Tutorial

Stochastic Modeling Techniques: Understanding and using hidden Markov models

Leslie Grate
leslie@cse.ucsc.edu

Richard Hughey
rph@cse.ucsc.edu

Kevin Karplus
karplus@cse.ucsc.edu

Kimmen Sjölander
kimmen@cse.ucsc.edu

Contents

1 Introduction

This tutorial is organized conceptually to provide, first, a theoretical framework for stochastic modeling, and second, to enable readers to use stochastic models to their advantage. The tutorial includes a wide variety of examples, mostly drawn from the Sequence Alignment and Modeling System (SAM) [20, 16], which will be the focus of the hands-on session in the second half of the tutorial. Many of the tasks we discuss can also be performed (sometimes better!) with HMMer, about which Sean Eddy will talk in the second half of the tutorial [10, 9]. The appendix includes both the SAM and the HMMer documentation.

If you have any questions about this material, feel free to direct them to any of the authors. If you have specific questions relating to only one section, you may wish to ask the primary author of that section, as listed at the top of the section. The SAM hidden Markov model implementation is constantly undergoing revisions and additions. If you have any questions, comments, or suggestions about current or future features, please contact Richard Hughey (rph@cse.ucsc.edu, sam-info@cse.ucsc.edu). If you would like to try out the SAM WWW server, or obtain a copy of SAM, please read our WWW page

`http://www.cse.ucsc.edu/research/compbio/sam.html`

for instructions. If you have comments or questions about HMMer, contact Sean Eddy (email address: eddy@genome.wustl.edu) or visit the HMMer WWW page

`http://genome.wustl.edu/eddy/hmm.html`

2 Mathematical Foundations of Stochastic Models

Kevin Karplus (karplus@cse.ucsc.edu)

This section will present a mathematical foundation for the stochastic approach to modeling biological sequences. The basic idea behind stochastic modeling is to construct a *model* that describes a set of sequences, then to use the model for finding related sequences, or examine the model to determine properties of the sequences.

This section is almost entirely mathematical foundations—the interesting applications to biology don't really start until Section 3, where we describe a particular type of stochastic model: the hidden Markov model.

The mathematical foundations are necessary for answering questions like "Which sequences in a database fit a model?" or "How well do they fit?" or "Which of a set of models best describes a set of sequences?" The first two questions arise naturally when searching a database for examples of a motif or protein family, or when trying to find introns, exons, and splice sites in a stretch of DNA. The last question comes up in the *fold-recognition problem*, when the models represent different possible protein folds or domains, and we want to find the most likely fold for a new protein.

The approach used in stochastic modeling to answer these questions mathematically employs Bayesian statistics and information theory as the foundation. This section attempts to give a very brief overview of these fields as they apply to biosequence analysis. The approach is very general and can be applied to many different sorts of models: alignment to single sequences, profiles, hidden Markov models, simple Markov chains, stochastic context-free grammars, threading models, ...

After this general introduction, we'll focus more closely on hidden Markov models, which we have found to be particularly useful for modeling protein motifs and families. (For RNA, we've had some success with stochastic context-free grammars, and for DNA parsing we've used mainly profiles, simple Markov models, and neural nets.)

2.1 What is a model?

This tutorial is about stochastic models in general, and hidden Markov models in particular. Before we get into details about how the models work, it would be good to make sure we all have the same understanding of what a model is.

There are two rather different views of models in the scientific community. One view is a *mechanistic* one, in which models elucidate the mechanism by which something happens. These models are very powerful, but they are also very difficult to create, often requiring years of experimental work and difficult intellectual insights. A different view of models treats them as "black boxes," and makes no claims that the mechanism of the model matches anything in the real world. In this approach, a model's value is determined solely by the accuracy of its predictions, not by the mechanism used to make those predictions.

Making numerically accurate and fully mechanistic models is rarely possible in the realm of biosequences—there is far too much that is still unknown about how large, complex molecules work. Most of the stochastic modeling techniques are black-box techniques; they examine the data and try to fit some class of models to it, without making any claim that the models explain the data.

The scientific test for a mechanistic model is a combination of its predictive power, its elegance, and its consistency with other accepted models. The scientific test for a black-box model is mainly its predictive power.

In the arena of sequence analysis and modeling, the sorts of predictions a black-box model can make are somewhat limited. These models are mainly designed for recognition, discrimination, and database search tasks, answering questions like "Is this protein a globin?" or "Does this look more like a hemoglobin or a myoglobin?" or "What are all the examples of sequences that look like these known calcium-binding sites?"

The models of most interest to us today, hidden Markov models, fall somewhere between the extremes of mechanistic models and pure black-box models. They don't provide mechanistic explanations, but they

have some internal structure that can be examined for biological insights. Using hidden Markov models, we can sometimes answer more detailed questions, such as "What amino acids in this sequence correspond to the ones that bind calcium in this other sequence?" In Section 4.2, when we look in more detail at hidden Markov models, we'll see how these models help answer these more probing questions. First, let's start with the more general questions that don't require any knowledge of the inside of the black box.

The basic idea of a stochastic black-box model M is to assign a number $P_M(s)$ to every possible sequence s. For ease in interpreting and manipulating the numbers, we add the constraint that the infinite sum of the numbers is one,

$$\sum_s P_M(s) = 1 \ ,$$

so that the numbers can be interpreted as probabilities. Note that $P_M(s)$ is not the probability that the sequence belongs to the interesting class—rather it is the probability that if you select a sequence randomly from the interesting set, you will get this particular sequence. In the next section we'll look at how to manipulate these probabilities to answer the recognition and discrimination questions.

For example, let's say we want to recognize the following sequences {AAACA, ATA, ATACA, TACA} and no others. We could create a model that assigns $P(\text{AAACA}) = 0.25$, $P(\text{ATA}) = 0.25$, $P(\text{ATACA}) = 0.25$, $P(\text{TACA}) = 0.25$, and zero to all other sequences. Of course, biologically interesting models are more complex than this, since we want to recognize not just a small set of already known sequences, but a large class of closely related sequences.

Furthermore, we won't usually have such a sharp cutoff between sequences that fit the model and ones that don't—there will be a fuzzy area where sequences are somewhat similar to ones we want to recognize. That is, we are not dealing with precisely defined sets of sequences, but with probability distributions over all possible sequences.

In this small example, we might want to assign small probabilities to sequences similar to the ones in the set, such as AAAGA or AACA. To keep the sum over all sequences equal to one, we would have to "steal probability" from the other sequences in the set. Much of the work in stochastic modeling involves coming up with disciplined ways to assign these probabilities so that they reflect the real distributions of sequences.

2.2 Bayesian statistics—when does a model fit a sequence?

The stochastic model introduced in the first section is not directly usable, because it answers the wrong question. It is designed to answer the question: if a sequence is drawn from the distribution of sequences modeled by a particular model, what is the probability of getting this particular sequence? The recognition question we want to answer is: given this particular sequence, what is the probability that it came from the distribution described by this model?

If we use the notation of conditional probability, we can express $P_M(s)$ as

$$\mathrm{P}\left(x = s \;\middle|\; x \text{ is drawn from the model } M\right) \ .$$

This is typically abbreviated as $\mathrm{P}\left(s \;\middle|\; M\right)$, which is read, "the probability of s given M." Answering the recognition question is then a matter of computing the conditional probability

$$\mathrm{P}\left(x \text{ is drawn from the model } M \;\middle|\; x = s\right) \ ,$$

which is typically abbreviated as $\mathrm{P}\left(M \;\middle|\; s\right)$.

Bayes' rule gives us a way to do this computation:

$$\mathrm{P}\left(M \;\middle|\; s\right) = \frac{\mathrm{P}\left(s \;\middle|\; M\right) P(M)}{P(s)} \ .$$

All we need to know are two *prior probabilities*: the probability $P(M)$ that x is drawn from model M and the the probability $P(s)$that $x = s$. These prior probabilities are in some very real sense unknowable, and so the simplest form of the recognition question is unanswerable.

Although the situation for pure recognition looks hopeless, there is a standard solution—we turn all such questions into discrimination questions. Instead of asking "What it is the probability that the sequence came from model M?" we instead ask "What are the the odds that the sequence came from model M rather than model N?" That is we compute

$$\begin{aligned}\frac{\mathrm{P}\left(M \;\middle|\; s\right)}{\mathrm{P}\left(N \;\middle|\; s\right)} &= \frac{\mathrm{P}\left(s \;\middle|\; M\right) P(M)}{P(s)} \frac{P(s)}{\mathrm{P}\left(s \;\middle|\; N\right) P(N)} \\ &= \frac{\mathrm{P}\left(s \;\middle|\; M\right)}{\mathrm{P}\left(s \;\middle|\; N\right)} \frac{P(M)}{P(N)} \; .\end{aligned}$$

Now, in addition to the numbers provided by the models ($\mathrm{P}\left(s \;\middle|\; M\right) = P_M(s)$ and $\mathrm{P}\left(s \;\middle|\; N\right) = P_N(s)$) we need only one number, the prior expectation of the relative probability of the two models $\frac{P(M)}{P(N)}$. Since we no longer need the absolute probabilities of the models or the sequences, this is a much more manageable problem.

To answer the recognition question using this technique, we need to make up a *null model* N. The null model is a model that attempts to match all the sequences in the universe of possible sequences. This may be a model that tries to fit all the sequences in the database we are searching, or one that fits some theoretical universe of possible protein sequences. Note that it may do a good job of fitting the database or a poor one. A poorly fitting null model may cause the model M to fit more sequences than we had intended, since our recognition test is a competition between the models. Some people view the null model as the model that fits the *null hypothesis*; more correctly, the null model defines what the null hypothesis is.

Once we have defined a null model, we then can say that the sequence fits model M if $\mathrm{P}\left(M \;\middle|\; s\right) > \mathrm{P}\left(N \;\middle|\; s\right)$. Furthermore, the ratio $\mathrm{P}\left(M \;\middle|\; s\right) / \mathrm{P}\left(N \;\middle|\; s\right)$ expresses our confidence in the assertion that the sequence is more similar to those represented by the model M than a sequence drawn at random from the distribution represented by the null model. This is not equivalent to saying a sequence drawn at random from the database, since we can't usually have null models good enough to represent the unknown distribution the database is drawn from.

The probabilities $\mathrm{P}\left(M \;\middle|\; s\right)$ and $\mathrm{P}\left(N \;\middle|\; s\right)$ described above are typically very small, since there are a very large number of sequences that we want to give non-zero probabilities to. The ratios of the probabilities, on the other hand, can get very large, since the null model has to model an even larger set of sequences. To avoid having to write out or compute with these very large and very small numbers, we take logarithms. The *log-likelihood* of a sequence is $\log \mathrm{P}\left(s \;\middle|\; M\right)$ and our test that $\mathrm{P}\left(M \;\middle|\; s\right) > \mathrm{P}\left(N \;\middle|\; s\right)$ translates to $\log \mathrm{P}\left(M \;\middle|\; s\right) - \log \mathrm{P}\left(N \;\middle|\; s\right) > 0$. Let's use Bayes' Rule to rewrite this test:

$$\begin{aligned}\log \mathrm{P}\left(M \;\middle|\; s\right) - \log \mathrm{P}\left(N \;\middle|\; s\right) &= \log \frac{\mathrm{P}\left(M \;\middle|\; s\right)}{\mathrm{P}\left(N \;\middle|\; s\right)} \\ &= \log \frac{\mathrm{P}\left(s \;\middle|\; M\right)}{\mathrm{P}\left(s \;\middle|\; N\right)} \frac{P(M)}{P(N)} \\ &= \log P_M(s) - \log P_N(s) + \log \frac{P(M)}{P(N)} \; .\end{aligned}$$

The value $\log P_M(s) - \log P_N(s)$ is often referred to as the *score* of the model (though more properly it should be called the score relative to the null model N). Our test for something fitting the model can then be translated as requiring score $> \log P(N) - \log P(M)$. Furthermore, as described above, the amount by which the score exceeds the threshold expresses our confidence in the result.

This form of test, that a score be larger than some threshold, should be a familiar one, since almost all the recognition models used in computational biology fit this pattern [1]. The stochastic model approach tells us what setting the threshold means—we are making a statement of belief about our prior expectations. For example, if we expect some motif to be fairly rare, with maybe 10 occurrences out of 10,000,000 possibilities, then we would set $P(M)/P(N) = 10/10,000,000$ and our threshold at 19.93 bits (assuming we use base two logarithms, as most computer scientists do).

When we are doing a database search with a model, we score each sequence (or part of a sequence) with the model, and report any for which $\mathrm{P}\left(M \mid s\right) > \mathrm{P}\left(N \mid s\right)$, that is, ones whose score is above the threshold $\log P(N) - \log P(M)$. To simplify the setting of the threshold, most systems allow you to express how many hits E you expect. This number is divided by the number of different sequences or sequence parts scored to get an estimate for $P(M)/P(N)$, which can then be used to set the score threshold.[1]

Note: the search with the threshold set as above will find every sequence that is more probably from the model than the null model. Since our null model is often not a very good description of the set of sequences in the database, we sometimes want to set the threshold higher to increase our confidence in the results. For example, if we want only sequences that are significant at the 0.05 level (20 times more probably from the model than the null model), we need to raise the threshold by 4.3 bits. This is most conveniently done by reducing our expected number of hits E by a factor of 20.

Note that changing the expected number of hits E changes how many hits we actually get—if we expect more, we generally get more. Luckily, with good models the number of hits we get is not extremely sensitive to E. We usually get only a small change in the number of hits over a very wide range of E values, and so the exact setting of E is usually not too important. If you want to be very careful, you can do a search with a reasonable value (say E between 0.1 and 10), then repeat the search with E set to the number of hits found multiplied by the significance level desired. There is no absolutely correct way to set E, since it includes a statement of belief about the probabilities.

Quick review: This section has covered

- *Bayes' rule for conditional probabilities,*
- *turning recognition questions into discrimination questions,*
- *the meaning of a score (as the logarithm of a likelihood ratio),*
- *using discrimination tests to search a database,*
- *setting the score threshold for database search.*

2.3 Information theory—what is a bit?

The last section discussed choosing between two models (one of which is usually a null model). In this section we want to present a slightly different way of looking at the numbers so that we can apply other mathematical tools—those of information theory and data compression.

Information is a measure of how surprising something is. If we already know something, there is no information in getting the knowledge again. If we strongly suspect something, then confirmation contains a small amount of information. If we are very certain that something is not true, then find out that we were wrong, a large amount of information is conveyed by the new data.

[1] The BLAST program uses a different approach to reach essentially the same technique for more arbitrary scoring systems [2].

Technically, we measure information in a sequence (or set of sequences) relative to some model. The information content of a sequence s relative to a model M is $-\log P_M(s)$. If the logarithm is taken in base 2, then the information is said to be in *bits*.[2]

Since our notion of a model is a probability distribution over sequences, we can talk about the *entropy* of a model $H(M)$ as the weighted average information for all sequences, where the weight is the probability that the sequence is generated by the model:

$$H(M) = -\sum_s P_M(s) \log P_M(s) \ .$$

Again, the computation can be done with logarithms base 2 to get entropy in bits or with natural logarithms to get entropy in nats.

One sometimes sees the information content of a sequence or set of sequences referred to as the *encoding cost*. This comes from a theorem by Shannon, the fundamental theorem of information theory, that any encoding system for items drawn from some distribution must take at least as many bits on average as the entropy of that distribution. Data compression techniques consist of two parts: finding a model for the data, then choosing an encoder that assigns codes based on the estimated probabilities. The second part is not very interesting to us here, and so we are often somewhat sloppy, and refer to the information content of a sequence as its encoding cost, as if we had an optimal encoder. Note that choosing a model to minimize the encoding cost of a sequence is equivalent to choosing a model to maximize the sequence's probability.

The score that we discussed in Section 2.2 is just the difference in the information content of a sequence when computed using different models. We often talk about the score as the number of bits saved using the model M (with an implicit null model N). If the sequence fits the model, then the model provides us extra information, which we can use to encode the sequence in fewer bits. The score is measuring exactly how much extra information the model gives us. A negative score (where the sequence does not fit the model) tells us how many more bits we would need to encode the sequence if we insist that it comes from model M rather than N.

There is a very nice way to look at the discrimination test of the Section 2.2 that allows us to generalize the discrimination test to any number of models. Consider the picture in Figure 1. The two boxes correspond to our two possible models M and N. The edges from the start node represent a choice we have to make of which model to use. To encode a sequence, we first choose one of the two models, and encode that choice, then encode the sequence according to the model chosen. To get an encoding for the choice, we need an estimate of how often we will choose M and how often N. Let $P(M)$ be the probability with which we expect to choose M, and $P(N) = 1 - P(M)$ be the probability of choosing N. The cost of encoding our choice is $-\log P(M)$ if we choose M and $-\log P(N)$ if we choose N. The overall encoding cost for a sequence is either $-\log P(M) - \log P_M(s)$, if we choose M, or $-\log P(N) - \log P_N(s)$, if we choose N. If we choose whichever model gives the smaller encoding, we'll choose M when

$$-\log P(M) - \log P_M(s) < -\log P(N) - \log P_N(s) \ ,$$

which is exactly the same test as we had for a sequence matching model M in Section 2.2.

The generalization of this test to more than two models is now straightforward. We can build a composite model of many different models, as in Figure 2, and choose whichever model gives the lowest overall encoding cost. The probabilities for choosing the various models have to add up to one, since we have to choose a model in order to do the encoding. Note that once again the cost for choosing a particular model M_i is $-\log P(M_i) - \log P_{M_i}(s)$, so that our prior beliefs about the relative probabilities of the models affects which one we choose to minimize cost.

Finding a model that fits a given set of data (usually referred to as a *training set*) will be the subject of this afternoon's tutorial. For now, it suffices to know that we select a class of models, then adjust

[2] Some people, mainly physicists, insist on using natural logarithms. The resulting units of information are then sometimes called *nats* or *nits*. Since the SAM program was initially written by a physicist, it uses nats rather than bits.

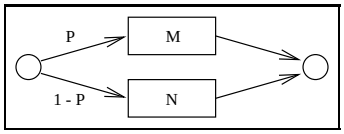

Figure 1: Two-part composite model, combining models M and N. To encode a sequence, we first choose one of the two models, and encode that choice, then encode the sequence according to the model chosen. To minimize the encoding cost for a sequence, we choose model M if the sequence fits the model (using the test of Section 2.2.

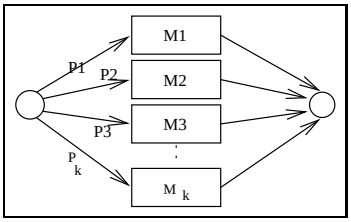

Figure 2: Multi-part composite model, combining several models. To encode a sequence, we first choose one of the models, encoding that choice, then encode the sequence according to the model chosen. Choosing the model to minimize the encoding cost gives us a clean way to do discrimination among many models.

parameters in an attempt to minimize the encoding cost for the sequences in the training set. For some types of models there are very simple techniques for doing this, but for other types the optimization problem is more complicated, and we have to use sophisticated optimization routines and still have no guarantee of convergence to a global optimum.

To make things a little more complicated for us, we don't really want a model that just encodes the training set well—we could get that by just memorizing the training set and checking if a sequence had been seen in the training set or not. What we really want is a model that generalizes the training set to similar sequences.

The structure of the model determines what sorts of generalizations are possible, but even without knowing the structure of a model, the entropy of a model measures how much generalization the model makes. A model with a low entropy must assign high probability to very few sequences making a very specific model with little generalization. A model with high entropy assigns somewhat lower probabilities to many more sequences, providing more generalization but less specificity. This afternoon we'll talk about ways of controlling the entropy of a model to get different degrees of specificity.

Quick review: This section has covered

- *the definition of information,*
- *the definition of entropy,*
- *composite models and their relationship with discrimination tests,*
- *the relationship between minimizing encoding cost and maximizing probability.*

2.4 Computable models—simplifying assumptions

So far all our models have been purely mathematical objects—black-box functions with no internal structure. Although we can learn a lot about how to use models that way, in order to do any real work we have to have models that are computable. Ideally, we would also like the models to have some sort of biological

significance, so that we can learn more about the sequences they model by examining the details of the model.

One type of model that is commonly used in data compression work is the *Markov model* (also called a *Markov chain*). In this model we assign a probability for each character based only on what the preceding few characters of the sequence were. The number of preceding characters is referred to as the *order* of the Markov model.

For example, if we are dealing with protein sequences and wanted an order-2 Markov model, we would have a table of 20×20 probabilities, $P[i,j,k]$, with the constraint that probabilities sum to one for any fixed context (i,j). That is, $\sum_k P[i,j,k] = 1$. To get the probability of a sequence of amino acids, we take the product of the probabilities of the individual amino acids. We have to do something special for the first two amino acids in the sequence, since we don't have a full context for them. One simple trick is to sum over all possible contexts: for the first position $P[k] = sum_{i,j} P[i,j,k]$ and for the second position $P[j,k] = sum_i P[i,j,k]$. If we have the sequence DNDNDG, we would assign it the probability $P(s) = P[D]P[D,N]P[D,N,D]P[N,D,N]P[D,N,D]P[N,D,G]$.[3]

As discussed in Section 2.2, the product of these probabilities can get very small, and so we use logarithms. If the probability tables $P[k]$, $P[j,k]$, and $P[i,j,k]$ are replaced by tables of the logarithms, computing the log of the probability of s is just the summation of 6 table lookups, making these simple Markov models very fast to compute. We can also train simple Markov models easily, since we can just count how often each triple occurs in the training set, and normalize to get the probabilities to sum to one in the last dimension. (Note: if you use a high-order Markov model, you might not have enough counts in any context to get a reliable estimate of the probabilities—we'll discuss strategies for estimating probabilities from too-small samples this afternoon.)

One common use of a simple Markov model is for defining the null model to use in comparisons. Many of the search program have an implicit assumption of a zero-order Markov model as their null model (the probabilities of the letters are the same in every position, independent of the context). Higher-order models have been used for compositional models (for example, for recognizing introns and exons by using 5th-order Markov models on the bases [4] and for recognizing repeated regions in DNA [17]).

The beauty of simple Markov models is that they provide a very simple computational technique both for using the model and for training the model. This simplicity comes from defining specific easily determined contexts that allow us to compute the probabilities of the amino acids, and an assumption that the probabilities don't depend on anything except these contexts.

Unfortunately, the simple Markov models provide little insight into the structure of the sequences they model, and the generalizations they make are not always the most appealing biologically. The next section will describe a related type of model, the *hidden Markov model*, which still has the concept of distinct contexts for determining the probabilities of amino acids, but which provides a much more transparent description of sequences.

Quick review: This section has covered

- *simple Markov models,*
- *computing probabilities by summing log probabilities,*
- *table lookup of log probabilities,*
- *zero-order Markov models as null models.*

[3] Technically, this is not $P(s)$, but $\mathrm{P}\left(s \;\middle|\; \text{length}(s) = 6\right)$. To get $P(s)$ we have to also include in the product the probability that the length of s is 6.

3 Overview of HMM Architecture

Leslie Grate (leslie@cse.ucsc.edu)

This section covers realizing HMMs as state machines, the reasoning behind the choice of the linear HMM structure and comparisons to multiple alignments and profiles.

3.1 State machine visualization of Markov models

The simple Markov model introduced in Section 2.4 computed the probability of a sequence as the product of separately estimated probabilities for each residue or base in the sequence. A hidden Markov model does essentially the same thing, but provides a different way of computing the probabilities of the individual letters of the sequence. Instead of relying on the previous k letters to determine what table to look up letter probabilities in, hidden Markov models have a set of *states*, each of which has a table of letter probabilities. Which state the hidden Markov model will use for a given letter is not immediately obvious from looking at the sequence—what state you are in for each letter is what is "hidden" in a hidden Markov model.

In addition to states, a hidden Markov model needs *edges* or *transitions* between the states. There may be multiple edges out of a state, in which case we assign a probability to each of the edges out of that state. These branch probabilities are taken into account when we compute the probability of a sequence.

To compute the probability for a sequence, we have to consider a path through the states of the HMM. Each time the path goes through a state, one letter from the sequence is "used up", and the probability for that letter is computed from the table for that state. We also have to pay for the freedom of choosing different paths—the probability of the sequence is the product of the letter probabilities with the probabilities of the edges we have used. Mathematically, we can express this computation as

$$\mathrm{P}\left(s \;\middle|\; M, \text{path}\right) = \prod_i \mathrm{P}\left(s_i \;\middle|\; \text{path state}_i\right) \prod_i P(\text{edge from path state } i \text{ to path state } i+1) \ .$$

When using HMMs, there are two standard approaches for choosing paths: either we take the single path which has the highest probability (known as the Viterbi path)

$$\text{Viterbi probability}(s|M) = \max_p \mathrm{P}\left(s \;\middle|\; M, p\right) \ ,$$

or we add up the probabilities over all paths (known as the Baum-Welch method):

$$\mathrm{P}\left(s \;\middle|\; M\right) = \sum_p \mathrm{P}\left(s \;\middle|\; M, p\right) \ .$$

For a more thorough introduction to HMMs, please refer to L. Rabiner's paper, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition" in the appendix [22].

3.2 Linear HMMs for Biological Sequences

A general HMM is an arbitrary graph of states and edges, and so can be very difficult for a biologist to interpret. To make HMMs more comprehensible, we usually restrict the structure to a simple linear arrangement.

The linear arrangement is motivated by the common biological modeling technique of aligning sequences to a consensus sequence. In the linear HMM we will have *match* states corresponding to the letters of the consensus sequence. In addition we will have *insert* states corresponding to the positions between the letters of the consensus sequence where gaps can be opened up and letters inserted. To get a complete analogy to the alignment model, we also allow special *delete* states that do not "use up" any characters of the sequence

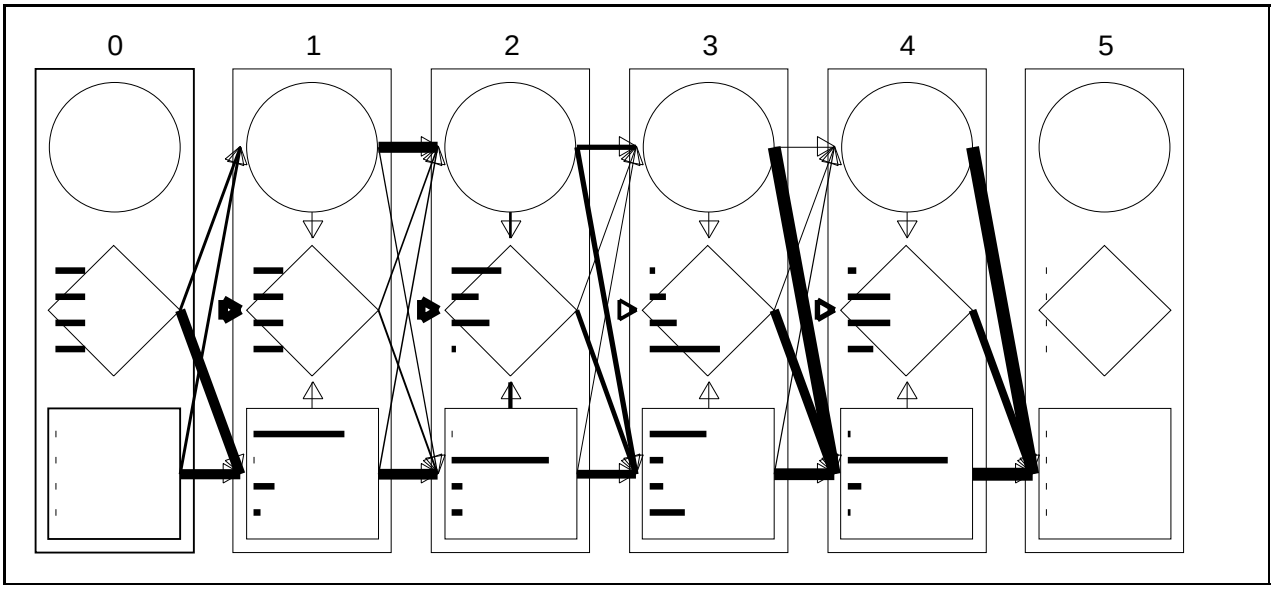

Figure 3: A small HMM model as displayed in Hmmedit. The Begin node is node 0, at left, and the end is node 5 on the right. The thickness of an edge indicates its probability. The alphabet is 4 letters as can be seen from the probability distribution bar charts inside of each Match (square) and Insert (diamond) state. The Delete state (circle) does not consume any letters, hence has no distribution.

whose probability we are computing, but that allow paths to skip match states (just as alignments may delete parts of the consensus sequence).

Figure 3 gives an example of a small model for RNA sequences. The delete, match, and insert states are grouped into *nodes*. Each node has 3 states:

- Match state. These correspond to "good" columns of a multiple alignment and form the core of the model. Each contains an independent distribution over the letters in the alphabet.
- Insert state. These states have self loops (edges from the state back to the same state), that allow for arbitrary length insertions of letters. Each contains an independent distribution over the letters in the alphabet.
- Delete state. These states are like Match states, but no letter from the sequence is used up. These handle cases where a letter has been deleted relative to the consensus.

Each state has three incoming transitions and three outgoing transitions. The outgoing transitions are to Insert in the same node and Match and Delete of the next node. Combining all the edges for a node, we get three internal edges (to the Insert state), six external edges coming in (to Match and Delete), and six external edges going out (to Match and Delete of hte next node). These allowed transitions define the possible paths through the HMM. Two special nodes are added to the beginning and end (Begin and End), and we only consider paths from Begin to End.

For instance, when in a Match or Delete state, we can only exit to 3 other states:

- The next node's Match state. In most HMMs, this will be the most probable transition.
- The next node's Delete state. This is used to skip the next letter relative to the consensus.
- The current node's Insert state. This allows insertions of letters relative to the consensus.

Likewise, the insert state can exit to only 3 states: itself (forming a self loop), and the next node's Match or Delete states.

One of the nice properties of this particular HMM is that exactly the same dynamic programming techniques that are used for sequence-sequence alignment can be used for finding the Viterbi path through the HMM and computing the corresponding probability.

One major disadvantage of HMMs is that they cannot directly model long-distance interactions such as base pairing in RNA. For this, more complex models such as *stochastic context-free grammars* [23, 8, 11] are needed.

3.3 Multiple Alignments, Profiles, and HMMs

Several modeling techniques that are conceptually quite similar to linear HMMs have been used by biologists, including multiple alignments, profiles, and generalized profiles. A good overview of these techniques can be found in [5].

A multiple alignment is not really a modeling technique, but a way of presenting a group of related sequences to highlight the parts they have in common. A multiple alignment is easily created from a linear HMM by finding the Viterbi path in the HMM for each sequence. Each match state of the HMM creates a column in the multiple alignment. The letters that come from insert states on the Viterbi path can either be hidden (as is done in HSSP files) or shown in lower case to indicate that they are not necessarily aligned with other letters from the same insert state (as we do in the output from `align2model`).

Converting a multiple alignment to a linear HMM is fairly straightforward— a node is created for each column in which most sequences have an aligned residue. Columns with many missing letters are mapped to insert states between match states. The probabilities used for the letters in Match states are computed from the letters seen in the multiple alignment, using the techniques we'll describe later (Section 5.1). For Insert states, there are generally not enough examples to compute probabilities reliably from the observed residues, and so an appropriate background frequency is used for Insert states.

Setting the transition probabilities is equivalent to setting gap penalties in sequence alignment and remains more of an art than a science. The program `modelfromalign` in the SAM suite of tools creates HMMs from multiple alignments automatically, with fairly reasonable default transition probabilities.

A profile is a summary of the alignment columns of a multiple alignment, giving the probability for each letter in each column. This is very similar to the linear HMM we have described, but without the possibility of having insertions or deletions. A profile can be converted to an HMM in the same way that multiple alignments are, but we have not provided any explicit program for this conversion, since the conversion from multiple alignments provided by `modelfromalign` is more versatile.

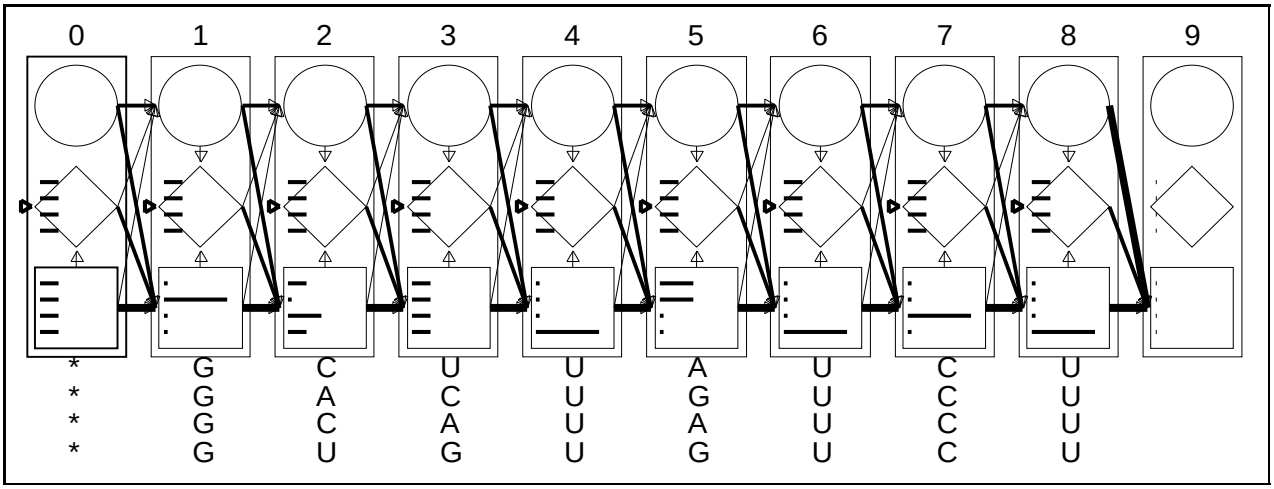

Figure 4: Hmmedit display of a small model made from the alignment. The ordering of the bar charts is AGCU, A at the top. Note the action of the built-in regularizer. The regularizer has made all letters in all columns at least slightly possible, even for pure columns.

4 Basic Uses of Hidden Markov Models

Leslie Grate (leslie@cse.ucsc.edu)

This section discusses what can be done with the HMM methodology. HMMs can be built from aligned and unaligned sequences, sequences can be aligned to a HMM, and databases can be searched for matches to the HMM.

4.1 Building HMMs from existing alignments

The program `modelfromalign` creates a HMM from a group of aligned sequences. The fact that the linear HMM structure maps alignment columns to nodes makes this process straightforward.

As in Figure 4, the program first creates a node for each column in the alignment. The transition probabilities and insert distributions are set to defaults, but the match state distribution is computed from the letters in the corresponding columns.

An important issue arises in the process of computing match state distributions from a single (possibly very small) column of letters. It is likely that some letters will not occur in a given column, hence have zero counts, which directly translates to zero probability of occurring based on the raw frequencies. While this might be appropriate for some columns, most of the time we should not totally exclude the possibility of observation of letters: to do so would limit the ability of the model to generalize.

Fixing these zero probabilities involves the process of *regularization* which will be covered in Section 5.1. SAM has a built-in regularizer that users can override.

4.2 Aligning Sequences to HMMs

The program `align2model` creates a multiple alignment by aligning sequences to a HMM. Note that each sequence is aligned individually to the model.

Alignment of a sequence to a model means that each letter in the sequence is associated with a match or insert state in the model. Two 5-character sequences, A and B, are shown in a 4-state model in Figure 5, along with the corresponding alignment between the sequences.

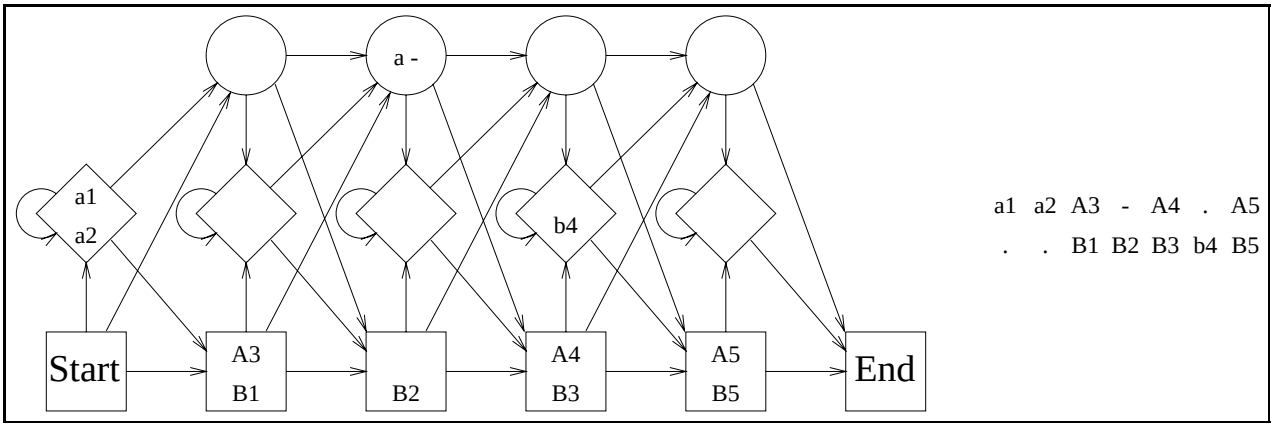

Figure 5: An example of two sequences whose characters are matches to states in an HMM, and the corresponding alignment.

One can specify such an alignment by giving the corresponding sequence of states with the restriction that the transition lines in the figure must be followed. For example, to match a letter to the first match state (m_1) and the next letter to the third match state (m_3) can only be done by using the intermediate delete state (d_2), so that part of the alignment can be written as $m_1 d_2 m_3$. In HMM terminology such an alignment of a sequence to a model is called a *path* through the model. A sequence can be aligned to a model in many different ways, just as in sequence-to-sequence or sequence-to-profile alignments. Each alignment of a given sequence to the model is scored by using the probability parameters of the model. The best-scoring path is usually reported as the "correct" alignment.

4.3 Scoring and Database Discrimination using HMMs

Once you have a model, any sequence can be scored against it by computing the probability that the sequence was generated by that model. An interpretation of this form of score is that it tells you how "far away" a sequence is from the model.

However, this is not the final word on scoring because these types of scores have a strong dependence on sequence length: there is not a single "good" method for a scoring function.

The SAM suite program `hmmscore` has a few different scoring methods the user can choose from (refer to the SAM manual [15] for details). Either the Viterbi ("best path") or Baum-Welch ("sum over all paths") (see Section 3) method is used to compute the probability $P(s|M)$ that a given sequence was generated by a given model. The negative logarithm of this value is termed the *NLL cost*. A small cost corresponds to a high probability, and so a good match to the model.

SAM and HMMER also score each sequence against a user-adjustable null model. The difference between the NLL score for the model and the NLL score of the null model is termed the *log-odds score*. SAM reports NLL−Null costs, for which a negative value means a good match to the model.

Discrimination involves choosing a score threshold which gives the best separation between sequences that are in the family you are attempting to model and those that are not. Computing the scores of many sequences from a data base and plotting them in histogram form is a useful method of visualizing the results. The program `makehist` creates a `gnuplot` script for making histograms of `hmmscore` results (the `.dist` files). Examples of these histograms are shown in Figure 6.

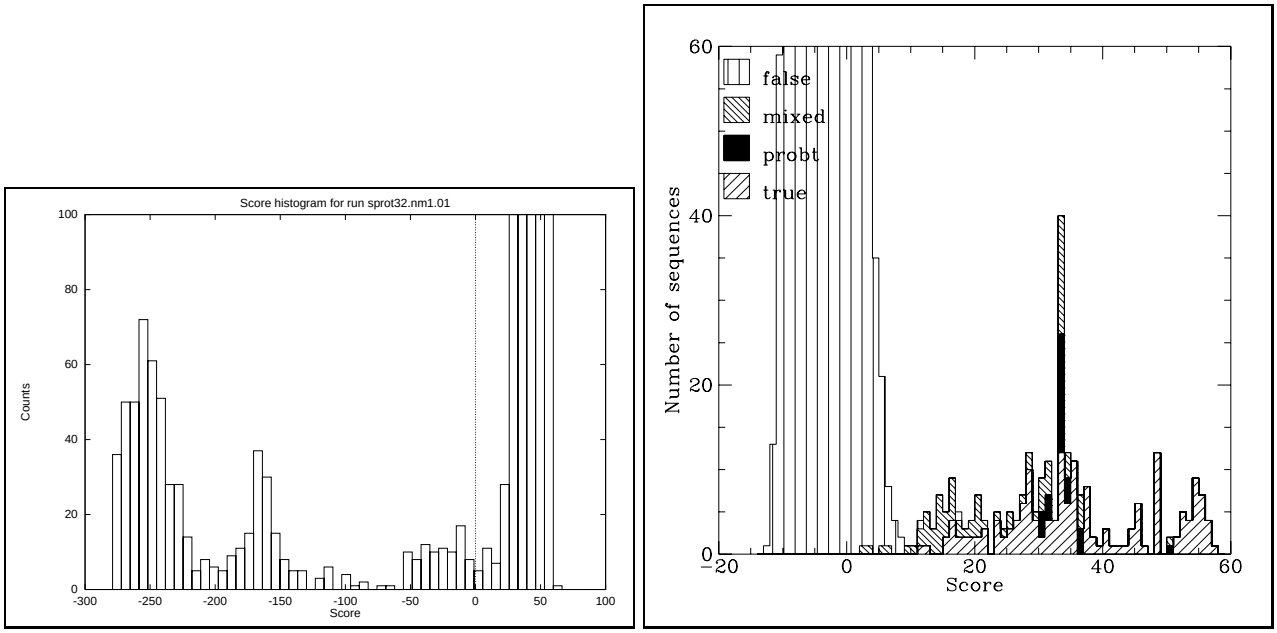

Figure 6: Histograms showing results of discrimination tests using HMMs. The left plot is of a SAM model of the globins made using **`makehist`**. Negative scores mean the model fits a sequence well, so the more negative the better. The right plot is a HMMER model for the efhands made with a special plot program. Here positive scores are better. Regardless of the sign of the score, note that there is a valley between "good" and "bad" fits to the model. *Courtesy of Christian Barrett.*

5 Building an HMM from training data

Kimmen Sjölander (kimmen@cse.ucsc.edu)

Stochastic models for proteins are objects, like profiles, that capture the statistics defining a protein family or domain. Along with parameters expressing the expected amino acids at each position in the molecule or domain, and possibly other parameters as well, a stochastic model will have a scoring function (as described in Section 2) for sequences with respect to the model. Models are built to accomplish a particular task; thus the method used to build the model must be developed with that task in mind. In this section, we will focus primarily on how to build HMMs which are effective at recognizing homologs in the sequence databases which may have low primary residue identity.

The effectiveness of a model in database search is tied closely to the accuracy of the parameters of the model. These parameters must fit both the data seen (the training data) and the remote homologs—sequences which are related by phylogeny and structure to those used in training, but which have low primary residue identity and are thus difficult to recognize using general profile methods. The parameters must excel at conflicting tasks: they must give high probability to sequences seen (specificity), but also give probability to sequences not seen but which are homologous (sensitivity). Along the way, they must identify as *not* belonging to the set all the other sequences.

The complexity of the model also influences the kind of training data needed. Generally, the more parameters in the model, the more data you need to train it effectively. In early work modeling protein families and domains using HMMs [12], we found this reliance on sufficient data to be pronounced, such that many (upwards of 100) training sequences from the protein family or domain of interest were generally required in order to obtain reasonable results. For these rather complex models, the more sequences available for training, and the more variation among the sequences, the better. However, sheer quantity of data is not sufficient, since alignments of virtual copies of the same sequence with only a few positions mutated give almost no more information than a single sequence.

Unfortunately, this kind of data is not always available. Most protein families contain a far smaller number of sequences. In the extreme, we may have a single sequence we are trying to characterize. One of the first tasks, in fact, when a new protein is sequenced, is to find all possible homologs in the protein sequence databases. Any homologs identified will contain information which assists the analysis of the unknown protein. If we are lucky, the annotation of the homolog will include information about its function or structure. But even without such annotations, the multiple alignment of all the sequences together will reveal much about the important positions in the protein: which are conserved and which are variable, which give clues for active sites, and so on.

Moreover, the problem of finding homologous sequences, close or remote, is not limited to the case where one has a single protein. One may have several sequences available for a given family, but expect that other family members exist in the databases, and want to locate these putative members.

Various methods have been developed to address the problem of recognizing homologous sequences. Among the first developed were those that were based on string-matching algorithms. These methods were developed based on the observation that when two sequences share at least 25% residue identity, and each is at least 80 residues in length, then the two sequences are homologous[24, 6]. These methods attempt to find a way of way of maximizing the number of matches between two sequences. Examples of this group are algorithms that search for the longest common subsequence, or exact matches to motifs of particular lengths.

In contrast with homology determination by residue identity, stochastic models use a very different technique to determine whether two sequences share a common structure. During database search with these models, each sequence in the database is assigned a score (or, negatively, a cost). This score is computed by adding the scores at each position, as described in Section 2. For instance, a typical cost for aligning residue a at position i, is $-\log \mathrm{P}(a \mid \text{position } i)$, where the base of the logarithm is arbitrary. As we described in Section 2.2, a sequence is identified as a match to the model if the score of the sequence is above a cutoff (or, negatively, if the cost of the sequence with respect to the model is below a cutoff). In

SAM scoring, we report the cost for a sequence, thus smaller values indicate better matches. In HMMER scoring, scores are reported, with higher values indicating better matches.

The naive approach for setting the amino acid probabilities to be the fraction of times each amino acid was seen, is inadvisable under this kind of scoring system. The reason is that allowing zero probabilities at positions gives an infinite penalty to sequences having the zero-probability residues at those positions. Even if a sequence is homologous to those used in training the model, a single mismatch at such a position would render that sequence unrecognizable by the model.

If there are particular positions where we know (due to additional external information) that a particular position *must* contain a residue with probability 1, then we can artificially set that probability. But as a general rule, the observed frequencies over the amino acids will generate many zero probabilities, and should be avoided in stochastic models which use this kind of scoring scheme[4].

5.1 Regularizer methods

We have included in the Appendix two papers which go into some detail describing the relative performances of substitution matrices, pseudocount methods, and Dirichlet mixture priors, for regularizing amino acid distributions (see the Technical Reports on Dirichlet mixtures (Sjölander et al) and Regularizers (Karplus)). In this section, we simply give an overview of some of these differences.

Substitution matrices

Several approaches have been proposed to solve the problem of *regularizing*, or generalizing amino acid distributions in positions to be able to give probability to amino acids of similar types to those seen. Undoubtedly the most popular of these methods involves the use of *substitution matrices* [27, 13, 1, 21]. These matrices are derived from alignments of homologous proteins, which reveal certain substitutions to be more likely than others among amino acids. For instance, isoleucine and valine are often found in the same position in homologous molecules, but neither is particularly likely to substitute for glutamate. From this observation, substitution matrices that formalized the cost of substituting one amino acid for another were created. These substitution matrices have been used with good results to generalize the observed amino acids in a protein sequence to create a profile which performs well at database search for homologous sequences.

There are two drawbacks associated with the use of substitution matrices. First, each amino acid has a fixed substitution probability with respect to every other amino acid[5]. However, an amino acid seen in one context, for instance, in a position that is functionally conserved, will have different substitution probabilities than the same amino acid seen in another context, where there are few functional or structural constraints. Second, only the relative frequency of amino acids is considered, while the actual number observed is ignored.

Pseudocount methods

Pseudocount[6] methods were designed to handle the problems described above: avoiding zero probabilities, and adding some generalization capacity to the estimated probabilities. In these methods, probability estimates are obtained in a two-step process. First, pseudocounts z_i for each possible symbol i in the alphabet (amino acids, in the case of proteins) are added to the number of observed counts n_i in the data for that symbol. Then, the total counts for each symbol (observed plus pseudocounts) are divided by the total counts over all symbols (observed plus pseudocounts), to obtain the probability of each symbol. That is, the

[4] This kind of problem does not exist in methods that search databases for remote homologs using the fraction of identical residues to score sequences. In these methods, all that is crucial is that the sequence being matched in the database be alignable to the target sequence (the sequence used to search the database) in such a way that 25% of the residues are identical. It is expected that most positions won't have the same residue in both the target and in the aligned sequence.

[5] In the work by Overington *et al*, there has been an attempt to use structural information to create environment-specific substitution matrices. However, this assumes that such structural information is available, and this is not always the case in our work.

[6] The word "pseudocounts" is chosen to convey the artificial nature of these added counts.

expected probability of a letter i is

$$\hat{p}_i = \frac{n_i + z_i}{|n| + |z|}$$

Not surprisingly, such pseudocount methods are limited in their effectiveness. Variants of these methods have been developed that attempt to take into account some of the additional information in the column being regularized. Such methods are called *data-dependent pseudocounts* (for example, [25]).

Dirichlet mixtures

The dependency on sufficient data to estimate the parameters of a HMM prompted us to look for ways to include prior information over amino acid distributions into our model-building process. We used the excellent work by Duda and Hart on estimating mixtures of Gaussian densities [7] as a template to estimate mixtures of Dirichlet densities. We chose Dirichlet densities because they had several nice mathematical properties which made them very appealing.

Dirichlet densities are densities on probability distributions. In the case of a Dirichlet density over amino acid distributions, these densities give the likelihood of every point $\vec{p} = p_1 \ldots p_{20}$. In our work, we estimate these densities from columns extracted from thousands of multiple alignments[7], and the densities estimated come to represent the probabilities of different amino acid distributions within the context of the database used to train the density's parameters.

Dirichlet mixtures are, quite simply, mixtures of Dirichlet densities, which jointly assign probabilities to all distributions of the symbol alphabet in question. For instance, a distribution over amino acids that gave tryptophan probability 0.5, and glycine probability 0.5, would probably be given low probability by a mixture of densities estimated on alignment columns. We simply don't see too many distributions like that. However, since the alignments we used to estimate these densities were of fairly close homologs, these mixtures give pure distributions and mixtures of amino acids sharing common physico-chemical attributes fairly high probability.

Each density is defined by a set of parameters, which we refer to as $\vec{\alpha}$. These parameters define the probability given each distribution of amino acids by the density. In the case of a mixture, each density is a *component* of the mixture, and the probability of that component within the mixture is referred to as the *mixture coefficient*. In our work, we refer to this prior probability as q.

The Dirichlet mixtures are used to add *data-dependent pseudocounts* from each component of the mixture to the observed counts. The total counts for each symbol are then divided by the total counts over all symbols (as described in the previous section), to form the posterior probability of each symbol in the alphabet given the observed counts and the Dirichlet mixture prior.

These mixtures have been shown in various experiments [26, 14, 3] to be more effective at reducing the numbers of false positives and false negatives in database discrimination tests. A theoretical rationale for these results is given in [18, 19], and is included in the Appendix.

The SAM suite of HMM programs incorporates the use of one nine-component Dirichlet mixture prior which we have found to be the most effective. HMMer includes both mixture priors and structure-based mixture priors [10].

5.2 Weighting schemes

It's undoubtedly not news to anyone attending this tutorial that alignments of protein or DNA sequences often contain a large number of very close homologs or even exact duplicates. The first problem with this kind of skewed data is that it directly conflicts with the assumption of independence among the data, which lies at the heart of the stochastic model typically used.[8]

Unfortunately, the problem is not only theoretic. Skewed data can restrict the capabilities of a model estimated from the data in several ways:

[7] Alignments used have come from the BLOCKS and HSSP databases.

[8] Actually, whenever we have aligned sequences we are not going to have independent data; sequences are aligned because they are related by function or structure, and are, hence, clearly correlated.

1. Poor generalization capacity.

   When one subfamily in the data is well represented, but the others are less so, the statistics in the dominant subfamily overwhelm the statistics in the other subfamilies. This results in models that perform poorly in database discrimination or search tasks; they recognize sequences which are similar to the dominant subfamily, but fail to recognize sequences which are similar to the less-populated subfamilies used in training. They may even fail to recognize sequences used in training, if these sequences are in the smaller subfamily.

   If the method used to compute the expected amino acids at each position takes into account the actual numbers of observations at each position (as is the case with Dirichlet mixtures), then an attempt must also be made to estimate the actual number of independent counts in the data. Otherwise, models built using large numbers of highly similar sequences will have probability estimates for the amino acids at each position which are sharply peaked around the residues seen. In an alignment containing many columns, such high requirements for matching at each position the residues seen in the training data will make these models unable to recognize even fairly close homologs.

2. Difficulty distinguishing important, or conserved, positions, from the less important positions.

   The same type of situation gives rise to another problem. Without an estimate of the actual number of independent counts in the data, it is hard to differentiate a truly conserved position from a position which is not conserved. On the other hand, if one has a fairly diverse and large set of sequences, any positions which are conserved will stand out.

3. Difficulty building alignments from skewed data.

   Similar problems can arise when constructing an alignment in a nearest-neighbor approach, if the training data contains a dominant subfamily with high residue identity. By the time any outliers are included, it may be difficult to align them properly using the expected amino acids estimated from the dominant subfamily.[9]

Because of this, methods have been developed that attempt to compensate for bias among sequences. Such methods are called *sequence weighting schemes*, or simply *weighting schemes*. Most of these methods allot smaller weights to sequences in the dominant set(s), and larger weights to outliers, in an attempt to even up the playing field. Increasing the weight of outliers is not without risk, however, as spurious inclusions in the data can result in models that give too much weight to statistics from spurious members, to the detriment of the effectiveness of the model in representing the true members.

As noted above, a weighting scheme on sequences must take into account the method which will be used to compute the distribution of amino acids used for scoring. If the method incorporates the actual number of observations, it will be crucial to have the total weight allotted the sequences be tuned to the goal of the model. If the distribution-estimation procedure is not affected by the total weight, then relative weights alone are sufficient. Since, historically, profile methods have used substitution matrices in computing the expecting amino acids (see Section 5.1), sequence weighting schemes that ignore the total weight are sufficient.

However, our preferred method to compute the expected amino acids, Dirichlet mixture priors, is designed to take the number of observations into account. In the absence of data, our method relies heavily on prior information for estimating distributions. But when much data is available, we want to believe the evidence, and our amino acid estimates will converge to the actual frequencies observed. In our case, then, we need to pick a weighting scheme that also carefully tunes the total weight allotted the sequences.

For instance, lowering the total weight in a set of aligned sequences produces more diffuse models with greater generalization and less specificity. The opposite occurs when the total weight is increased. Since the specificity (or generality) of the parameters is mirrored in the information content of the model (see

[9] As described earlier, this is more of a problem with methods for computing expected amino acids which take the number of observations into account.

Section 2.3), we can use this measure to adjust our models so that they are suited to remote homolog search for particular databases.

The weighting schemes developed by Kevin Karplus (unpublished) provide the user with a direct control over the diffuseness of the model created using them. The user specifies how many bits per column on average should be saved relative to using just the background frequencies, and the total weight of the sequences is adjusted until the Dirichlet mixture provides the right amount of generalization for the given multiple alignment.

Although modifying the total weight for training sequences is not the means employed to obtain this effect, the various PAM matrices are designed with the same goal in mind. In fact, Altschul's paper gives a table for translating the various PAM matrices into bits saved per column [1]. A PAM distance of 120 is a savings of about 1 bit per column and a PAM distance of 240 is a savings of about 0.5 bits per column.

There are two ways one can pick the savings per column:

- One can specify the savings per column based on the PAM distance one wants to generalize the model to, using the table in Altschul's paper. Figure 7 shows the translation between bits saved and the Blosum or PAM matrix number. Some researchers prefer to think in terms of percent residue identity or residue difference—Figure 8 provides a way of translating between percent difference and PAM distance.

- One can set the savings as low as possible while still saving enough to find significant hits in the database. For example, let's say we want to find sequences of length 65 or more, in a database of 50 million residues, with significance 0.01. This means that our models must save at least $\log_2 50\,000\,000 - \log_2 0.01$ bits (about 32.2). That means we need to save about 0.5 bits per column on the average. If we try to generalize further than that, we need longer matches to compensate for the lower score per column (we can go to 0.33 bits per column, if we are willing to demand up to 100 residues). If we are looking for a short motif (say 20 residues), we cannot generalize nearly as much without losing statistical significance.

  In practice, we often find shorter sequences than the above calculation implies. Since adjacent residues are not independent, a series of good matches in a row can make significant savings in less than the length one would expect for a random sequence from the distribution implied by the model.

Quick review: This section has covered how to build HMMs so that they are effective at recognizing homologs in the sequence databases, including

- *The kind and quantity of training data needed.*
- *The need for incorporating prior information.*
- *The need for and use of sequence weighting schemes.*
- *Different techniques for* regularizing *model parameters, especially those representing amino acid distributions.*

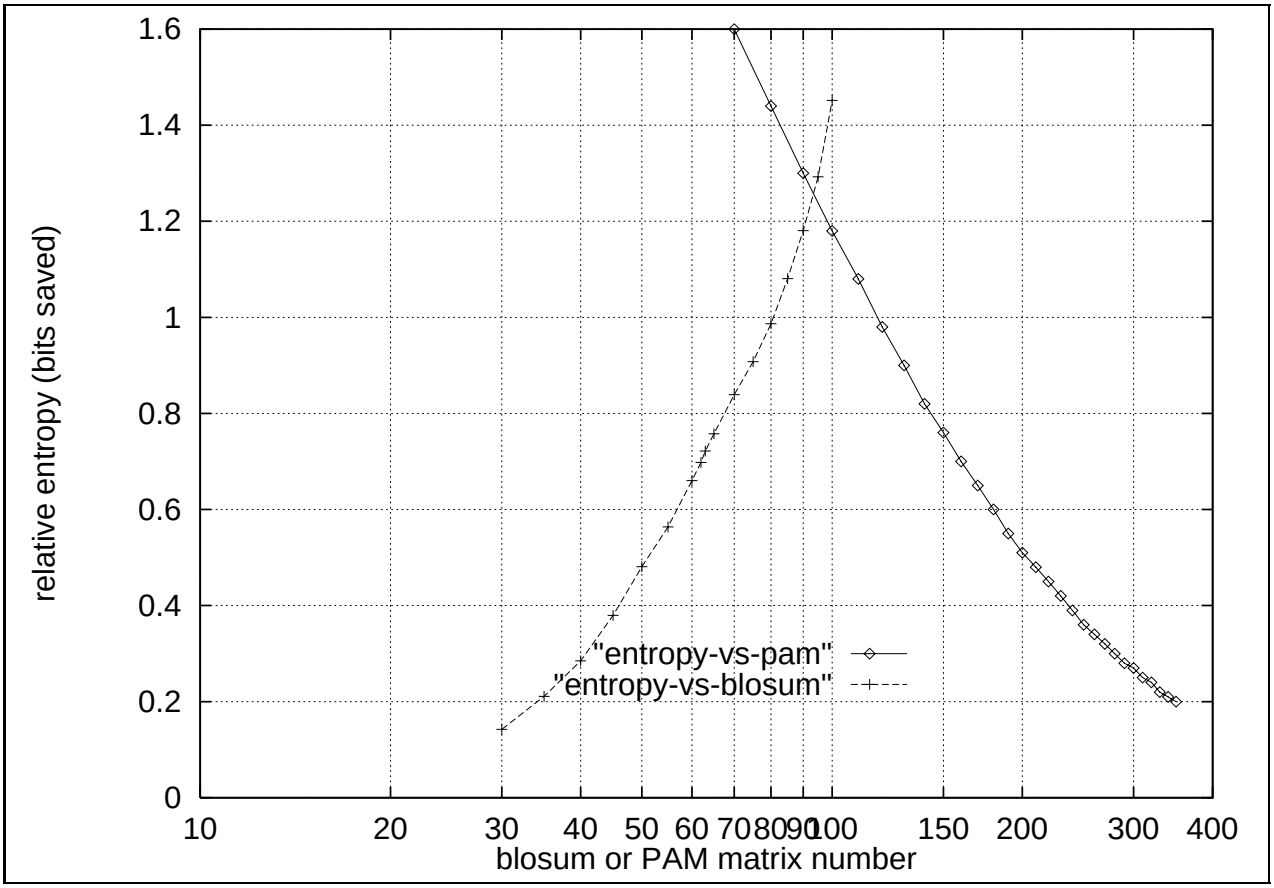

Figure 7: Graph showing the translation between average bits saved per position and Blosum matrix number or PAM matrix number. The PAM curve is from Altschul [1], and the BLOSUM curve is from the Henikoffs' program (version 5.0 of the Blocks database).

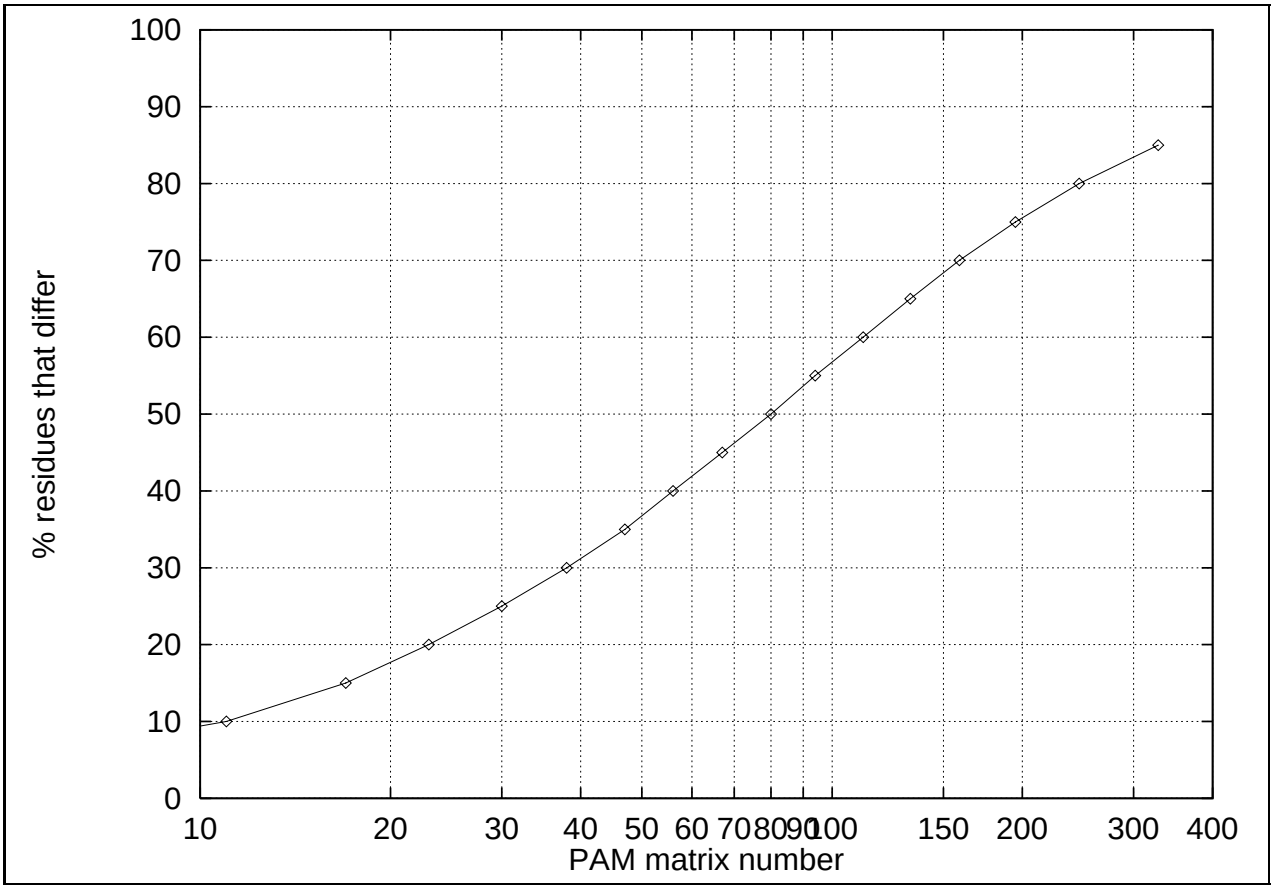

Figure 8: Graph showing the approximate number of different residues expected with different PAM matrices. The PAM matrix is put on the x-axis with the same scale as Figure 7 so that one can translate from percent residue difference to PAM distance to entropy visually. Numbers taken from `http://www.lmb.uni-muenchen.de/groups/bioinformatics/ch4/matrices.html`

6 Advanced Uses of Hidden Markov Models

6.1 Building multiple alignments from unaligned sequences

Kevin Karplus (karplus@cse.ucsc.edu)

This section will describe one method that we have used to build adequate hidden Markov models starting with just a single target sequence. The method presented here was developed for making models from the target sequences of the CASP2 protein-structure prediction contest, and is still being improved. It is intended for use by computer scientists who have little knowledge of proteins, and so has been made as automatic as possible. The `Makefile` used to create the examples for this section is included in the appendix, as are the alignments produced.

As with any model building, fully automatic techniques do not always provide the best models—a competent molecular biologist who has additional biological information about a sequence can often improve on them. Since the automatic techniques rely heavily on constructing and learning from multiple alignments, it is fairly easy for a biologist to introduce his or her knowledge by modifying the automatically produced alignments, or by editing the resulting model to adjust the importance of different columns.

For the contest, we do not need a sharp discrimination between the sequences that match the model and those that do not—indeed we want a model that matches as distant a homolog as we can find, while still having high enough scores to get significant hits in a large database. Ideally, we would like to generalize the models enough to find a significant hit in a database of known structures, but we don't expect to be able to do this.[10] The problem of finding remote homologs, whether for structure prediction, insight into the function, or determining which residues might be productively modified in wet-lab work, is a fairly common one.

In this section, we'll construct both sharply discriminating models whose purpose is to align close homologs of the target sequence and a more diffuse model whose purpose is to find more remote homologs. For the examples, we'll use target t0005 from the CASP2 contest, the C-terminal domain of human gamma fibrinogen.

You can build a model directly from a single sequence using the program `modelfromalign`. The resulting model has one state for each letter in the target sequence, heavily weighted toward matching the letter in the target. SAM does not currently support using substitution matrices to set the probabilities of the letters in this model, but it does allow the use of Dirichlet mixtures, which produce very similar probabilities given a single sequence.

Demonstrate `modelfromalign`*, show resulting model with hmmedit.*

One can use the model constructed from the target sequence to search a database for homologs, but this is a rather inefficient use of resources, since this initial model is not really any better for searching than the faster techniques used by BLAST [2]. A better technique for finding the close homologs is to use BLAST on the SwissProt or NR database to get a reasonably large set of homologous sequences quickly. Even faster is to use Entrez to get the "protein neighbors" of a sequence, and download the entire set as a FASTA file. Note that these two methods can produce somewhat different sets of sequences, and it is sometimes worth merging the results of both methods to create a larger, more diverse training set.

In general, having more sequence data produces better models, but the model-building process can get confused if there are too many copies of some of the data, giving the extra copies too much importance. Of course, sequence weighting can help reduce the effect of duplication. Having extra data also slows down the model building process (which is generally linear in the number of sequences). One technique that is both efficient and fairly effective is to build a model from a small number of sequences (say the BLAST hits in SwissProt) then retrain it on a larger set (say the BLAST hits in NR or the Entrez neighbors). For the examples here, we'll build models first using just 18 sequences (the BLAST hits in SwissProt), then retrain using just under 100 sequences (the neighbors identified by Entrez).

[10] For more details on what we *do* expect to be able to do, wait to see what we have for the CASP2 contest.

Worldwide web demo here: take a target sequence, use BLAST to find homologs, and use the link to Entrez to get the neighbors and download them.

Once we have a set of homologous sequences, we have some decisions to make about the structure of the model: Do we want the model to preserve the one-to-one relationship with the target sequence? Do we want to construct a more general model for the whole family, possibly discarding some pieces of the target sequence?

If we just want a generic model for the whole class, we can use SAM's `buildmodel` program with default parameters. This program then builds a number of random models with lengths around the average length of the sequences. For each model, the sequences are aligned to the model and the model is updated based on that alignment. Noise is added to the observed counts of amino acids and transitions, so that the model space is searched a little more broadly. The `buildmodel` program then selects the best of the resulting models and tries to modify it with *model surgery*, which cuts out nodes of the model that aren't used often and adds nodes where inserts were common. After surgery, the modified model is retrained. The training/surgery loop can be repeated several times.

Running `buildmodel` *with the default parameters is a little slow for these long sequences, and so will not be demonstrated live.*

The appendix includes two files that result from building a model with default parameters:

`t5_default.stat` which shows the progress of the model building, and

`t5_default.pretty` which shows the resulting alignment.

Note that the entire sequence has been modeled, not just the C-terminal domain whose structure we want to predict. Part of the region we're interested in (between alignment columns 312 and 314) has been shoved into an insertion—despite obvious homologies for the first 5 sequences. Also the large insertions in five sequences between positions 309 and 310 make the section before 309 grossly misaligned—there is a much better alignment hidden in the insertion.

In the `t5_default.stat` file, we can see the training method gradually improving the model, then losing much of the gain when it does model surgery, and almost recovering by retraining. The problem isn't the model surgery, but the noise that is added to the model after surgery. The current default adds about 10% of the noise used in the initial model, that is, about the amount of noise in the 10th iteration of retraining. It is possible to reduce the amount of noise added, keeping the training process from losing so much information.

The default training method described above tends to find and correctly align the more highly conserved blocks of the family, but often makes rather arbitrary choices on the less conserved regions. Different runs (with different noise) will produce different results, some of which are better alignments than the one shown here.

There is a better way to use SAM that is faster and produces better alignments. Since we have a specific target sequence that we are interested in, we are better off starting with a model that has a one-to-one relationship with our target. We can build such a model use `modelfromalign` on just the target sequence. The alignment of the sequences to this model is shown in the appendix as `t5_0.pretty`.

Show alignment of homologous sequences to the model built from the single sequence. Look at the `hmmscore` *results to ensure that the homologous sequences match the model. (True for the BLAST hits, not necessarily true for the Entrez neighbors, some of which are fragments that don't overlap with the target.)*

We can take the model created from the target sequence and retrain it on the homologous sequences. Since we want to preserve the relationship with the original sequence, we turn off model surgery and noise. We want to allow the alignment here to open gaps freely to align the highly conserved blocks well, even at the cost of some very poor alignments in the variable regions. We'll later try to keep just those gaps which seem to be most useful. The gap costs in SAM are encoded in the transition probabilities—we can control them somewhat by setting a special regularizer for the transitions.

Setting gap or transition costs appropriately is the hardest problem in building alignment-based models (including hidden Markov models)—we have no magic bullet for making this decision automatically. One approach is to try a variety of different transition costs, and pick the range that produces the best alignments,

where "best" can be measured automatically (highest probability for the sequences in the training set) or judged manually (highly conserved columns in known active positions, cysteines in locations known to have disulphide bridges, gaps in loop regions, and so forth).

Demo of `buildmodel` *with* `cheap_gaps.regularizer`*; the resulting alignment is in the appendix as* `t5_1.pretty`*.*

The resulting model may have created an alignment with too many places where insertions have been allowed, and with too many deletions. Retraining with a different regularizer for transitions can clean up these extraneous gaps. The `long_match` regularizer favors match-match transitions, accepting somewhat worse matches to reduce the number of gaps.

Demo of `buildmodel` *with* `long_match.regularizer`*; the resulting alignment is in the appendix as* `t5_2.pretty`*.*

All the models we've built so far have been very selective models, built to get excellent discrimination of the training set from other sequences. If we want to find more remote homologs than those in the the training set, we need to modify the model to make it less specific. We do this by giving less weight to the sequences, so that the Dirichlet mixture provides a somewhat flatter distribution of amino acids in each position. See Section 5.2 for more information about weighting schemes for sequences. For example, if we use Karplus's entropy-based weighting scheme, with the goal of getting 0.33 bits/column of savings, we get sequence weights ranging from 0.0257 to 0.0501 with a total weight of 0.599 for the set of 18 sequences.

The alignment that results form using this diffuse model is given in the appendix as `t5_2w7.pretty`. Aligning the training sequences to the model produces a somewhat different result than with the sharper models, since there is less penalty for letter mismatches, and so the gaps tend to get grouped together more.

Demo of weighting and building diffuse model. Note that we stop the training after just one re-estimation, so that the alignment used for training is not perturbed from the alignment from our sharply discriminating model.

We can, perhaps, create a slightly more general model by training on all the Entrez sequence neighbors, instead of just the small set of homologs found in SwissProt. To avoid including fragments that match parts of the sequence that we are not interested, we first select the interesting domains from the training set using `multdomain` with a sharp model built from the SwissProt sequences. The model is then retrained using this set of found domains.

We then do a minimal retraining of the model using weighted sequences to make the model less sharply discriminating. With Karplus's entropy-weighting scheme and a target of 0.33 bits/column, we get weights ranging from 0.00389 to 0.0119 with a total weight of 0.6757 for the set of 97 sequences. The alignment of the original 18 sequences to this modified model is only slightly different from the alignment we get using the similarly diffused model trained on just 18 sequences, but the range of scores is narrower, indicating a less sharply peaked model.

Demo: make `t5_2.more.mult` *(the matching domains from the larger set of sequences) and models* `t5_3.mod` *and* `t5_3w7.mod`*.*

This diffuse model can be used to search a large database to find more remote homologs, and the resulting set of sequences used to retrain the model. This iteration of finding homologs and retraining the model to match all the known homologs can be repeated until no new homologs are found. The iteration is a very effective way to build models [26].

Unfortunately, searching a large database (such as NR) can take several hours using hidden Markov models, and so will not be demonstrated here.

Searching NRP with the `t5_3w7.mod` model finds 79 matching domains, most of which are duplicates of ones already found in the Entrez nearest neighbors search. The number of sequences is only 77, because two of the sequences had ambiguous alignments, producing two overlapping hits. Some of the Entrez neighbors are not found in this search, even though the significance threshold was set very loosely to get even tentative hits.

Most of the low-scoring sequences are not remote homologs but fragments of fairly close homologs. Because SAM does not have local alignment, but only global and domain-global, fragments generally get low

total NLL−Null values, even though their score per match position is good.

At least one genuine distant homology is found, `GP:CELD1009`, but it is not very remote—BLAST was able to find it (with name `gi|1072170`) from just the target sequence.

Based on savings per character, the most distant homology is `PIR2:A05295`, which is a fairly old sequencing of bovine fibrinogen gamma fragments. A higher quality sequencing of bovine fibrinogen gamma is the SwissProt sequence `FIBG_BOVIN`, which is in the training set.

The next most distant homology is `PIR2:PC2036`, which is a fragment fairly closely homologous to `GPN:HUMMFAPA`, which is in the training set as `gi|790817`.[11]

Other models with other weightings and significance thresholds are being tried, but are not ready in time for the camera-ready copy deadline for the tutorial—if any of them produce convincing remote homologs, we'll bring additional appendices to the tutorial.

6.2 Reestimating existing alignments using HMMs

Kimmen Sjölander (kimmen@cse.ucsc.edu)

There are two primary reasons why we might want to refine an existing alignment. First, we might expect that some errors are in the alignment, and want to correct them. Second, we may wish to extend a local alignment into a global alignment. In this section, we focus on the first (easier) task. We will discuss methods for doing the second task during the tutorial, if time permits.

When the alignments are generally fairly good to start with, the method we use to refine these alignments is conservative. For instance, we have found that the best alignments in general are produced by simply *avoiding* some of the features available to us: we don't add noise into the model, we don't allow surgery, we don't let the program change the model length in any way[12]. We simply change the gap initiation and extension parameters and the amino acid probabilities at each position to maximize the probability of the sequences in the family.

Alignments can take many forms, and the method to refine an alignment has to take that form into account. For instance, at UCSC, most of our work reestimating alignments has been done on HSSP alignments [24]. These alignments indicate deleted (or inserted, depending on the perspective) regions by lower-case letters, and put the excised residues at the end of the alignment file. Because of this, we don't have a direct way of estimating the insertion or deletion probabilities for those positions, without some complicated I/O routines. Since we figure that we already write enough complicated I/O routines, we avoid that task, and estimate the model transition probabilities in a different way. We obtain an initial model from the alignment, ignoring the fact that we don't know the length of the inserts. Then, we let the sequences align themselves to the model, and reestimate the transition parameters only[13]. Once the transition parameters are tuned, we have in effect the model we would have had in one step if we'd had an initial alignment with all inserts noted directly. At this point, we allow all the parameters to be reestimated, which results in changes in the actual residues seen in each position.

An example of this process is the reestimation of an HSSP alignment for a family of hydrolase proteins (2prd.hssp). This family contains 9 sequences, forming two distinct subfamilies. The first six sequences form the first subfamily, and the last three form a second subfamily[14]. The initial alignment is fairly good, except for the region between positions 36 and 50, where the last sequence, IPYR_KLULA, is misaligned. The reestimated alignment shows an almost exact match of IPYR_KLULA to its closest homologs in the set, IPYR_BOVIN and IPYR_SCHPO, in this region. Note that the first alignment was obtained directly

[11] Wouldn't it be great if all the databases used a single name space, so you could check identity by looking at the names?

[12] This is accomplished by setting the SAM program parameters nsurgery, initial_noise, anneal_noise, and modellength all to zero in command-line arguments to `buildmodel`, as follows: % buildmodel outputmodel -i initialmodel -train sequences -initial_noise 0 -anneal_noise 0 -modellength 0 -nsurgery 0.

[13] This is achieved by setting each node in the model to be of type K, indicating amino acid emission probabilities may not be changed during training.

[14] This subfamily identification is easy to make from visual inspection of the alignment; it is also confirmed by an information theoretic program we have for determining subfamilies in an alignment.

```
                    10        20        30        40        50
                     |         |         |         |         |
ipyr_theth    ANLKSLPVGDKAPEVVHMVIEVPRGSGNKYEYDPDLGAIKLDRVLPGAQFY
ipyr_ecoli    -SLLNVPAGKDLPEDIYVVIEIPANADIKYEIDKESGALFVDRFMSTAMFY
ipyr_thep3    -------------KIVEAFIEIPTGSQNKYEFDKERGIFKLDRVLYSPMFY
ipyr_theac    --YHSVPVGPKPPEEVYVIVEIPRGSRVKYEIAKDFPGMLVDRVLYSSVVY
ipyr_arath    --WHDLEIGPEAPTVFNCAVEISKGGKVKYELDKNSGLIKVDRVLYSSIVY
ipyr_haein    -----LTPGDVDAGIINVVNEIPEGSCHKIEWNRKVAAFQLDRVEPAIFAK
ipyr_bovin    ----------ADKEVFHMVVEVPRWSNAKMEIATKLNPIKQDVKKGKLRYY
ipyr_schpo    -------YANAEKTILNMVVEIPRWTQAKLEITKELNPIKQDTKKGKLRFY
ipyr_klula    -------YADEANGIFNMVVEIPRWTNAKLEITKEKGKLRFVRNCFPHHGY
```

Table 1: Alignment from HSSP database for IPYR_THETH and homologs (2prd.hssp). Note, the columns printed are just the initial 51 for reasons of space.

```
                   10        20          30          40        50
                    |         |          |           |         |
IPYR_THETH  ANLKSLP.VGDKAPEVVHMVIEVPRGSG.NKYEYDPD..LGAIKLDRVLPGAQF.........YPGD
IPYR_ECOLI  S-LLNVP.AGKDLPEDIYVVIEIPANADpIKYEIDKE..SGALFVDRFMSTAMF.........YPCN
IPYR_THEP3  AF-----.----ENKIVEAFIEIPTGSQ.NKYEFDKE..RGIFKLDRVLYSPMF.........YPAE
IPYR_THEAC  SFYHSVP.VGPKPPEEVYVIVEIPRGSR.VKYEIAKD..FPGMLVDRVLYSSVV.........YPVD
IPYR_ARATH  HPWHDLE.IGPEAPTVFNCAVEISKGGK.VKYELDKN..SGLIKVDRVLYSSIV.........YPHN
IPYR_HAEIN  DFNQILT.PGDVDAGIINVVNEIPEGSC.HKIEWNRK..VAAFQLDRVEPAIFA.........KPTN
IPYR_BOVIN  SPFHDIP.IY-ADKEVFHMVVEVPRWSN.AKMEIATKdpLNPIKQDVKKGKLRYvanlfpykgYIWN
IPYR_SCHPO  SSWHDIPlYANAEKTILNMVVEIPRWTQ.AKLEITKEatLNPIKQDTKKGKLRFvrncfphhgYIWN
IPYR_KLULA  SAFHDIPlYADEANGIFNMVVEIPRWTN.AKLEITKEepLNPIIQDTKKGKLRFvrncfphhgYIHN
```

Table 2: Reestimated alignment for ipyr_theth and homologs.

from the HSSP file, and did not show the inserted residues with respect to the consensus. Those residues, as noted earlier, are listed at the end of the HSSP file. The second alignment shows the inserted residues with respect to the consensus, highlighting the subfamilies in the data, which make similar inserts.

For reasons of space, we include here only that particular region where the alignment shifted so dramatically.

7 Validating a model

Kevin Karplus (karplus@cse.ucsc.edu)

We've discussed several techniques for building models, but haven't asked the very important question: how do we know that the model is any good? Before we can answer that, we need to know "good for what?"

There are several different tasks that we have proposed HMMs for: recognizing sequences, finding sequences or motifs in a database, searching for remote homologs, aligning sequences, and improving existing alignments. Obviously, a model is good if it does well at the task it is intended for.

For discrimination tasks, we can give the model a number of test sequences to recognize (some that should match the model and some that shouldn't), and count how many it gets correct. The standard of truth should be the result of wet-lab work or other solid evidence, not just sequence similarity found by some other model (otherwise we're just testing the similarity of the models).

We can adjust the scoring threshold to trade off false negatives (real sequences that the model doesn't recognize) and false positives (sequences that the model thinks it recognizes, but which it shouldn't). Various measures have been proposed for summarizing this tradeoff in a single number, but none of them seem very satisfactory, since in different applications there are very different penalties for the different sorts of mistakes. Perhaps the best thing to do is to plot the number of false positives versus the number of false negatives as the threshold is adjusted.

One commonly used plot, the receiver operating characteristic (ROC), plots the sensitivity of the test versus the selectivity as the scoring threshold is changed.[15] The area under the ROC curve represents the average probability, over all pairs of sequences with differing true outcome, that the classification technique will assign a higher score to responses with a correct value of 1 compared to those with a correct value of 0. This method is used extensively in medical applications, where tests are typically rather unreliable. In our applications the curve is not usually very interesting—most of our models have no trouble getting 95–99% sensitivity with 100% selectivity.[16] This means that the area under the curve is almost always very close to one—that is, that sequences that are supposed to be modeled almost always score higher than ones that aren't. We're mainly interested in the last few percent sensitivity (finding the remote homologs) where selectivity drops rather sharply.

Another method of presentation is to provide score histograms for the positive and negative examples—the overlap between these histograms shows the tradeoff in errors. This histogram presentation is also good for a very good model, for which there will be a range of settings in which there are no mistakes made. For such models the scoring gap between the lowest scoring positive and the highest scoring negative is also a measure of the quality of the model, though one that is overly sensitive to the exact test set used. The histograms generally need to be clipped, since there are a usually a lot of negative examples—we're mainly interested in the high-scoring tail of that distribution, where we are likely to get false positives.

When validating a model used for database search, the usual approach is to search a large database and see how many of the known instances of the motif or sequence are missed. Testing sensitivity this way without a control on selectivity is not very useful, but in most cases we do not have the necessary information to claim that something found by the model is a false positive, since it is most likely an unannotated sequence about which nothing is yet known.

Searching for remote homologs is even more problematic, since we often do not have a standard of truth to compare with—if the model reports a sequence as matching, then it is a sequence homolog. There is, of course, no guarantee that it shares structure or function with the sequences in the training set, or any other property that we might be hoping homologs will share.

[15] *Sensitivity* and *selectivity* are such useful concepts in many fields that they have several different names. Sensitivity is also referred to as *recall* or *generality*—it measures how many of the sequences that are supposed to be found actually are found (the ratio of true positives to all true sequences). Selectivity is also called *precision* or *specificity*—it measures how many of the sequences that are identified by the model are correct (ratio of true positives to all positives).

[16] The high accuracy of our models may be more a function of the problems we apply them to than to any intrinsic power—"gold standard" data is only available for fairly easy discrimination problems.

When HMMs are used to improve multiple alignments, we do have an internal check to see if we have made progress—we can examine the alignment columns and see how likely such distributions are to occur in good alignments. Dirichlet mixtures provide a way of computing the probability of a particular alignment column, and we can score a multiple alignment by how likely all its alignment columns are. This does not take into account how well the insertions and deletions are handled, and it is quite possible to get misalignments that look very good in the alignment columns but which have a biologically implausible mapping of the residues.

One of the best checks for alignment is to check residues that are known to correspond (either from chemical tests or from structure-structure alignments). If these residues are not correctly aligned, our confidence in the multiple alignment drops substantially. Unfortunately, such information is rarely available for verifying new models.

8 Local HMM installation

8.1 Obtaining SAM and HMMer

To use SAM locally, the first step is to get a copy of the source code. SAM distribution is via a WWW interface. The SAM distribution can be clicked to from the main SAM page, but requires a password. To receive a password, send e-mail to sam-info@cse.ucsc.edu.[17] In response to your email, you'll receive the password, which you can then use to get to the distribution page. You'll also be added to the list of people who have received a copy of SAM, and will receive messages about any updates.

Once you gotten a copy of SAM, such as `sam.tar.gz`, you will need to do the following:

```
gunzip sam.tar.gz
tar -xf sam.tar
```

This will create a SAM directory with the source files.

See the SAM manual Section 10 for information on editing the `Makefile` and other aspects of installation.

To use HMMer locally, download a copy of the software from the HMMer WWW page and follow the instructions of the documentation's Appendix C.

8.2 SAM runtime

Once you have installed your own copy of SAM, you will have direct access to all the programs discussed in this tutorial. When using your own copy, it is important to remember that several of SAM's operations can be quite time consuming, especially building models and scoring large databases. For this reason, it is well worth the time to discover what compilers and compiler options work best on your machine.

There are several effective ways to reduce `buildmodel` runtime. One is to start with a preliminary alignment, essentially giving `buildmodel` a jump start. The number of sequences in the training set, however, has the strongest influence on runtime. `Buildmodel` can randomly select a subset of the sequences to be used as a training set (by setting the `Nseq` variable). Unfortunately, this may eliminate critical sequences. A better method is to trim the set of training sequences, either by removing homologues, or by using a weighting program to determine low-weight sequences, remove them, and then recompute the sequence weighting on this smaller set of sequences.

The scoring procedure of `hmmscore` will be faster if Viterbi scoring is used (by setting the `viterbi_score` parameter to 1). This will produce slightly different scores but the final discrimination results will be similar.

We are currently building an MPI (Message Passing Interface) implementation of the major SAM programs so that you will be able to run SAM on a network of workstations.

[17] Commercial users will have to sign a licensing agreement and pay a nominal fee.

8.3 SAM parameter settings

The SAM programs have a unified parameter setting interface that combines parameter files and command-line options. Command-line options have been seen throughout this tutorial, as have been parameter files, which are specified on the command line with the `-include`, or `-i` for short. Further discussion of parameters can be found in the documentation.

References

[1] Stephen F. Altschul. Amino acid substitution matrices from an information theoretic perspective. *JMB*, 219:555–565, 1991.

[2] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Meyers, and David J. Lippman. Basic local alignment search tool. *JMB*, 215:403–410, 1990.

[3] Timothy L. Bailey and Charles Elkan. The value of prior knowledge in discovering motifs with MEME. In *ISMB-95*, pages 21–29, Cambridge, England, July 1995.

[4] M. Borodovsky and J. McIninch. Genmark: Parallel gene recognition for both DNA strands. *Computers and Chemistry*, 17(2):123–133, 1993.

[5] Philipp Bucher, Kevin Karplus, Nicolas Moeri, and Kay Hoffman. A flexible motif search technique based on generalized profiles. *Computers and Chemistry*, 20(1):3–24, January 1996.

[6] R. F. Doolittle. *Of URFs and ORFs: A primer on how to analyze derived amino acid sequences.* University Science Books, Mill Valley, California, 1986.

[7] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* Wiley, New York, 1973.

[8] S. R. Eddy and R. Durbin. RNA sequence analysis using covariance models. *NAR*, 22:2079–2088, 1994.

[9] Sean Eddy. Multiple alignment using hidden Markov models. In *ISMB-95*, pages 114–120, Cambridge, England, July 1995. AAAI/MIT Press.

[10] S.R. Eddy, G. Mitchison, and R. Durbin. Maximum discrimination hidden Markov models of sequence consensus. *J. Comput. Biol.*, 2:9–23, 1995.

[11] Leslie Grate. Automatic RNA secondary structure determination with stochastic context-free grammars. In *ISMB-95*, pages 136–144, Menlo Park, CA, July 1995. AAAI/MIT Press.

[12] D. Haussler, A. Krogh, I. S. Mian, and K. Sjölander. Protein modeling using hidden Markov models: Analysis of globins. Technical Report UCSC-CRL-92-23, University of California at Santa Cruz, Computer and Information Sciences Dept., Santa Cruz, CA 95064, 1992.

[13] Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices from protein blocks. *PNAS*, 89:10915–10919, November 1992.

[14] Steven Henikoff and Jorja G. Henikoff. Personal communication, January 1995.

[15] R. Hughey and A. Krogh. SAM: Sequence alignment and modeling software system. Technical Report UCSC-CRL-95-7, University of California, Santa Cruz, Computer Engineering, UC Santa Cruz, CA 95064, 1995.

[16] Richard Hughey and Anders Krogh. Hidden Markov models for sequence analysis: Extension and analysis of the basic method. *CABIOS*, To appear 1996.

[17] Kevin Karplus. Using Markov models and hidden Markov models to find repetitive extragenic palindromic sequences in *Escherichia coli.* Technical Report UCSC-CRL-94-24, University of California, Santa Cruz, July 1994.

[18] Kevin Karplus. Regularizers for estimating distributions of amino acids from small samples. In *ISMB-95*, Cambridge, England, July 1995.

[19] Kevin Karplus. Regularizers for estimating distributions of amino acids from small samples. Technical Report UCSC-CRL-95-11, University of California, Santa Cruz, March 1995. URL ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-95-11.ps.Z.

[20] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. *JMB*, 235:1501–1531, February 1994.

[21] John Overington, Dan Donnelly, Mark S. Johnson, Andrej Šali, and Tom J. Blundell. Enviroment-specific amino acid substitution tables: Tertiary templates and prediction of protein folds. *Protein Science*, 1:216–226, 1992.

[22] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. of the IEEE*, 77(2):257–286, February 1989.

[23] Yasubumi Sakakibara, Michael Brown, Richard Hughey, I. Saira Mian, Kimmen Sjölander, Rebecca C. Underwood, and David Haussler. Stochastic context-free grammars for tRNA modeling. *NAR*, 22:5112–5120, 1994.

[24] C. Sander and R. Schneider. Database of homology-derived protein structures and the structural meaning of sequence alignment. *Proteins*, 9(1):56–68, 1991.

[25] Manfred J. Sippl. Calculation of conformational ensembles from potentials of mean force: an approach to the knowledge-based prediction of local structures in globular proteins. *JMB*, 213:859–883, 1990.

[26] Roman L. Tatusov, Stephen F. Altschul, and Eugen V. Koonin. Detection of conserved segments in proteins: Iterative scanning of sequence databases with alignment blocks. *PNAS*, 91:12091–12095, December 1994.

[27] Gerhard Vogt, Thure Etzold, and Patrick Argos. An assessment of amino acid exchange matrices in aligning protein sequences: The twilight zone revisited. *JMB*, 249:816–831, 1995.

9 Appendices