2 \cdot 0.25 + 3 \cdot 0.125 + 3 \cdot 0.125 = 1.75$$

$$H = -\sum p_j \log_2 p_j =$$

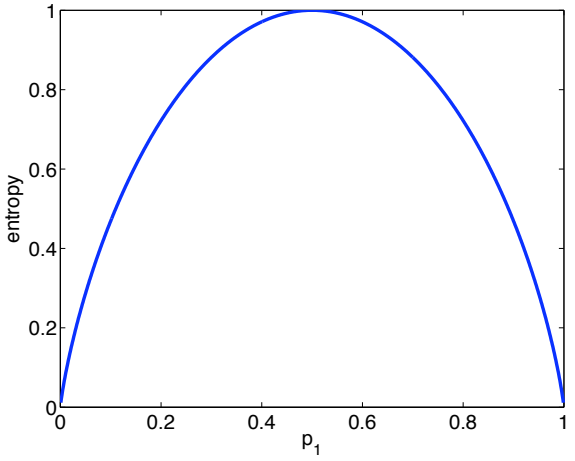
$$-(0.5 \log_2 0.5 + 0.25 \log_2 0.25 + 2 \cdot 0.125 \log_2 0.125) = 1.75$$

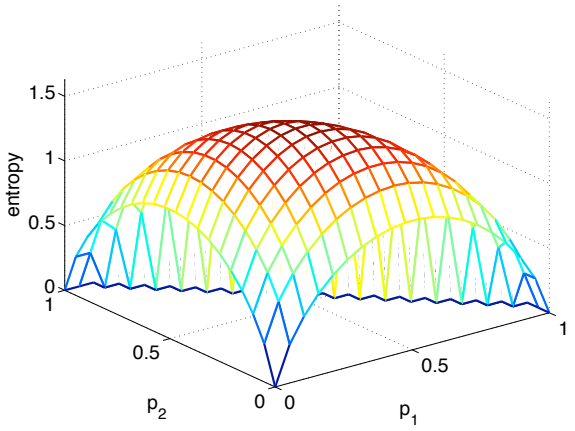
Shannon: there are codes that will communicate the symbols with efficiency arbitrarily close to $H(P)$ bits/symbol. There are no codes that will do it with efficiency greater than $H(P)$ bits/symbol.

Properties of entropy

$$H(P) = \sum_{i=1}^{k} p_i \log \frac{1}{p_i}$$

- Non-negative: $H(P) \geq 0$, with equality if and only if any $p_i = 1$.
- $H(P) \leq \log k$ with equality if and only if $p_i = \frac{1}{k}, \forall i$
- The further P is from uniform, the lower the entropy





Entropy applied to binary classfication

- Consider data set D and let
  - $p_{\oplus} =$ the proportion of positive examples in D
  - $p_{\ominus} =$ the proportion of negative examples in D
- Entropy measures the impurity of D, based on empirical probabilities of the two classes:

$$H(D) \equiv p_{\oplus} \log_2 \frac{1}{p_{\oplus}} + p_{\ominus} \log_2 \frac{1}{p_{\ominus}}$$

  So we can use it to measure purity!

Example

Suppose I am trying to predict output y and I have input x, e.g.:

x =HasKids	y =OwnsDumboVideo
Yes	Yes
Yes	Yes
Yes	Yes
Yes	Yes
No	No
No	No
Yes	No
Yes	No

- From the table, we can estimate $P(y = YES) = 0.5 = P(y = NO)$.
- Thus, we estimate $H(y) = 0.5 \log \frac{1}{0.5} + 0.5 \log \frac{1}{0.5} = 1$.

Example: Conditional entropy

$x =$HasKids	$y =$OwnsDumboVideo
Yes	Yes
Yes	Yes
Yes	Yes
Yes	Yes
No	No
No	No
Yes	No
Yes	No

Specific conditional entropy is the uncertainty in y given a particular x value. E.g.,

- $P(y = YES|x = YES) = \frac{2}{3}$, $P(y = NO|x = YES) = \frac{1}{3}$
- $H(y|x = YES) = \frac{2}{3}\log\frac{1}{(\frac{2}{3})} + \frac{1}{3}\log\frac{1}{(\frac{1}{3})} \approx 0.9183.$

Conditional entropy

- *The conditional entropy,* $H(y|x)$, is the average specific conditional entropy of y given the values for x:

$$H(y|x) = \sum_v P(x = v)H(y|x = v)$$

- E.g. (from previous slide),
  - $H(y|x = YES) = \frac{2}{3}\log\frac{1}{(\frac{2}{3})} + \frac{1}{3}\log\frac{1}{(\frac{1}{3})} \approx 0.9183$
  - $H(y|x = NO) = 0\log\frac{1}{0} + 1\log\frac{1}{1} = 0.$
  - $H(y|x) = H(y|x = YES)P(x = YES) + H(y|x = NO)P(x = NO) = 0.9183 * \frac{3}{4} + 0 * \frac{1}{4} \approx 0.6887$
- Interpretation: the expected number of bits needed to transmit y if both the emitter and the receiver know the possible values of x (but before they are told x's specific value).

Information gain

- Suppose one has to transmit y. How many bits on the average would it save if both the transmitter and the receiver knew x?

$$IG(y|x) = H(y) - H(y|x)$$

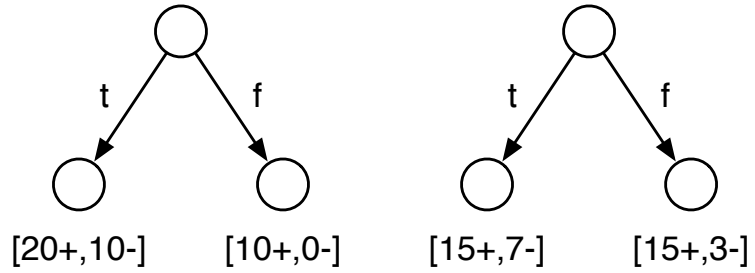
  This is called *information gain*

- Alternative interpretation: what reduction in entropy would be obtained by knowing x
- Intuitively, this has the meaning we seek for decision tree construction

Information gain to determine best test

- We choose, recursively at each interior node, the test that has highest information gain. Equivalently, test that results in lowest conditional entropy.
- If tests are binary:

$$\begin{aligned} IG(D, \text{Test}) &= H(D) - H(D|\text{Test}) \\ &= H(D) - \frac{|D_{\text{Test}}|}{|D|} H(D_{\text{Test}}) - \frac{|D_{\neg\text{Test}}|}{|D|} H(D_{\neg\text{Test}}) \end{aligned}$$



Test 1 Test 2

Check that in this case, Test 1 wins.

Caveats on tests with multiple values

- If the outcome of a test is *multi-valued*, the number of possible values influences the information gain
- The more possible values, the higher the gain! (the more likely it is to form small, but pure partitions)
- C4.5 (the most popular decision tree construction algorithm in ML community) uses only binary tests:
  - Attribute$=$Value for discrete attributes
  - Attribute $<$ or $>$ Value for continuous attributes
- Other approaches consider smarter metrics (e.g. gain ratio), which account for the number of possible outcomes

Alternative purity measures

- For classification, an alternative to the information gain is the *Gini index*:

$$\sum_y P(y)(1 - P(y)) = 1 - \sum_y (P(y))^2$$

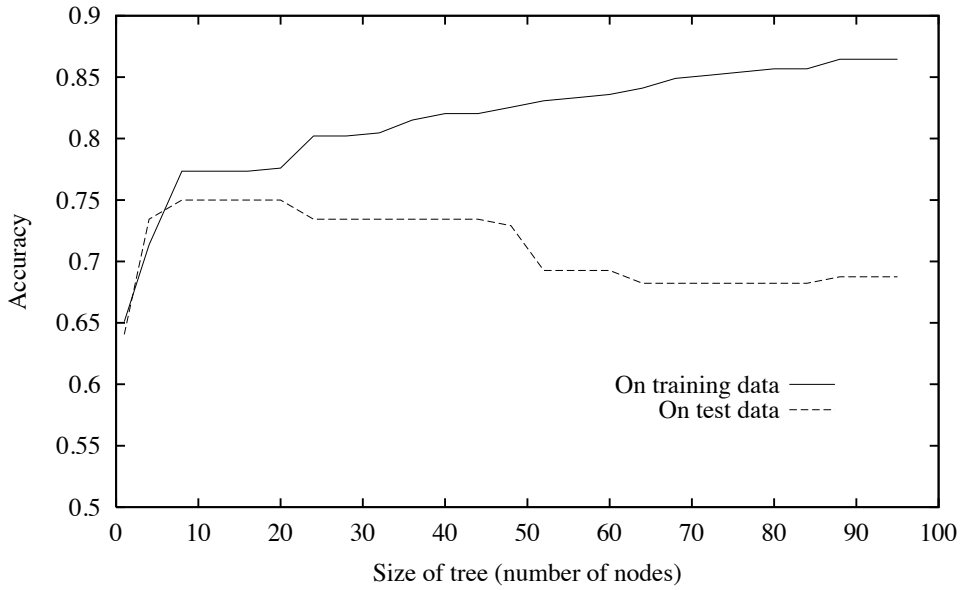
  Same qualitative behavior as the entropy, but not the same interpretation

- For regression trees, purity is measured by the average mean-squared error at each leaf

  E.g. CART (Breiman et al., 1984)

Overfitting in decision trees

- Remember, decision tree construction proceeds until all leaves are pure – all examples having the same y value.
- As the tree grows, the generalization performance starts to degrade, because the algorithm is finding *irrelevant attributes / tests*.



Example from (Mitchell, 1997)

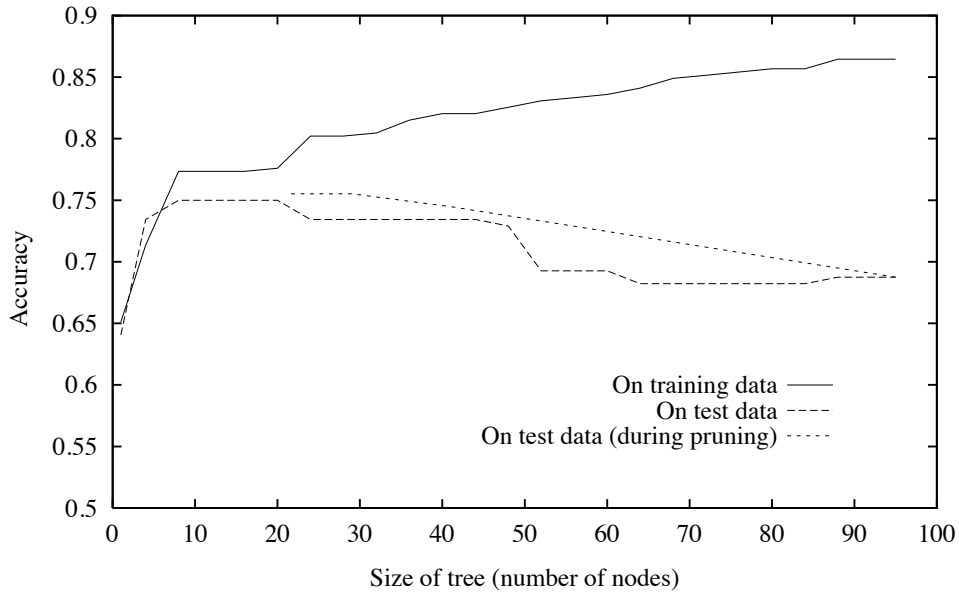
Avoiding overfitting

- Two approaches:
  1. Stop growing the tree when further splitting the data does not yield a statistically significant improvement
  2. Grow a full tree, then *prune* the tree, by eliminating nodes
- The second approach has been more successful in practice, because in the first case it might be hard to decide if the information gain is sufficient or not (e.g. for multivariate functions)
- We will select the best tree, for now, by measuring performance on a separate validation data set.

Example: Reduced-error pruning

1. Split the “training data” into a training set and a validation set
2. Grow a large tree (e.g. until each leaf is pure)
3. For each node:
   (a) Evaluate the validation set accuracy of pruning the subtree rooted at the node
   (b) Greedily remove the node that most improves validation set accuracy, with its corresponding subtree
   (c) Replace the removed node by a leaf with the majority class of the corresponding examples.
4. Stop when pruning starts hurting the accuracy on the validation set.

Example: Effect of reduced-error pruning



Example: Rule post-pruning in C4.5

1. Convert the decision tree to rules
2. Prune each rule independently of the others, by removing preconditions such that the accuracy is improved
3. Sort final rules in order of estimated accuracy

Advantages:

- Can prune attributes higher up in the tree *differently on different paths*
- There is no need to reorganize the tree if pruning an attribute that is higher up
- Most of the time people want rules anyway, for readability

Missing values during classification

- Assign “most likely” value based on all the data that reaches the current node.
- Assign all possible values with some probability.
  - Count the occurrences of the different attribute values in the instances that have reached the same node.
  - Predict all the possible class labels with the appropriate probabilities

Missing values during tree construction

- Introduce an “unknown” value
- Modify information gain to take into account the probability of an attribute being known:

$$IG(D, \text{Test})P(\text{Test})$$

where P(Test) is the fraction of the instances that reached the node, in which the value of the attribute tested by Test was known.

Variations: Constructing tests on the fly

- Suppose we are interested in more sophisticated tests at nodes, such a linear combinations of features:

$$3.1 \times \text{radius} - 1.9 \times \text{texture} \geq 1.3$$

- We can use a fast, simple classifier (such as logistic) to determine a decision boundary, and us it as a test
- Special-purpose methods also search over logical formulae for tests

Costs of attributes

- Include cost in the metric, e.g.

$$\frac{IG^2(D, \text{Test})}{Cost(\text{Test})}$$

- Mostly a problem in specific domains (e.g. medicine).

  Multiple metrics have been studied and proposed, without a consensus.

Decision and regression tree summary

- Very fast learning algorithms (e.g. C4.5, CART)
- Attributes may be discrete or continuous, no preprocessing needed
- Provide a general representation of classification rules
- Easy to understand! Though. . .
  - Exact tree output may be sensitive to small changes in data
  - With many features, tests may not be meaningful
- In standard form, good for (nonlinear) piecewise axis-orthogonal decision boundaries – not good with smooth, curvilinear boundaries
- In regression, the function obtained is discontinuous, which may not be desirable
- Good accuracy in practice – many applications!
  - Equipment/medical diagnosis
  - Learning to fly
  - Scene analysis and image segmentation