Lectures 12, 13, and 14:
Gene Prediction

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

Sequence Annotation

As new DNA sequence data becomes available, we seek to identify interesting features in this raw text.

The most interesting features are *genes*, the portions of the chromosome which describe how to make proteins.

Since genes and the promoter sites associated with them make promising drug candidates, there is a considerable pressure to quickly identify them computationally.

Indeed, automatic annotation is a big game. Every genome sequencing project is expected to do annotation prior to publication.

Transcription

Transcription is the process of copying the portion of the DNA containing a gene into RNA.

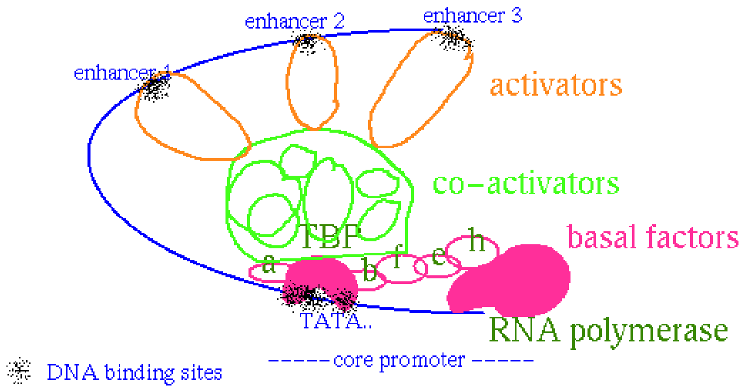
Something has to happen to instigate transcription at genes and not at non-coding regions. Thus there must be signals in the DNA sequence which tell where to start the transcription.

An enzyme called *RNA polymerase* binds to specific patterns at approximately 10 and 35 bases before the gene to start transcription.

Other binding sites upstream from before the gene called *promoters* help signal when to express or inhibit the gene from expressing as RNA.

Termination

Transcription stops when it encounters a DNA palindrome flanking repeated *A*s, forming a ‘knot’.

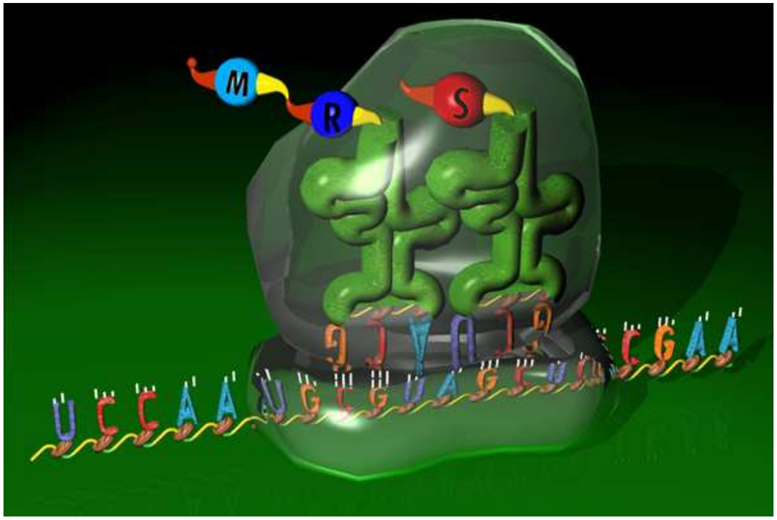


Translation

Translation is the process of building proteins according to the template RNA.

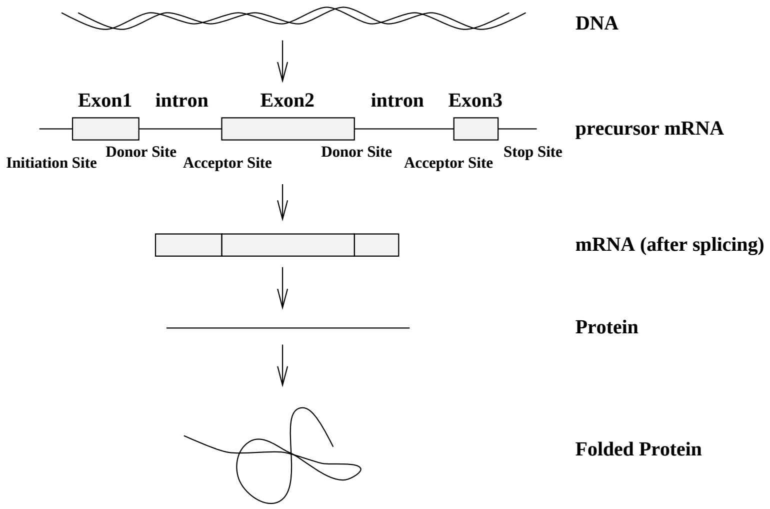
A complex molecule called a *ribosome* works its way along an RNA molecule, grabbing the appropriate amino acid for the next codon and adding to the end of the given protein.

The appropriate bases get to the right places by essentially random motion, guided by electrostatic forces. Binding sites ensure that the right things stick together when they bang into each other.


M
R
S
U C C A A U G C G U A G C U C C G A A

Introns and Exons

Gene recognition in higher organisms (eukaryotes) is complicated by the presence of *introns*, or non-coding regions. The coding regions of genes are called *exons*.

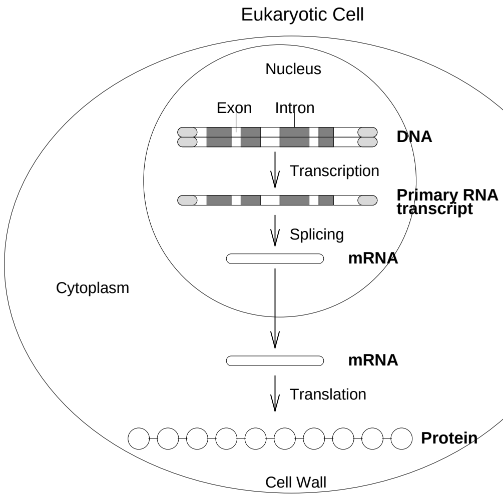


Why Introns?

There is controversy about why introns exist. Presumably it is easier to evolve new genes by shuffling small parts, i.e. exons. Some theorize that prokaryotes originally also had introns, but lost them.

Synthesis on a Cellular Level

DNA in Eukaryotes resides in the cell’s *nucleus*, but proteins are translated outside the nucleus.



Issues of how proteins/RNA cross membrane boundaries are critical in understanding their function, and designing drugs.

Features Which Ease Gene Prediction

In general, introns are flanked by *donor* and *acceptor* sites GT and AG – however, such pairs should each happen by chance every $4^2 = 16$ bases.

Genes start with ATG and end with a stop codon (TAA, TAG, or TGA) – however, such codons should happen every $64/3 \approx 20$ codons.

The length of all coding regions must be a multiple of three – however coding regions can be split over multiple exons.

The distribution of base triples and heximers differs between coding and non-coding regions – but you need a sufficiently long enough region to trust statistical variations.

Problems Which Complicate Gene Prediction

Gene transfer mechanisms often introduce extra copies of genes into genomes, which then diverge through evolution. Distinguishing broken *pseudo-genes* from working genes is a difficult problem.

Sequencing errors can step on donor/acceptor sites and cause apparent frame shifts.

Exons can be separated by several thousand bases.

Genes can overlap each other, appear in different reading frames and on different strands.

Exons can be assembled in multiple ways through *alternative splicing*.

Laboratory-Based Approaches to Gene Prediction

The traditional way to find genes was to do it in the laboratory. One method is to extract and sequence RNA, since most RNA is expressed to code for proteins.

A problem with such laboratory methods is that relatively few genes tend to dominate the population of expressed sequences, and hence one discovered duplicates instead of new genes.

Directly sequencing proteins is a difficult procedure, but is becoming easier through mass spectrometry.

Feature-Based Approaches to Gene Prediction

Gene recognition systems such as *Grail*, *GeneID*, and *GeneParser* work by searching for various ad hoc features of genes, and then identifying regions which score high enough. Typical features include codon bias, donor / acceptor sites, and coding frame length.

Since stop codons should occur every 20 codons or so, long *open reading frames* or ORFs without stop codons are strongly suggestive of genes.

Dynamic programming can be used to identify the highest scoring regions.

The best gene recognition systems tend to be species-specific, trained on examples of known genes in the given organism.

Homology-Based Approaches to Gene Prediction

Biology is an inherently finite discipline. There are only a given number of genes in each of a given number of species. Further, because of evolution, we would assume that there are strong homologies between genes in related species.

Homology-based gene prediction systems such as *Procrustes* scan databases find similarities to previously identified coding regions.

Such homology-based approaches can only identify previously known genes, of course, but the fraction of known genes is growing rapidly.

Conserved Sequences

A different homology-based approach to identify totally unknown genes is to compare two whole genomes and look for conserved regions, on the theory that sequence is only conserved if it is important.

Alignment of many genomes (e.g. four yeasts) can be used to accurately determine gene boundaries and eliminate psuedogenes.

HMM 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 on fuzzy tasks.

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.

Learning Methods

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.

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.

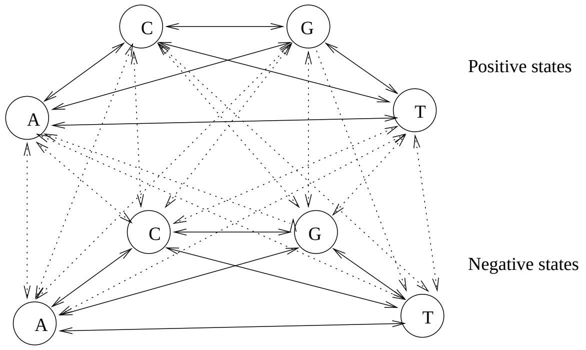
Higher-Order Markov Models

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 MMs

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

Hidden 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 Thru 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.

Computations

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.

A similar algorithm can be used to find the probability of winding up in each state for a given string, by summing instead of maximizing.

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.

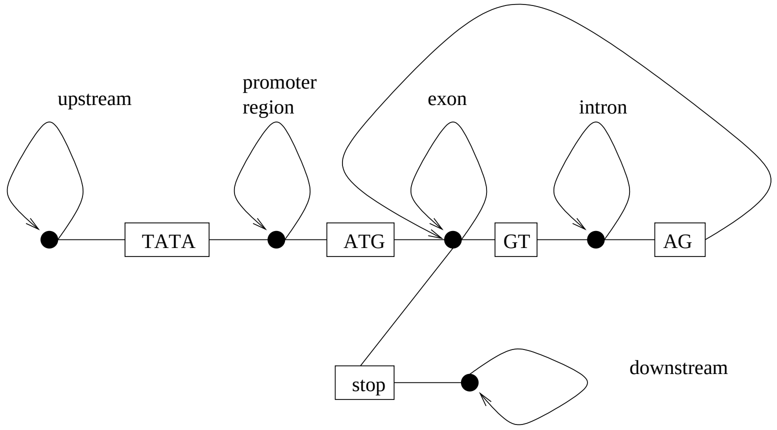
Topologies

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.


upstream
promoter region
exon
intron
TATA
ATG
GT
AG
stop
downstream

Sequence Homology Models

Topologies for sequence homology should permit the insertion and deletion of symbols.

Silent states or ϵ-moves can shift to successors without any input characters. These can be used to reduce the number of transition parameters needed, at the cost of restricting the model topology.

The previously described learning algorithms can be easily extended provided there are no cycles of all silent states.

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

HMM models can only succeed if the training set is sufficiently *large* and *representative*.

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.

The approximate correlation AC is

$$\begin{aligned} AC \;=\; & ((TP/(TP+FN))+(TP/(TP+FP))+ \\ & (TN/(TN+FP))+(TN/(TN+FN)))/2-1 \end{aligned}$$

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.