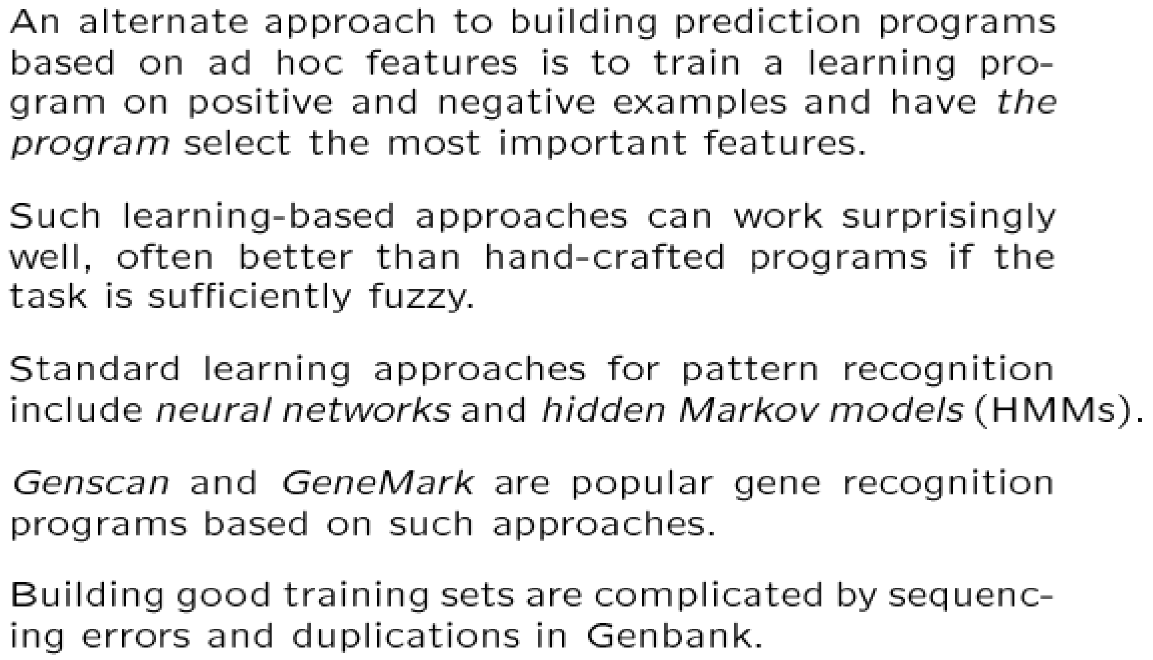
HMM based approaches to gene prediction



An alternate approach to building prediction programs based on ad hoc features is to train a learning program on positive and negative examples and have *the program* select the most important features.

Such learning-based approaches can work surprisingly well, often better than hand-crafted programs if the task is sufficiently fuzzy.

Standard learning approaches for pattern recognition include *neural networks* and *hidden Markov models* (HMMs).

Genscan and *GeneMark* are popular gene recognition programs based on such approaches.

Building good training sets are complicated by sequencing errors and duplications in Genbank.

Finding CG islands

CG islands are regions in DNA sequences where the dimer CG repeatedly occurs.

CG sites are typically modified by *methylation*. Methylated sites are likely to mutate to TG sites, so concentations of CGs denote where methylation is suppressed and thus have biological significance.

My approach to locating such islands would likely be to produce a list of all positions where CG's occur, and then use an $O(n^2)$ algorithm or heuristic to quickly identify all sufficiently long, sufficiently dense sequences.

An alternate approach would be to *train* a program on appropriately identified examples of CG islands and non-islands and have it *learn* to recognize them.

Hidden Markov Models (HMMs), neural networks, decision trees, and other AI formalisms offer approaches to machine learning.

Markov models

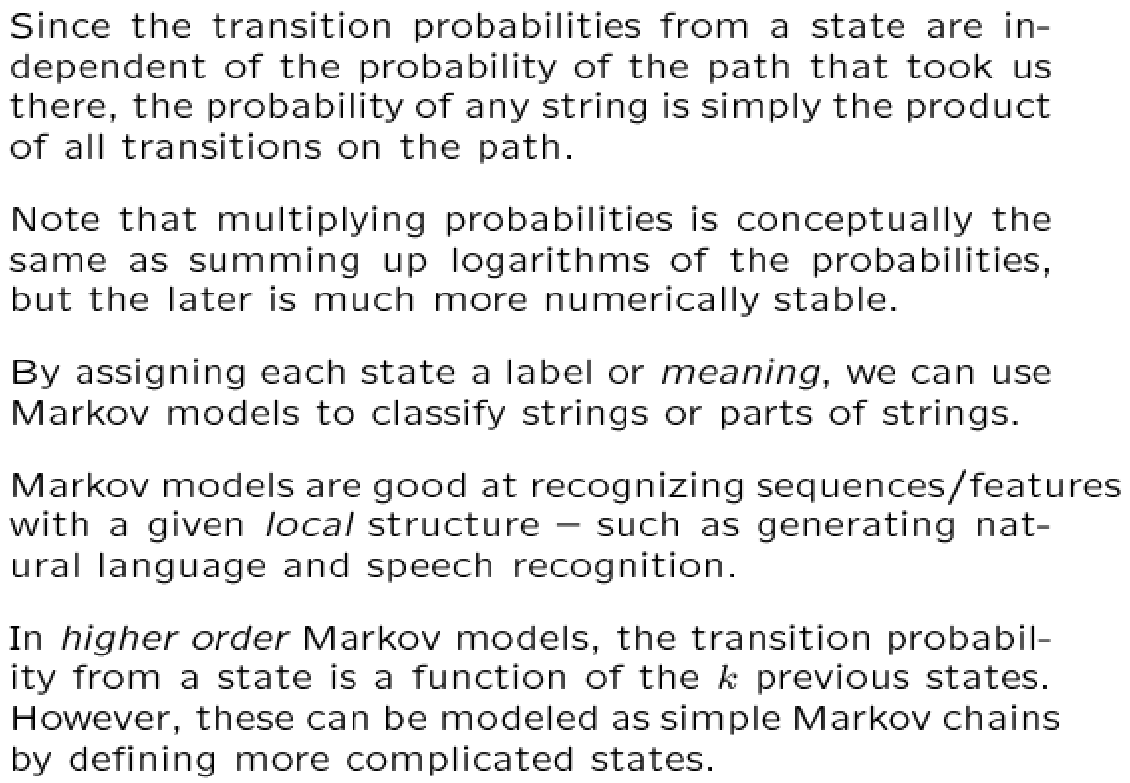
Markov chains are networks of *states* where there is a given probability of *transition* between each pair of states.

The probability of being in state s at time t is completely a function of (1) the probability of each state at time $t-1$, and (2) the state transition function giving the probability of mapping each state to s.

The states in a Markov chain can be used to record some knowledge about previous states, but *not* the path we used to get to this state if there is any branching.

Typically a character or symbol is associated with each state transition. Thus any string defines a path through the model.

Markov models



Since the transition probabilities from a state are independent of the probability of the path that took us there, the probability of any string is simply the product of all transitions on the path.

Note that multiplying probabilities is conceptually the same as summing up logarithms of the probabilities, but the later is much more numerically stable.

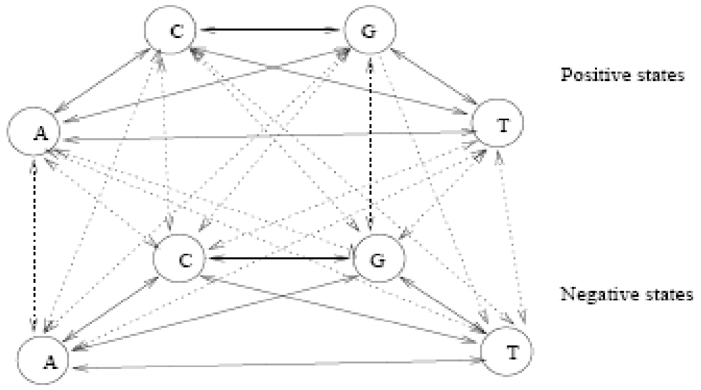
By assigning each state a label or *meaning*, we can use Markov models to classify strings or parts of strings.

Markov models are good at recognizing sequences/features with a given *local* structure – such as generating natural language and speech recognition.

In *higher order* Markov models, the transition probability from a state is a function of the k previous states. However, these can be modeled as simple Markov chains by defining more complicated states.

Recognizing CG islands with Markov models

By separately tabulating the base pair transition probabilities in CG islands and non-islands, we can use simple Markov models for recognition.



The critical transition CG has a probability of 0.27 in the CG island examples, but is only 0.078 in the negative examples

Recognizing CG islands with Markov models

The two models can be collapsed into one provided we allow more than one possible next state for a given character.

This permits us to arbitrarily transition back and forth between the two types of states, enabling subsequence recognition.

Such models are called *hidden Markov models*, since the actual state the model is in as a function of the string is “hidden” from the observer.

Finding Optimal Paths through HMMs

HMMs represent *non-deterministic automata* where there can be exponentially many ways through the machine for any string.

The *Viterbi* algorithm gives a simple $O(nm^2)$ dynamic programming algorithm to find the most probable path for an n character string through an m state automata.

The labels of the states on this path can be used to annotate the input sequence.

The probability that the ith character passes through state j is clearly defined given (1) the probability we are in each of the m states associated with the $(i-1)$st character, and (2) the probability of each transition from states from the $(i-1)$st to the ith characters.

The first is computed by dynamic programming, while the second is specified by the input automata.

The Backward algorithm

We have seen how the Viterbi algorithm can be used to find the highest probability path through a model, and that the labels of the states on this path can be used to annotate the sequence.

But fixating on a single path could be risky.

An alternate and perhaps more defensible annotation strategy would be based on knowing the probability $P(x, \pi_i = k)$ that the ith symbol of the sequence being in state k of the automata summed over *all* paths for a sequence x.

$$P(x, \pi_i = k) = P(x_1 \ldots x_i, \pi_i = k) P(x_{i+1} \ldots x_L | x_1 \ldots x_i, \pi_i = k)$$

$$P(x, \pi_i = k) = P(x_1 \ldots x_i, \pi_i = k) P(x_{i+1} \ldots x_L | \pi_i = k)$$

Given the probabilities of being in each state at each time, we can annotate each symbol/position according to which classification has the highest weight.

The values of $P(x_1 \ldots x_i, \pi_i = k)$ are exactly what is computed by the Viterbi algorithm.

The values of $P(x_{i+1} \ldots x_L | \pi_i = k)$ can be computed analogously in a right-left dynamic programming computation.

Training HMMs

If the fine structure of the training examples are properly annotated in accord with the states of the model, the state transition probabilities can be easily determined.

If not, parameters can be found through *iterative* algorithms, where each training sequence is run through the model and weights adjusted to increase the probability that training examples are correctly classified.

In the *Baum-Welch* algorithm, we calculate the forward and backward probabilities for each sequence/each state, and adjust accordingly. In the *Viterbi* algorithm, we only reinforce the strongest path for an input sequence.

The set of training sequences is run through the model multiple times until either (1) we have hit a local optimum and the parameters stop changing, or (2) the quality of the model is good enough.

The quality of the model can be estimated by multiplying (or summing the logarithms of) the probability of each of the training sequences as scored by the model.

To guard against *overfitting* the exact training instances, each training example might have random noise added to it, with the amount of added noise decreasing as the training progresses.

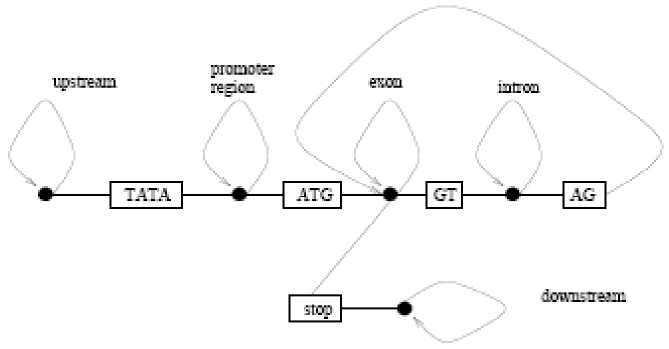
Topologies

To reduce the number of parameters the model must learn, it is often a good idea to initialize, force, or combine certain parameters in light of a priori knowledge.

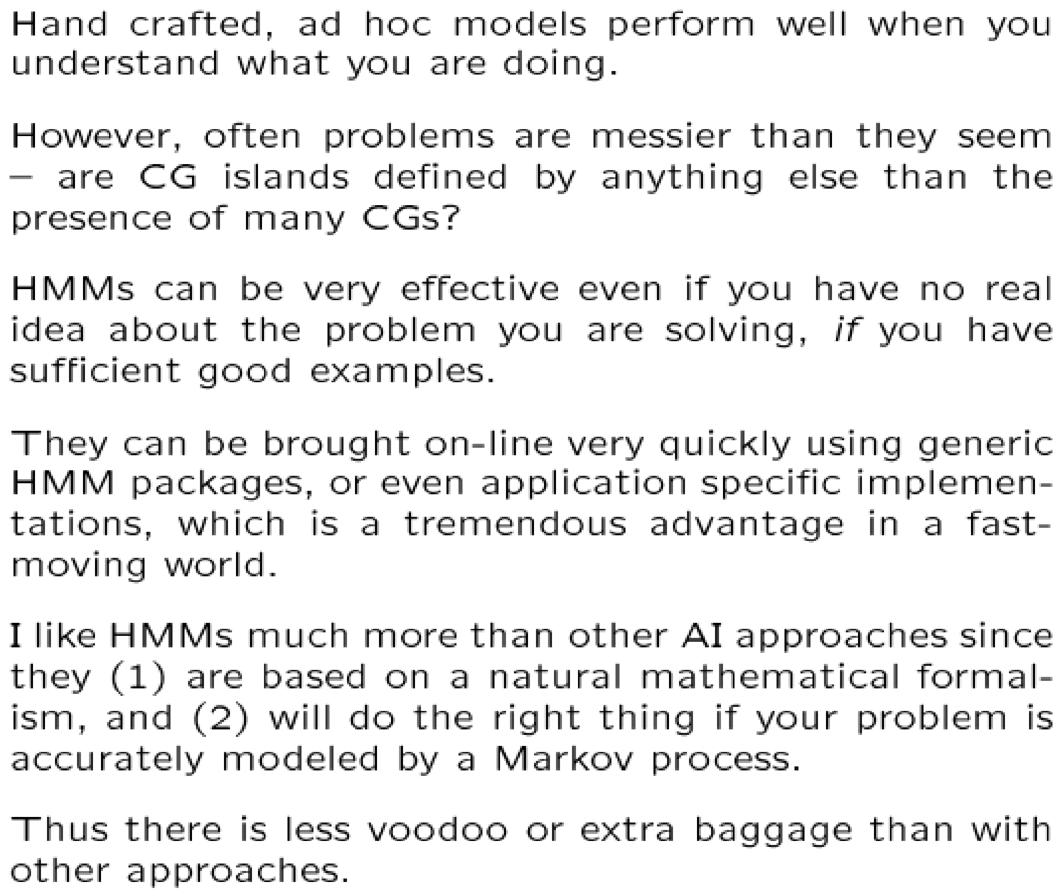
It is a bad idea to set certain parameters to zero just because you haven't seen any examples in a given small dataset.

Forcing certain transition probabilities to zero imparts a non-complete *topology* to the network.

Multistage recognition problems such as prokaryotic genes (promoter sites, start codon, coding sequences, stop codon) are best modeled as progressing sequentially across stages, moving backwards on errors.



HMMs or Ad Hoc models?



Hand crafted, ad hoc models perform well when you understand what you are doing.

However, often problems are messier than they seem – are CG islands defined by anything else than the presence of many CGs?

HMMs can be very effective even if you have no real idea about the problem you are solving, *if* you have sufficient good examples.

They can be brought on-line very quickly using generic HMM packages, or even application specific implementations, which is a tremendous advantage in a fast-moving world.

I like HMMs much more than other AI approaches since they (1) are based on a natural mathematical formalism, and (2) will do the right thing if your problem is accurately modeled by a Markov process.

Thus there is less voodoo or extra baggage than with other approaches.

Validating the model

Our model can only succeed if the training set is sufficiently *large* and *representative* for the learned weights to represent reality.

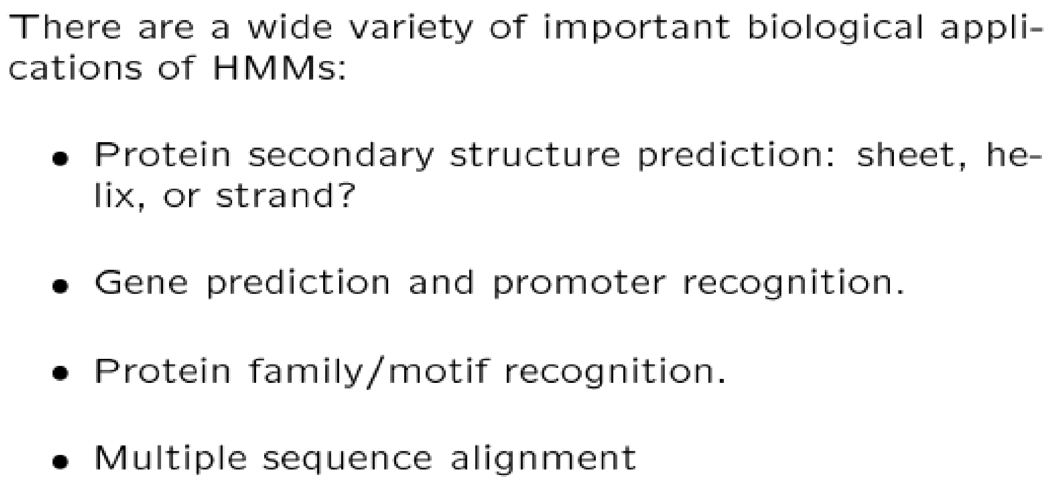
One approach to cross-validating a model from a small data set is to train a model from each set of $n-1$ training examples and see how well it predicts the remaining one.

HMMs can easily be built from *any* set of labeled examples, e.g. stock market historical data. Such models usually do great in predicting the past on small enough training sets.

Remember: *garbage-in, garbage-out*!

Cautionary tales from neural networks are appropriate to remember, (1) distinguishing cars from trucks from images, and (2) red-lining loan models.

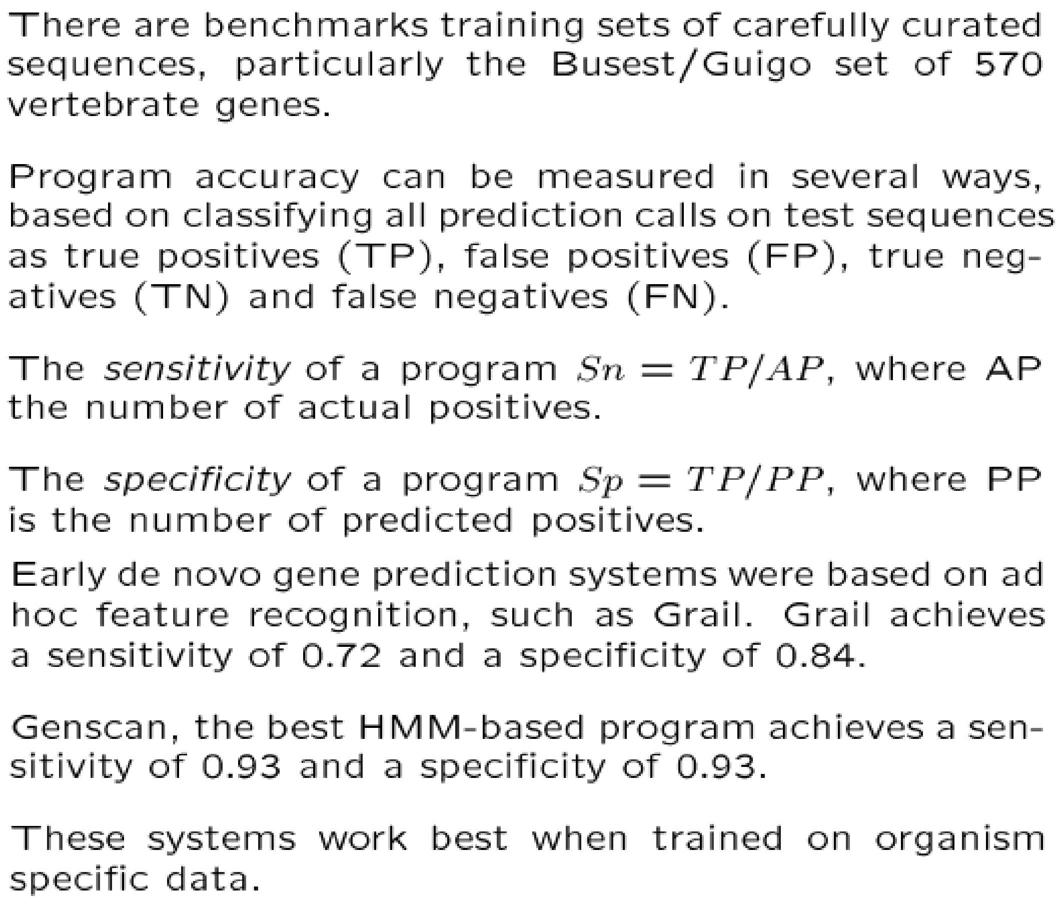
Biological applications of HMMs



There are a wide variety of important biological applications of HMMs:

- Protein secondary structure prediction: sheet, helix, or strand?
- Gene prediction and promoter recognition.
- Protein family/motif recognition.
- Multiple sequence alignment

Gene Prediction systems



There are benchmarks training sets of carefully curated sequences, particularly the Busest/Guigo set of 570 vertebrate genes.

Program accuracy can be measured in several ways, based on classifying all prediction calls on test sequences as true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN).

The *sensitivity* of a program $Sn = TP/AP$, where AP the number of actual positives.

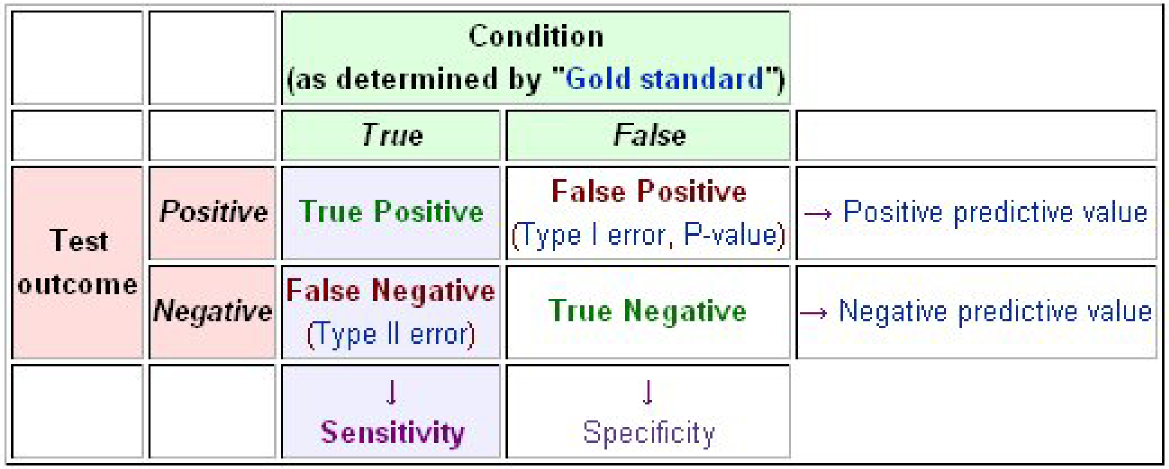
The *specificity* of a program $Sp = TP/PP$, where PP is the number of predicted positives.

Early de novo gene prediction systems were based on ad hoc feature recognition, such as Grail. Grail achieves a sensitivity of 0.72 and a specificity of 0.84.

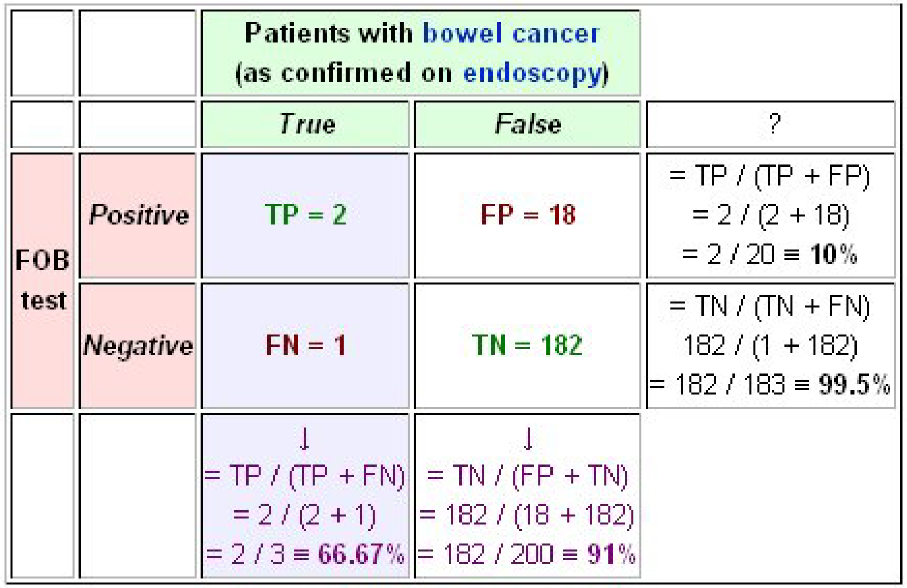
Genscan, the best HMM-based program achieves a sensitivity of 0.93 and a specificity of 0.93.

These systems work best when trained on organism specific data.

Sensitivity and Specificity



		Condition (as determined by "Gold standard")		
		True	*False*	
Test outcome	*Positive*	True Positive	False Positive (Type I error, P-value)	→ Positive predictive value
	Negative	False Negative (Type II error)	True Negative	→ Negative predictive value
		↓ Sensitivity	↓ Specificity	



		Patients with bowel cancer (as confirmed on endoscopy)		
		True	*False*	?
FOB test	*Positive*	TP = 2	FP = 18	= TP / (TP + FP) = 2 / (2 + 18) = 2 / 20 ≡ **10%**
	Negative	FN = 1	TN = 182	= TN / (TN + FN) 182 / (1 + 182) = 182 / 183 ≡ **99.5%**
		↓ = TP / (TP + FN) = 2 / (2 + 1) = 2 / 3 ≡ **66.67%**	↓ = TN / (FP + TN) = 182 / (18 + 182) = 182 / 200 ≡ **91%**	