Regularizers for Estimating Distributions of Amino Acids from Small Samples

Kevin Karplus

UCSC-CRL-95-11
30 March 1995

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

This paper examines several different methods for estimating the distribution of amino acids in a specific context, given a very small sample of amino acids from that distribution. These distribution estimators, sometimes called *regularizers*, are frequently used when aligning sequences to each other or to models such as profiles or hidden Markov models.

The distribution estimators considered here are *zero-offsets*, *pseudocounts*, *substitution matrices* (with several variants), *feature alphabets*, and *Dirichlet mixture regularizers*.

A new method is presented for setting the parameters of the regularizers to minimize the *encoding cost* (also called the *entropy*) of the training data, for all possible samples from the training data. The optimal parameter settings depend on the size of the sample, but the optimization method can also be used to get good performance over a range of sample sizes. The optimal settings with this method are not the same as the traditional values used for the parameters.

The regularizers are evaluated based on how well they estimate the distributions of the columns of a multiple alignment—specifically, the expected encoding cost per amino acid using the regularizer method and all possible samples from each column.

The differences between the regularizers are fairly small (less than 0.2 bits per column), but large enough to make a significant difference when many columns are combined as is done in an an alignment.

In general, the pseudocounts have the lowest encoding costs for samples of size zero, substitution matrices have the lowest encoding costs for samples of size one, and Dirichlet mixtures have the lowest for larger samples. One of the substitution matrix variants, which added pseudocounts and scaled counts, does almost as well as the best Dirichlet mixtures, but with a lower computation cost.

Keywords: regularizers, pseudocounts, Gribskov average score, substitution matrices, data-dependent pseudocounts, Dirichlet mixture priors, feature alphabets, entropy, encoding cost

Contents

1 Why estimate amino acid distributions?

For many search and comparison algorithms involving protein sequences, we need to estimate the probability of seeing each of the twenty amino acids in a given context. This probability is often expressed indirectly as a *score* for each of the amino acids, with positive scores for expected amino acids and negative scores for unexpected ones.

For example, in sequence-sequence alignment, the traditional scoring matrices assign a positive score for each amino acid that would be a good match to the one in the reference sequence, and a negative score to each that would be a poor match.

As Altschul pointed out [Alt91], any alignment scoring system is really making an assertion about the probability of the test sequences given the reference sequence. The score for an alignment is the sum of the scores for individual matched positions, plus the costs for insertions and deletions. We'll only look at the match positions here, although one could make similar arguments for the amino acids in insert positions. For sequence-sequence alignment, the only information about a match position that we can use for alignment is what amino acid was seen in that position in the reference sequence.

For each match position, there are twenty scores—one for each of the possible amino acids in the test sequence. Each match score can be interpreted as the logarithm of the ratio of two estimated probabilities: the probability of the test amino acid given the amino acid in the reference sequence and the probability of the the test amino acid in the background distribution.

Let's define $\hat{P}_j(i)$ as the estimated probability that amino acid i will be seen in the test sequence aligned with amino acid j in the reference sequence and $\hat{P}_0(i)$ as the estimated probability that an amino acid i will be seen in any position of the test sequence. Then the score for matching test amino acid i to reference amino acid j is $\log_b(\hat{P}_j(i)/\hat{P}_0(i))$ for some arbitrary logarithmic base b. [Alt91]

Any method for estimating the probabilities $\hat{P}_j(i)$ and $\hat{P}_0(i)$ defines a match scoring system for sequence-sequence alignment. Rather than looking at the final scoring system, this paper will concentrate on the methods that can be used for estimating the probabilities themselves.

In more sophisticated models than single sequence alignments, such as multiple alignments, profiles [GME87], and hidden Markov models [KBM$^+$94, BCHM94], we may have more than one reference sequence in our training set. Each position of such a model will define a context for which we to want to estimate the probabilities of the twenty amino acids. The only information we will use about the context is the sampling of the amino acids we have seen in that position in the reference sequences. In this paper, I'll use s to refer to a sample of amino acids and $s(i)$ to the number of times that amino acid i appears in that sample. Our problem, then is to compute the estimated probabilities $\hat{P}_s(i)$ for the context from which sample s was taken, given only the twenty numbers $s(i)$.

Note that aligning a test sequence to a single reference sequence is a special case of this problem, in which the sample consists of just a single amino acid. Similarly, estimating the background $\hat{P}_0(i)$ is a special case in which the sample is empty ($\forall i, s(i) = 0$).

For alignment and search problems, we usually add scores from many positions, and so fairly small improvements in computing the individual match scores can add up to significant overall differences. For example, the small differences between the PAM scoring matrices and the BLOSUM scoring matrices have been shown to make a significant difference in the quality of search results [HH92].

The differences between regularizers is often fairly small. In this paper we attempt to quantify these small differences for several different methods for estimating the distributions. Section 2 explains the measure used to quantify the tests, Section 3 lists the different methods tested, Section 4 explains how the parameters of the different methods are set, Section 5 describes the data used for training and testing, and Section 6 gives the quantitative comparisons of the different methods.

2 Quantitative measure for regularizers

The traditional way in computational biology to demonstrate that a technique or set of parameters is better is to pick a biologically interesting problem and compare methods for solving it. Many of the regularizers in Section 3 have been validated in this way [HH92, BHK$^+$93, TAK94].

This sort of anecdotal evidence is very valuable for establishing that techniques are useful in real biological problems, but is very difficult to quantify. It is difficult to determine how much improvement is expected on different problems, and whether the improved technique is better in general, or just on the specific problem it was applied to.

In this paper, the regularizers are compared quantitatively on a rather generic problem—independently encoding the columns of multiple alignments. This generic problem has some attractive features:

- There are large data sets of multiple alignments available for training, making it easy to optimize the parameters of methods.
- Many of the search and alignment techniques that will use the regularizers are attempting to produce multiple alignments, and so finding a good regularizer for the encoding problem should produce a good regularizer for the search and alignment algorithms.

- By using trusted alignments, we have fairly high confidence that each amino acid distribution we see is for amino acids from a single biological context, and not just an artifact of a particular search or alignment algorithm.

Throughout this paper, the trusted alignments used are the BLOCKS database [HH91] with the sequence weighting scheme mentioned in Section 5.

2.1 Encoding cost

The encoding cost (sometimes called *conditional entropy*) is a good measure of the residual variation among sequences of the multiple alignment. Since entropy is additive, the encoding cost for independent columns can be added to get the encoding cost for entire sequences, and strict significance tests can be applied by looking at the difference in encoding cost between a hypothesized model and a null model [Mil93].

Each column t of a multiple alignment will give us a count of amino acids, $F_t(i)$. If we use sequence weights (such as those suggested in [ACL89] or [HH94]), then $F_t(i)$ is the sum of the sequence weights for sequences having amino acid i in column t. These "counts" need not be integers.

If we write the sum of all the counts for a column as $|F_t|$, we can estimate the probability of each amino acid in the column as $\hat{P}_t(i) = F_t(i)/|F_t|$. This is known as the *maximum-likelihood estimate* of the probabilities. Note: throughout this paper the notation $|y|$ will mean $\sum_{\text{amino acid } i} y(i)$ for any vector of values y.

Unfortunately, we have no way to get the true probabilities of the amino acids for a column, and the maximum-likelihood estimator is the best estimate we can make.

Because we don't have true probabilities of amino acids for each column, we can't evaluate regularizers applied to the whole column in a meaningful way. Instead, we will take a small sample of amino acids from the column, apply a regularizer to it, and see how well the regularizer estimates the probabilities for the whole column.

Let's use $s(i)$ to be the number of occurrences of amino acid i in the sample, and $|s| = \sum_i s(i)$ to be the size of the sample. The estimated probability of amino acid i given the sample s will be written as $\hat{P}_s(i)$. The Shannon entropy or optimal encoding cost of amino acid i given the sample is $-\log_2 \hat{P}_s(i)$. The encoding cost for column t given sample s is the weighted average over all amino acids in the column of the encoding for that amino acid:

$$H_s(t) = -\sum_i \frac{F_t(i)}{|F_t|} \log_2 \hat{P}_s(i) \ .$$

The better the estimation $\hat{P}_s(i)$ is of $\hat{P}_t(i)$, the lower the encoding cost $H_s(t)$ will be. The lowest possible value would be obtained if the estimate were exact (Shannon's Theorem):

$$H_{\min}(t) = -\sum_i \frac{F_t(i)}{|F_t|} \log_2 \hat{P}_t(i) = -\sum_i \frac{F_t(i)}{|F_t|} \log_2 \frac{F_t(i)}{|F_t|} \ .$$

To make a fair comparison of regularizers, we should not look at a single sample s, but at the expected value when a sample of size k is chosen at random:

$$H_k(t) = \sum_{\text{sample } s, |s|=k} P(s|t) H_s(t) \ .$$

The weighting for each of the encoding costs $H_s(t)$ is the probability of obtaining that particular sample from that column. If the samples of size $|s|$ are drawn by independent selection with replacement from the density $\hat{P}_t$, then the probability of each sample can be computed from the counts F_t:

$$\begin{aligned} P(s|t) &= |s|! \prod_i \hat{P}_t(i)^{s(i)}/s(i)! \\ &= |s|! |F_t|^{-|s|} \prod_i F_t(i)^{s(i)}/s(i)! \ . \end{aligned}$$

We can do a weighted average of the encoding costs over all columns to get the expected cost per amino acid for a given sample size:

$$\begin{aligned} H_k &= \frac{\sum_{\text{column } t} |F_t| H_k(t)}{\sum_{\text{column } t} |F_t|} \\ &= \frac{\sum_{\text{column } t} |F_t| \sum_{\text{sample } s, |s|=k} P(s|t) H_s(t)}{\sum_{\text{column } t} |F_t|} \\ &= -\frac{\sum_{\text{column } t} \sum_{\text{sample } s, |s|=k} P(s|t) \sum_i F_t(i) \log_2 \hat{P}_s(i)}{\sum_{\text{column } t} |F_t|} \ . \end{aligned}$$

2.2 More efficient computation of encoding cost

The final formula for H_k looks formidable, but we can reduce the CPU time required for computing it by rearranging the summations and precomputing some of them. If we precompute the total count $T = \sum_{\text{column } t} |F_t|$, and *summary frequencies* for each sample

$$T_s(i) = \sum_{\text{column } t} P(s|t) F_t(i) \ ,$$

then we can simplify the computation to

$$H_k = -\frac{1}{T} \sum_{\text{sample } s, |s|=k} \sum_i T_s(i) \log_2 \hat{P}_s(i) \ . \qquad (2.1)$$

We can see that the average encoding cost H_k would be minimized if $\hat{P}_s(i) = T_s(i)/|T_s|$, giving us a lower bound on how well a regularizer can do for samples of size k:

$$\begin{aligned} H_{\min,k} &= -\frac{1}{T} \sum_{s,|s|=k} |T_s| \sum_i \frac{T_s(i)}{|T_s|} \log_2 \frac{T_s(i)}{|T_s|} \\ &= -\frac{1}{T} \sum_{s,|s|=k} \sum_i T_s(i) \log_2 \frac{T_s(i)}{|T_s|} \ . \end{aligned}$$

Table 2.1 shows this lower bound on average encoding cost of the columns of the Blocks multiple alignment [HH91] (see Section 5 for details on how the database is used in this paper), given that we have sampled $|s|$ amino acids from each column. A large encoding cost means that there is a lot of variation in which amino acids occur, while a small encoding cost means that a few amino acids have very high probability, and the rest have very low probability.

The last row of the table is the average encoding cost for the columns if we use the full knowledge of the probabilities for the column $\hat{P}_t$, rather than just a random sample. This is the best we can hope to do with any method that treats the columns independently. It is probably not obtainable with any finite sample size, but we can approach it if we use information other than just a sample of amino acids to identify the column.

The relative entropy in the last column of Table 2.1 measures how much information we have gained by seeing a sample of $|s|$ amino acids (rather than $|s|-1$). The larger the sample we take from a distribution, the better we can estimate the distribution, and the fewer bits it takes to encode a column drawn from the distribution. We get the greatest gain (1.4 bits) from knowing one amino acid (as in sequence-sequence alignment) rather than zero amino acids. Each additional amino acid (for example, in a profile based on a multiple alignment) contributes less information: 0.39, 0.22, 0.14, and 0.10 bits for the next four amino acids known.

sample size $\lvert s\rvert$	encoding cost in bits $H_{\lvert s\rvert}$	relative encoding cost $H_{\lvert s\rvert-1} - H_{\lvert s\rvert}$
0	4.19666	
1	2.78084	1.41582
2	2.38691	0.39393
3	2.16913	0.21778
4	2.02703	0.14210
5	1.92380	0.10323
full	1.32961	

Table 2.1: Encoding cost of columns from the weighted Blocks database, given that a sample of $|s|$ amino acids is known. The encoding cost is a lower bound on the encoding cost for any regularizer. The last row (labeled "full") is the encoding cost if we know the distribution for each column of the alignment exactly, not just a sample from the column The relative encoding cost is the information gain from seeing one more amino acid.

$\lvert s\rvert$	Number of samples
0	1
1	20
2	210
3	1540
4	8855
5	42504
6	177100
7	657800
8	2220075

Table 2.2: The number of distinct samples of size $|s|$ grows exponentially with $|s|$, but remains manageable for $|s| \leq 5$.

The extra 0.86 bits from knowing a sample of five amino acids from a column, rather than one may seem small, but is quite important. Since searching a fairly large database may require a score difference of only 20 bits to identify a sequence as significant, the small increase of 0.86 bits per position could make an enormous difference in the quality of the searches, permitting many more short sequences to be significantly found. This extra information is what makes profiles and linear hidden Markov models so much more successful than simple sequence alignment for searching.

One disadvantage of the encoding cost computation used in this paper is the cost of pre-computing the $T_s(i)$ values and computing the $\hat{P}_s(i)$ values for each of the possible samples. The number of distinct samples to be examined is $\binom{20+|s|-1}{|s|}$, which grows exponentially with $|s|$, but remains manageable for $|s| \leq 5$ (see Table 2.2).

3 Estimation methods

In Section 2.1, the simplest method for estimating probabilities from counts. was introduced. The maximum-likelihood method, $\hat{P}_s(i) = s(i)/|s|$ is asymptotically optimal as $|s| \to \infty$, but performs very badly for small sample sizes. Using the maximum-likelihood method, the encoding cost for any amino acid not seen in the sample is $-\log_2 0 = \infty$. To avoid this infinitely high cost, we will constrain regularizers to provide non-zero estimates for all probabilities: $0 < \hat{P}_s(i) < 1$.

Many of the regularizers make a small adjustment to the sample counts to produce what we at UCSC refer to as *posterior counts.* If we use X_s to refer to the posterior counts produced by some method from sample s, then the estimated probability is

$$\hat{P}_s(i) = \frac{X_s(i)}{\sum_i X_s(i)} \ .$$

To get legal estimated probabilities, the primary constraint on X_s is that the result be positive $X_s(i) > 0$.

Note: there will be several different formulas given for computing X_s, corresponding to different regularizers. The symbols $X_s(i) \leftarrow$ will be used for defining the different methods.

The rest of this section will describe how the posterior counts are determined for each of the methods we'll be comparing. The notations $P_0(i)$ and $\hat{P}_0(i)$ refer to the background probabilities and their estimates (that is, the probabilities given a sample of size zero).

3.1 Zero-offset

The simplest method for ensuring that no probability is estimated as zero is to add a small positive *zero-offset* to each count to generate the posterior counts:

$$X_s(i) \leftarrow s(i) + z \ .$$

For large sample sizes, the zero-offset has little effect on the probability estimation, and $\hat{P}_s(i) \rightarrow P_s(i)$ as $|s| \rightarrow \infty$.

For $|s| = 0$, the estimated probability distribution will be flat ($\hat{P}_0(i) = 1/\text{alphabet size} = 0.05$), which is generally a poor approximation to the amino acid distribution in an context about which nothing is known yet.

It is fairly traditional to use $z = 1$ when nothing is known about the distributions being approximated, but this value is much too large for highly conserved regions like the Blocks database—the optimal value is between 0.048 and 0.054. Using a zero offset of $1/n$ for an n-character alphabet (0.05 for amino acids) works much better than the add-one prior for the blocks database.

3.2 Pseudocounts

Pseudocount methods are a slight variant on the zero-offset, intended to produce more reasonable distributions when $|s| = 0$. Instead of adding a constant zero-offset, a different positive constant is added for each amino acid:

$$X_s(i) \leftarrow s(i) + z(i) \ .$$

These zero-offsets are referred to as *pseudocounts*, since they are used in a way equivalent to having counted amino acids.

Again, as $|s| \rightarrow \infty$ the pseudocounts have diminishing influence on the probability estimate and $\hat{P}_s(i) \rightarrow P_s(i)$. For $|s| = 0$, we can get $\hat{P}_0(i) = P_0(i)$, by setting $z(i) = aP_0(i)$, for any positive constant a. This setting of the pseudocounts has been referred to as *background pseudocounts* [LAB$^+$93] or the *Bayesian prediction method* [TAK94] (for the Bayesian interpretation of pseudocounts, see Appendix B). For the Blocks database and $|s| > 0$, the optimal value of a is near 1.0.

For non-empty samples, the pseudocounts that minimize the encoding cost of Section 2.1 are not necessarily multiples of $P_0(i)$ (see Section 4 to see how the pseudocounts are optimized). For example, Figure 3.1 shows the the probability density implied by the optimal pseudocounts for different values of $|s|$. To get the actual pseudocounts, multiply the densities by the weight at the top of each column.

For $|s| = 0$, the weight is arbitrary, since no real counts are added to the pseudocounts, and the normalization of the posterior counts to probabilities will eliminate the overall weight. Since the weight is arbitrary, the reported weight for $|s| = 0$ is chosen to get the best performance for $|s| = 1$, holding the probabilities fixed so that optimality is not lost for $|s| = 0$.

Note that four amino acids (G=glycine, P=proline, W=tryptophan, C=cysteine) consistently have much smaller pseudocounts than would be expected from the background distribution, while three (M=methionine, Q=glutamine, and S=serine) have consistently higher pseudocounts than expected.

	optimized for $\|s\| =$				
residue	0	1	2	3	0, 1, 2, 3
weight	0.989	0.986	1.102	1.150	1.067
A	0.078	0.084	0.085	0.086	0.082
C	0.024	0.017	0.017	0.018	0.020
D	0.052	0.048	0.047	0.046	0.050
E	0.058	0.059	0.058	0.057	0.058
F	0.043	0.042	0.041	0.040	0.042
G	0.083	0.057	0.052	0.049	0.068
H	0.024	0.023	0.025	0.025	0.024
I	0.062	0.071	0.071	0.070	0.066
K	0.055	0.059	0.059	0.059	0.057
L	0.091	0.090	0.087	0.084	0.089
M	0.024	0.030	0.032	0.034	0.028
N	0.042	0.045	0.046	0.047	0.044
P	0.044	0.032	0.029	0.028	0.037
Q	0.034	0.041	0.043	0.045	0.039
R	0.050	0.049	0.049	0.049	0.050
S	0.060	0.070	0.073	0.076	0.066
T	0.055	0.062	0.064	0.065	0.059
V	0.073	0.082	0.082	0.081	0.077
W	0.014	0.010	0.009	0.009	0.012
Y	0.034	0.032	0.031	0.030	0.033

Table 3.1: Density functions corresponding to optimal pseudocounts for different sample sizes $|s|$. The pseudocounts were optimized for the entire blocks database, with weighted sequences. To get the actual pseudocounts, multiply the density by the weight for the pseudocounts given in the first row. Note that G, P, C, and W have smaller optimal pseudocounts than would be expected from scaling the background distribution ($|s| = 0$).

The pseudocounts roughly reflect the chances of seeing the amino acid in a context in which we have not previously seen it. A low pseudocount for an amino acid means that the amino acid is not often seen in a context in which some other amino acid has already been observed. If the pseudocount is lower than we would expect from the background probabilities, then the amino acid must be more highly conserved than other amino acids. Using this reasoning, we expect that G, P, W, and C are often highly conserved. Using symmetric reasoning for pseudocounts that are higher than expected from the background probabilities, we also expect that M, Q, and S are less conserved than other amino acids.

3.3 Gribskov average-score method

The *Gribskov profile* [GME87] or *average-score method* [TAK94] computes the weighted average of scores from a *score matrix* M. There are several standard scoring matrices in use, most notably the Dayhoff matrices [DSO78] and the BLOSUM matrices [HH92], which were originally created for aligning one sequence with another ($|s| = 1$).

The scores are best interpreted as the logarithm of the ratio of the probability of the amino acid in the context to the background probability [Alt91]:

$$M_{i,j} = \log \hat{P}_s(i)/P_0(i) \ ,$$

where s is a sample containing exactly one amino acid: j.

The averaging of the score matrices is intended to create a new score. With the interpretation of scores given above, and assuming natural logarithms are used, the posterior counts are

$$X_s(i) \leftarrow P_0(i)\text{exp}\left(\frac{\sum_j M_{i,j}s(j)}{|s|}\right) .$$

We can avoid recording the extra parameters $P_0(i)$ by redefining the score matrix slightly. If we let $M'_{i,j} = M_{i,j} + \ln P_0(i)$, then

$$X_s(i) \leftarrow \exp\left(\frac{\sum_j M'_{i,j}s(j)}{|s|}\right) .$$

The BLOSUM substitution matrices provide a score matrix

$$M_{i,j} = \log\left(\frac{P(i,j)}{P_0(i)P_0(j)}\right)$$

for matching amino acid i and amino acid j, where $P(i,j)$ is the probability of i and j appearing as an ordered pair in any column of a correct alignment. Let's take natural logarithms in creating the score matrix (to match the exponential in the computation of $X_s(i)$). If we use j to name the sample consisting of a single amino acid j, then

$$\hat{P}_j(i) = X_j(i) = \frac{P(i,j)}{P_0(j)} .$$

This is the optimal value for $\hat{P}_j$, and so the Gribskov average score method is optimal for $|s| = 1$ (with a properly chosen score matrix).

Although the Gribskov average score method is optimal at $|s| = 1$, it does not perform well at the extremes. For $|s| = 0$, it predicts a completely flat distribution (just as zero-offset methods do). As $|s| \rightarrow \infty$, the Gribskov average-score method does not approach a maximum-likelihood estimate for $\hat{P}_s(i)$.

We can get much better performance for $|s| > 1$ by optimizing the score matrix as described in Section 4, but the Gribskov average-score method does not generalize to other values of $|s|$ as well the substitution matrix method described in Section 3.4.

3.4 Substitution matrices

A substitution matrix computes the posterior counts as a linear combination of the counts:

$$X_s(i) \leftarrow \sum_j M_{i,j}s(j) .$$

This method is similar to the Gribskov average-score method of Section 3.3, with one major difference—the matrix M is not a logarithmic score matrix.

Note that for $|s| = 0$, all the sample counts $s(j)$ are zero, and so the posterior counts $X_0(i)$ are also zero. This violates the constraints on posterior counts, and so some other method of deriving posterior counts is needed for $|s| = 0$. For the experiments in this paper, all-zero count vectors are replaced by all-one count vectors ($s(j) = 1$ and $X_0(i) = \sum_j M_{i,j}$). This is equivalent to adding an infinitesimal zero-offset to the count vectors before multiplying by the substitution matrix M.

Substitution matrices, like score matrices, are designed for use in sequence-sequence alignment, where the sample always consists of exactly one amino acid ($|s| = 1$). If we let P_j be the distribution we expect in a column in which amino acid j has been seen, we can can get $\hat{P}_j(i) = P_j(i)$ by setting $M_{i,j} = a_j P_j(i)$, for arbitrary positive constants a_j.

If we set $a_j = aP_0(j)$, then $X_0(i) = \sum_j aP_0(j)P_j(i) = aP_0(i)$, and we get optimal estimation for $|s| = 0$ ($\hat{P}_0(i) = P_0(i)$) as well as $|s| = 1$.

If we set $M_{i,j} = P(i,j)/P_0(j) = P(i|j)$, and choose s to be the sample for which $s(m) = 1$ and $s(j \neq m) = 0$, then $\hat{P}_s(i) = P(i|m)$. The value $P(i,j)/P_0(j)$ is known as the *relatedness odds ratio* and has been widely used, for example [JTT92].

			eigenvectors					
	background	frequency	subst		subst+pseudo		subst+pseudo+scaled	
residue	density	matrix	0–3	2	0–3	2	0–3	2
A	0.078	0.099	0.073	0.072	0.063	0.068	0.071	0.066
C	0.024	0.011	0.013	0.010	0.008	0.009	0.009	0.009
D	0.052	0.036	0.063	0.069	0.099	0.079	0.071	0.076
E	0.058	0.043	0.077	0.090	0.117	0.104	0.101	0.097
F	0.043	0.031	0.037	0.031	0.022	0.024	0.028	0.030
G	0.083	0.168	0.071	0.065	0.085	0.060	0.056	0.067
H	0.024	0.011	0.022	0.021	0.020	0.020	0.020	0.019
I	0.062	0.072	0.059	0.055	0.042	0.053	0.054	0.056
K	0.055	0.040	0.070	0.078	0.091	0.086	0.088	0.083
L	0.091	0.144	0.086	0.081	0.061	0.074	0.078	0.082
M	0.024	0.021	0.022	0.020	0.016	0.019	0.020	0.018
N	0.042	0.028	0.044	0.046	0.051	0.046	0.047	0.045
P	0.044	0.027	0.050	0.050	0.042	0.050	0.041	0.055
Q	0.034	0.022	0.039	0.044	0.048	0.046	0.048	0.041
R	0.050	0.034	0.057	0.061	0.064	0.069	0.064	0.062
S	0.060	0.055	0.056	0.056	0.052	0.056	0.057	0.051
T	0.055	0.048	0.053	0.052	0.046	0.049	0.052	0.047
V	0.073	0.088	0.068	0.064	0.048	0.060	0.063	0.063
W	0.014	0.005	0.011	0.009	0.007	0.007	0.007	0.007
Y	0.034	0.018	0.030	0.026	0.019	0.021	0.024	0.024

Table 3.2: Principal eigenvectors of substitution matrices for the blocks database, scaled so that each vector sums to one. Each gives the stationary distribution for a mutation process modeled by the substitution matrix. The eigenvectors are remarkably consistent, except for the frequency matrix, which is just $P(i,j)$, not an optimal substitution matrix. The headings "2" and "0–3" indicate that the regularizer was optimized for $|s| = 2$ or simultaneously for $|s| = 0, 1, 2, 3$ on the weighted blocks database. See Sections 3.5 and 3.6 for details on the substitution matrices used with pseudocounts and scaled counts.

If we are only interested in samples with a single amino acid, we can even use $M_{i,j} = P(i,j)$, since the normalization makes dividing by $P_0(j)$ irrelevant. Indeed, for $|s| = 1$ we can multiply each column by a different arbitrary number without affecting the resulting estimate of the distribution. The pure frequency matrix sets the column weights to be $P_0(j)$, while using the relatedness odd ratio sets the column weights to 1.0.

Pure frequency matrices do not work well for larger samples, and using the relatedness odds ratio is not much better. In optimal substitution matrices, the column weights vary over a range of about 2 to 1, not 7 to 1 (as they would be for $P(i,j)$) or uniform (as they would be for $P(i,j)/P_0(j)$).

Furthermore, the heaviest column weight is not necessarily for the most frequent amino acids—the most frequent amino acids in the data are L and G, but the heaviest weights in the optimum substitution matrices are on columns K and E.

For large values of $|s|$, the substitution matrix does not guarantee that the estimated distribution approaches the true distribution, unless the count vector s happens to be an eigenvector of the matrix. If the substitution matrix is interpreted as a mutation process modeled as a Markov chain with transition probabilities assigned by the substitution matrix, then the principal eigenvector should be the background distribution.

Eigenvectors were determined for several substitution matrices, and only the eigenvector with the largest eigenvalue had all positive components. Table 3.2 shows these principal eigenvectors for several substitution matrices. These principal eigenvectors are very similar to the background distribution, as one would expect. Eigenvectors have been used to examine distance matrices [TJ93, Hig92], but I have not yet found a reference to the use of eigenvectors with substitution matrices.

Note that each diagonal element of an optimal substitution matrix, divided by the total weight of the column, reflects how conserved that amino acid is. Based on the observations on the optimal pseudocounts, we expect the diagonal elements GG, PP, WW, and CC to be large, and the diagonal elements MM, QQ, and SS to be small. This indeed turns out to be the case.

3.5 Substitution matrices plus pseudocounts

In an attempt to avoid the rather ad hoc approach for handling $|s| = 0$ with substitution methods, I created a new method which combines substitution matrices and pseudocount methods:

$$X_s(i) \leftarrow z(i) + \sum_j M_{i,j} s(j) \ .$$

If one thinks of a substitution matrix as a mutation model, then the pseudocounts represent a mutation or substitution that does not depend on what is currently in the sequence. For doing single alignments, where there is exactly one $s(i)$ that is non-zero, one could get the same effect by adding the pseudocounts to each column of the substitution matrix, but for other sample sizes, the separation of residue-specific and background substitutions turns out to be quite useful.

If $z(i)$ is set to $aP_0(i)$ for a very small positive number a, then the method is essentially identical to the pure substitution matrix method. If M is set to be the identity matrix, then the method is identical to the pure pseudocount method. In practice, the optimal matrix is closer to the identity matrix than the simple substitution matrix is, but still has significant off-diagonal elements. The pseudocounts sum to between 0.15 and 0.45 (assuming the matrix M is scaled so that the diagonal elements sum to 20), rather than around 1.0 as they would in a pure pseudocount method.

As with substitution matrices, substitution matrices plus pseudocounts do not converge to the optimal distribution as $|s| \rightarrow \infty$. They do a little better than pure substitution matrices, since the matrix is closer to being an identity matrix.

The eigenvectors of the substitution matrices for this combined approach may be even more interesting than the ones for the simple substitution matrix, since any "noise" introduced by random mutation uninfluenced by what was previously in the position is modeled by the pseudocounts, rather than blurring the substitution matrix.

Eigenvectors were determined for several substitution matrices optimized for use with pseudocounts. As with the other substitution matrices, only the dominant eigenvector had all positive elements. Table 3.2 includes these eigenvectors.

We can apply the same analysis for which amino acids tend to be highly conserved as we did for straight substitution matrices. Again the diagonal elements GG, PP, WW, and CC are large, indicating that they are more highly conserved, and MM, QQ, and SS are small. The diagonal elements II and KK are also quite small.

3.6 Substitution matrices plus pseudocounts plus scaled counts

Substitution matrices can be modified to work better for $|s| = 0$ by adding pseudocounts as described in Section 3.5, but they still do not converge to the maximum-likelihood estimate as $|s| \rightarrow \infty$. This problem can be solved by adding one more term to the posterior counts, proportional to the counts and growing faster than the vector Ms does. One easy way to accomplish this is to add the counts scaled by their sum:

$$X_s(i) \leftarrow |s| s(i) + z(i) + \sum_j M_{i,j} s(j) \ .$$

The method here is almost equivalent to the data-dependent pseudocount method [TAK94]. The data-dependent method sets

$$X_s(i) \leftarrow s(i) + \frac{\sum_j BP_0(i) e^{\lambda A_{i,j}} s(j)}{|s|} \ ,$$

for arbitrary parameters B and λ and a substitution matrix A. Scaling this by $|s|$ and absorbing the constants and exponentiation into the matrix gives us

$$X_s(i) \leftarrow |s|s(i) + \sum_j M_{i,j} s(j) \ ,$$

which is identical to the method here, if the pseudocounts $z(i)$ are all zero. However, the construction of their matrices is rather ad hoc, and probably not optimized for the task.

A similar method was proposed by Claverie [Cla94]. His method is equivalent to setting $z(i) = 0$ and scaling the $s(i)$ by $\max(\sqrt{|s|}, |s|/20)$, instead of $|s|$. It might be interesting to try other scaling functions, besides $|s|$ or $\sqrt{|s|}$—any positive function such that $f(|s|) \rightarrow \infty$ as $|s| \rightarrow \infty$ would give the correct convergence to the maximum-likelihood estimate. Since there is not yet any theoretical justification for choosing one scaling function over another, I have chosen the simplest one.

3.7 Feature alphabets

The feature-alphabet method partitions the set of amino acids into disjoint *feature sets*, then treats the different feature sets as a reduced alphabet, with all amino acids in a feature set considered equivalent. This partition of the amino acids into disjoint feature sets is called a *feature alphabet*. For a feature alphabet f, let's call the set to which amino acid i belongs $f(i)$.

A single feature alphabet would not provide very good matches to the amino acid distributions (since all the amino acids in a particular feature get identical estimates), and so distribution estimation needs to combine the effects of several different feature alphabets.

Although any of the distribution estimation methods could be used for each feature alphabet, I've chosen to use the simplest method (zero-offsets), since the intent of the method is to capture the relationships between the amino acids in the feature sets, not in a large set of parameters. To combine the feature alphabets, I multiply the posterior counts together.

$$\begin{aligned} X_s(i) &\leftarrow \prod_{\text{feature alphabet } f} X_{s,f}(i) \\ X_{s,f}(i) &= z_f + \sum_{j, f(j)=f(i)} s(j) \end{aligned}$$

With four properly chosen feature alphabets and only 4 parameters, this method does better than pseudocount methods (which require 20 parameters) for $|s| = 1, 2, 3$.

One would not expect feature sets to work well for $|s| = 0$ since the zero-offset methods always predict a flat distribution for $|s| = 0$, and combining the flat predictions still results in a flat prediction.

For large samples ($|s| \rightarrow \infty$), we again expect the feature alphabets to do poorly compared to the pseudocount methods, but are superior to the substitution matrix methods (unless scaled counts are added to the substitution matrix pseudocounts, as in Section 3.6).

Feature alphabets have been used for creating patterns for searching protein databases [SS90], and feature sets have been quite popular for describing sets of amino acids (for example, [Tay86]). The feature alphabets are used in quite a different way by Smith and Smith [SS90] than the ones presented here, since their scores were computed based only on the set of amino acids previously seen in a context, not on the frequencies. Their method could be roughly approximated by the feature alphabet in Table 3.3.

A few feature alphabet sets were created by hand from the Taylor features [Tay86], with from four to fifty-eight feature alphabets. The largest one consisted of one alphabet for the individual amino acids and 57 binary alphabets corresponding to the "most relevant" feature sets given in [Tay86,

features	zero offsets
A, C, D, E, F, G, H, I, V, K, L, M, N, P, Q, R, S, T, W, Y	0.168058
DE, FRH, NQ, ST, ILV, FWY, C, M, AG, P	0.764163
DEKRHNQST, ILVFWYCM, AG, P	2.062930

Table 3.3: Feature alphabet set based on the amino acid class hierarchy [SS90]. Zero offsets were chosen for best performance on $|s| = 1, 2$.

features	zero offsets
A, C, D, E, F, G, H, I, V, K, L, M, N, P, Q, R, S, T, W, Y	0.102741
FILMPV, ACGHNQSTWY, DEKR	2.348270
DE, HKR, NQSTWY, ACFGILMPV	3.031990
AGS, C, DNPTV, EFHIKLMQRWY	3.624300

Table 3.4: Four-alphabet feature alphabet set based on Taylor's feature set [Tay86]. Zero offsets were chosen for best performance on $|s| = 1, 2$.

features	zero offsets
A, C, D, E, F, G, H, I, V, K, L, M, N, P, Q, R, S, T, W, Y	0.172476
AGS, C, P, DE, ILV, KMNQRT, FHWY	1.87884
ACFGILMTVWY, DEHKNPQRS	2.59881
ACDEGIKLMNPQRSTV, FHWY	2.65647
CFILMPV, ADEGHKNQRSTWY	2.91754
ACDEFGILMNPQSTVWY, HKR	5.04085
CEFHIKLMQRVWY, ADGNPST	5.64207
FHILMVWY, ACDEGKNPQRST	6.23212
ACDEGIKLNPQRSTV, FHMWY	6.19452
ACDFGIKLMNSTV, EHPQRWY	7.54467

Table 3.5: Ten-alphabet feature alphabet set based on Taylor's feature set [Tay86]. Zero offsets were chosen for best performance on $|s| = 1, 2$.

Figure 5].[1] The smaller sets attempted to improve performance and reduce the size of the regularizer by grouping non-overlapping features into the same feature alphabet, and omitting the less useful feature alphabets. Two of these are shown in Tables 3.4 and 3.5. The ten-alphabet set works well for $|s| = 1$ and $|s| = 2$, but does poorly for larger values of $|s|$, and the four-alphabet set works moderately well for larger $|s|$. None of these hand-created feature sets did as well as the automatically created ones, and none of the feature-based techniques worked particularly well.

The automatically generated feature sets used in these experiments were chosen with a simple greedy algorithm (after months of playing around with fancy search programs!). The algorithm starts with the empty set (0 alphabets), which predicts a flat distribution, then repeatedly adds an alphabet to the set, re-optimizing the zero offsets after each addition.

The new alphabet is chosen by starting with the alphabet that assigns each amino acid its own feature and gradually merging features together. A trial merge is made for every pair of features, and whichever produces the smallest entropy is chosen, until any merging will increase the entropy. For efficiency in computing the entropy in the inner loop, the zero offsets are not modified for any of the existing alphabets, and the entropy is only computed for samples with a single amino acid in them. Not changing the existing regularizer while doing the merging also makes it possible to apply the partitioning algorithm to other regularizers, since all that is needed are the $X_s(i)$ values for the 20 samples consisting of a single amino acid each.

[1] Figure 5 of [Tay86] lists 61 sets, but several of them are complements of others in the list and complementary sets generate the same feature alphabet, giving only 57 distinct feature alphabets.

	zero offsets	
features	4 alphabets	8 alphabets
A, C, D, E, F, G, H, IV, K, L, M, N, P, Q, R, S, T, W, Y	0.104006	0.179209
ADEGKNPQRSTV, CFHILMWY	0.976318	1.034
ADEFGHIKLMNQRSTVY, CPW	0.892489	0.882162
ACFGILMPSTVWY, DEHKNQR	1.52301	1.77172
ACDEILMNPSTV, FGHKQRWY		3.18251
ACDEGHNQSTWY, FIKLMPRV		3.41063
ACGHIKLMNPQRSTV, DEFWY		3.32751
AEFIKLMPQRSTVWY, CDGHN		2.96088

Table 3.6: Feature alphabet set for the entire blocks database, found by the greedy algorithm in Section 3.7. Note that the smaller the zero offset, the greater the effect of the feature alphabet on the estimated probabilities. Zero offsets are given for the four-alphabet and the eight-alphabet regularizers.

Table 3.6 gives an example of a feature alphabet produced by this algorithm, trained on the entire blocks database using position-specific weighting for the sequences. The zero offsets have been optimized to minimize the average entropy for $|s| = 1$ and $|s| = 2$ (though the feature alphabets themselves were selected using just $|s| = 1$, as explained above). Note, it is not claimed that these feature alphabets are the best one can do, but they are better than any of the hand-created feature sets tried, and as good as the ones found by my earlier search programs.

3.8 Dirichlet mixtures

The Dirichlet mixture method [BHK+93] has similarities to the pseudocount methods, but is somewhat more complex. They have been used quite successfully by several researchers [BHK+93, TAK94, HH95]. In Section 6, we'll see that Dirichlet mixtures are superior to all the other regularizers examined, and that there is not much room for improvement to better regularizers.

One way to view the posterior counts of Dirichlet mixtures is as a linear combination of pseudocount regularizers, where the weights on the combination vary from one sample to another, but the underlying regularizers are fixed. Each pseudocount regularizer is referred to as a *component* of the mixture. The weights for the components are the product of two numbers—a prior weight q_c called the *mixture coefficient* and a weight that is proportional to the likelihood of the sample given the component.

Each pseudocount regularizer defines a Dirichlet density function (ρ_1 through ρ_k) on the possible samples, with ρ_c characterized by the pseudocounts $z_c(i)$. Thus a 9-component Dirichlet mixture for the amino acids will have 168 degrees of freedom: 9 pseudocount vectors with 20-components each and 9 mixture coefficients (whose sum can be normalized to 1.0).

We need to introduce some notation—the Gamma and Beta functions. The Gamma function is the continuous generalization of the integer factorial function $?(n+1) = n!$ and the Beta function is a generalization of the binomial coefficients:

$$\mathrm{B}(a) = \frac{\prod_i ?(a(i))}{?(\sum_i a(i))} .$$

With this notation, we can define

$$X_s(i) \leftarrow \sum_{1 \le c \le k} q_c \frac{\mathrm{B}(z_c + s)}{\mathrm{B}(z_c)} (z_c(i) + s(i)) ,$$

where $z_c + s$ should be interpreted as the component-wise sum of the two vectors. The derivation of this formula using Bayesian statistics can be found in Appendices B and C.

It is tempting to combine the q_c and $\mathrm{B}(z_c)$ terms, since neither depends on s, but the B values can get quite large and quite small, and so it is useful to take the ratio of the two Bs, to keep the coefficients conveniently scaled. In any case, it is a good idea to compute and store $\log \mathrm{B}$ rather than B, to avoid range problems with floating-point numbers.

Because each of the pseudocount regularizers approaches the maximum-likelihood estimate as $|s| \to \infty$, the Dirichlet mixture will also have the correct behavior in the limit. For $|s| = 0$, the Beta functions cancel, and we have

$$X_0(i) \leftarrow \sum_{1 \leq c \leq k} q_c z_c(i) \ ,$$

which can easily be made to fit the background distribution.

4 Optimizing parameters of estimation methods

Once we have decided that the goal is to minimize the average encoding cost of the columns, and chosen a method to try, we can try to optimize the parameters of the method, using Newton's method or gradient descent to find parameter values at which all the first derivatives of the encoding cost are zero, and all the second derivatives are positive.

We can compute the derivatives of the encoding cost H_k (as given in Equation 2.1) with respect to some parameter p fairly easily from X_s and its derivatives:

$$\begin{aligned}
\frac{\partial H_k}{\partial p} &= -\frac{\log_2 e}{T} \sum_s \sum_i T_s(i) \left(\frac{\partial X_s(i)}{\partial p} / X_s(i) - \sum_j \frac{\partial X_s(j)}{\partial p} / \sum_j X_s(j) \right) \\
\frac{\partial^2 H_k}{\partial p^2} &= -\frac{\log_2 e}{T} \sum_s \sum_i T_s(i) \left(\frac{\partial^2 X_s(i)}{\partial p^2} X_s(i)^{-1} - \left(\frac{\partial X_s(i)}{\partial p} / X_s(i) \right)^2 \right. \\
&\qquad \left. - \sum_j \frac{\partial^2 X_s(j)}{\partial p^2} \left(\sum_j X_s(j))^{-1} + \left(\sum_j \frac{\partial X_s(j)}{\partial p} / \sum_j X_s(j) \right)^2 \right) \right)
\end{aligned}$$

For many of the methods, the second derivative of X_s is 0 for all the parameters, simplifying the optimization further.

Newton's method for optimizing the parameter vector v consists of iterating the assignment

$$v(i) \leftarrow v(i) - \frac{\frac{\partial H_k}{\partial v(i)}}{\frac{\partial^2 H_k}{\partial v(i)^2}} \ .$$

For most of the methods, if the second partial is negative, then the parameter is too large, and we replace it by a smaller value, not using Newton's method. Some care needs to be taken in doing the iterations to make sure that the parameters stay within legal range. There are various techniques that can be tried for accelerating the convergence, such as multiplying the correction term by a constant that is less than one if the iterations seem to be oscillating, or greater than one if the iterations seem to be approaching the optimum from one side. These tricks for accelerating convergence are beyond the scope of this tech report, but can be found in books on non-linear optimization. None of the tricks are particularly robust, and the optimization problem to be solved here is often ill-conditioned, with many rather different settings of the parameters giving very similar results.

Starting off with good estimates of the parameters helps a lot. Zero offsets were started at 0.05 (so that they sum to 1.0 over the alphabet). Pseudocounts were started at the observed probabilities for the individual amino acids in the entire data set. Substitution matrices were started with each column being the correct probabilities (from the summary T_s) for a sample consisting of the single amino acid corresponding to the column number. Pseudocounts that were added to substitution matrices were initialized like other pseudocounts, but scaled to sum to 0.2 instead of 1.0.

This optimization method works quite well with all the linear methods (zero-offset, pseudocount, substitution matrix, feature alphabets), but the Dirichlet mixtures cause a problem: the partial derivatives are difficult to compute for the $z_c(i)$ parameters (see Appendix A). A simpler approach, that seems to work at least as well, is to optimize the mixture coefficients q_c using the correct derivatives, but using approximate derivatives for the components of the mixture—pretending that $P(s|\rho_c)$ is a constant independent of z_c, but recomputing it when the z_c values actually do change.

One also needs good starting estimates for the components of the mixture. After trying several methods, I settled on using the same partitioning method as was used for finding feature alphabets (Section 3.7). To add n components to an existing (possibly empty) set of components, I merged features together until the number of features had been reduced to n. A component was created for each feature by summing the summary frequencies for all samples in which all elements were in the feature, and scaling (rather arbitrarily) so that the pseudocounts for the component added to 20. This provided good starting points for the optimization, though the best results were obtained by adding 1, 2, 3, ...components and optimizing after each addition, rather than adding a lot of components at once.

The optimized Dirichlet mixtures improve with increasing number of components fairly smoothly, though with diminishing returns, making the choice of number of components difficult.

5 Experimental method

Two slightly different methods were used for evaluating regularizers. Both involve computing the average entropy of a multiple alignment given a regularizer (as in Equation 2.1). In one method, the parameters of the regularizer are adjusted using the same multiple alignment, while in the other the regularizer is trained on a different multiple alignment.

The first method gives us an estimate of how well we can do with the best tuned regularizer, while the second method gives us an estimate of how well the regularizer generalizes to other similar problems.

The multiple alignments chosen are the BLOCKS database [HH91]. The sequences are weighted using a slight variant of the Henikoffs' position-specific weighting scheme [HH94], as implemented by Kimmen Sjölander. Sjölander's weighting scheme is proportional to the Henikoffs' position-specific weights, but instead of having the weights sum to 1.0 for each block, they sum to the number of sequences in the block, so that blocks with more sequences in them have more influence than blocks with only a few sequences. Other weighting schemes have been used by other researchers (for example, tree distances [THG94] or weighting for pairs of alignments [ACL89]), and the position-specific one used here was chosen rather arbitrarily for its ease of computation.

The experiments that used the same set for training and testing used the entire blocks database, but separate train-test sets were created as disjoint random subsets. The subsets were created putting entire blocks randomly into one of the subsets, not by randomly assigning individual columns.

6 Results for training and testing on full database

This section contains the average encoding costs obtained for different sample sizes and different regularizers. To simplify the presentation, the results for each class of regularizers will be presented separately. For each regularizer and sample size, the *excess entropy* is reported, that is, the difference between the average encoding cost per column using the regularizer and the average encoding cost per column using the best theoretically possible optimizer. For the entire database, the best possible encoding costs are reported in Table 2.1.

The last row of each table reports the excess entropy if the full column F_t is given to the regularizer, rather than a sample s. Since the entropy for the column is minimized if the $X_t(i)$ values exactly match the observed counts $F_t(i)$, this measures how much the regularizer distorts the data. Because some of the columns have few counts, it is not the same as letting $|s| \rightarrow \infty$, but offers a more realistic idea of what can be expected with large sample sizes.

	excess entropy			
$\|s\|$	$z = 1$	$z = 0.04851$	$z = 0.05420$	$z = 0.05260$
0	0.12527	0.12527	0.12527	0.12527
1	1.07961	0.20482	0.20677	0.20585
2	1.17080	0.18636	0.18457	0.18470
3	1.16489	0.16843	0.16587	0.16626
4	1.13144	0.15311	0.15063	0.15105
5	1.09164	0.14203	0.13989	0.14026
full	0.84541	0.08013	0.08884	0.08653

Table 6.1: Excess entropy for the zero-offset regularizers applied to the full blocks database. The popular "add-one" regularizer is clearly a poor choice for this database.

	excess entropy optimized for $\|s\| =$				
$\|s\|$	0	1	2	3	0,1,2,3
0	0.00000	0.01910	0.02848	0.03459	0.00610
1	0.14221	0.13384	0.13643	0.13925	0.13644
2	0.14499	0.13681	0.13455	0.13497	0.13720
3	0.13838	0.13127	0.12792	0.12757	0.13097
4	0.13006	0.12397	0.12060	0.12004	0.12350
5	0.12369	0.11845	0.11543	0.11484	0.11804
full	0.07966	0.07913	0.08767	0.09129	0.08521

Table 6.2: Excess entropy for pseudocount regularizers applied to the full blocks database.

6.1 Zero-offset

For the blocks database, the optimal zero offset is approximately 0.05. When optimizing for $|s| = 1$, the optimum is 0.04851, for $|s| = 2$ the optimum is 0.05420, and for the average over all samples of size 0, 1, 2, and 3, the optimum is 0.05260. Table 6.1 presents the excess entropy for each of these three regularizers, as well as the popular "add-one" regularizer.

6.2 Pseudocounts

The pseudocounts were optimized for $|s| = 0$ through $|s| = 3$, both separately, and minimizing the average entropy for all four sample sizes combined. The pseudocounts themselves are presented in Table 3.1, and the excess entropy for each is given in Table 6.2. The pseudocount regularizers do much better than the zero-offset regularizers for $|s| = 0$ and $|s| = 1$, but already by $|s| = 5$, the difference is only 0.025 bits per column.

6.3 Gribskov average score

Four different score matrices were tested: the BLOSUM62 matrix (appropriately modified to represent $\ln P(i,j)/P_0(j)$ for the BLOSUM62 data), the log-odds matrix $\ln P(i,j)/P_0(j)$ for the test data, a matrix optimized for $|s| = 2$, and one optimized for $|s| = 0,1,2,3$. The excess entropies are presented in Table 6.3.

6.4 Substitution matrices

Four different substitution matrices were tested: the frequency matrix from which the BLOSUM62 scoring matrix was derived, a frequency matrix computed from the weighted blocks database, a substitution matrix optimized for $|s| = 2$, and one optimized for $|s| = 0,1,2,3$. Table 6.4 presents the excess entropies.

	excess entropy			
	blosum62	log-odds	optimized for $\|s\| =$	
$\|s\|$			2	0,1,2,3
0	0.12527	0.12527	0.12527	0.12527
1	0.13294	0.00000	0.19046	0.08408
2	0.41066	0.13311	0.01694	0.03136
3	0.59404	0.27749	0.08442	0.12809
4	0.71962	0.38475	0.15471	0.21261
5	0.81315	0.46765	0.21601	0.28223
full	1.37003	0.98464	0.65103	0.74889

Table 6.3: Excess entropy for Gribskov average score regularizers applied to the full blocks database.

	excess entropy			
	blosum62	frequency	optimized for $\|s\| =$	
$\|s\|$		matrix	2	0,1,2,3
0	0.00369	0.00000	0.05348	0.05270
1	0.13294	0.00000	0.05723	0.03647
2	0.35455	0.08581	0.02495	0.02708
3	0.50493	0.17251	0.05577	0.06792
4	0.61172	0.24275	0.09300	0.11082
5	0.69341	0.30091	0.12933	0.15083
full	1.20369	0.71452	0.44748	0.48373

Table 6.4: Excess entropy for substitution matrix regularizers applied to the full blocks database.

The pure frequency matrix is optimal for $|s| = 0$ and $|s| = 1$, but degrades badly for larger samples, and is worse than pseudocounts for $|s| = 3$. The blosum62 matrix does not do well for any sample size greater than zero, probably because of the difference in weighting schemes used for building the matrix and for testing.

Optimizing the substitution matrix can preserve its superiority over pseudocounts up to $|s| = 4$, but the pseudocounts get closer to the optimum regularizer as the sample size increases, while the substitution matrices get farther from the optimum.

6.5 Substitution matrices plus pseudocounts

Adding pseudocounts, scaled counts, or both to the substitution matrices improves their performance significantly. Table 6.5 presents the excess entropies for these regularizers. The full method, using scaled counts and pseudocounts as well as the substitution matrix, has the best results of any of the methods tried so far.

The matrix for the best of these substitution matrix methods is shown in Table 6.6. Note that use of scaled counts allows the diagonal of the substitution matrix to be negative without risking negative or zero posterior counts (as long as $M_{i,i} > -1$), but this matrix has no negative entries.

6.6 Feature alphabets

The feature alphabets used are shown in Table 3.6. Because the feature alphabet sets were created by adding a new alphabet to an existing feature alphabet set, the n-alphabet set is just the first n rows of the table. The zero-offsets are given in the table for the 4-alphabet and 8-alphabet sets.

	excess entropy					
	subst+pseudo		subst+scaled		subst+pseudo+scaled	
	optimized for $\|s\| =$		optimized for $\|s\| =$		optimized for $\|s\| =$	
$\|s\|$	2	0,1,2,3	2	0,1,2,3	2	0,1,2,3
0	0.02555	0.00012	1.00651	0.00099	0.43012	0.00000
1	0.02670	0.01080	0.01970	0.00734	0.02960	0.00080
2	0.02498	0.02595	0.02502	0.03105	0.02496	0.02509
3	0.04969	0.04743	0.04099	0.04823	0.04157	0.03975
4	0.07834	0.06937	0.05093	0.05753	0.05210	0.04849
5	0.10718	0.09152	0.05833	0.06407	0.05973	0.05548
full	0.38692	0.32624	0.07968	0.07789	0.07492	0.09645

Table 6.5: Excess entropy for substitution matrix regularizers with pseudocounts and pseudocounts plus scaled counts applied to the full blocks database.

residue	A	C	D	E	F	G	H	I	K	L
A	0.7173	0.0638	0.0782	0.1558	0.0433	0.1378	0.0450	0.0859	0.1374	0.0755
C	0.0482	0.1955	0.0065	0.0078	0.0177	0.0098	0.0090	0.0279	0.0120	0.0203
D	0.0408	0.0000	1.2001	0.3464	0.0093	0.0475	0.0542	0.0091	0.1001	0.0148
E	0.0740	0.0000	0.3192	1.4279	0.0109	0.0293	0.0590	0.0231	0.2112	0.0278
F	0.0283	0.0126	0.0144	0.0199	0.6603	0.0102	0.0399	0.0919	0.0244	0.1386
G	0.1670	0.0126	0.0957	0.0766	0.0170	1.1503	0.0346	0.0182	0.0865	0.0173
H	0.0194	0.0031	0.0406	0.0486	0.0253	0.0129	0.4316	0.0118	0.0599	0.0168
I	0.0583	0.0196	0.0135	0.0399	0.1000	0.0079	0.0172	0.6955	0.0473	0.2905
K	0.0618	0.0009	0.0852	0.2037	0.0166	0.0326	0.0640	0.0282	1.2676	0.0343
L	0.0803	0.0166	0.0321	0.0645	0.2321	0.0117	0.0388	0.4441	0.0834	1.0022
M	0.0326	0.0109	0.0108	0.0279	0.0546	0.0068	0.0152	0.1106	0.0335	0.1511
N	0.0435	0.0092	0.1982	0.1090	0.0140	0.0555	0.1038	0.0232	0.1328	0.0180
P	0.0611	0.0009	0.0426	0.0637	0.0175	0.0219	0.0256	0.0189	0.0624	0.0250
Q	0.0498	0.0051	0.0673	0.2092	0.0121	0.0206	0.1006	0.0197	0.1906	0.0337
R	0.0450	0.0048	0.0463	0.1057	0.0167	0.0265	0.0847	0.0240	0.4181	0.0383
S	0.2358	0.0402	0.1147	0.1288	0.0323	0.0892	0.0650	0.0407	0.1292	0.0348
T	0.1214	0.0318	0.0700	0.0991	0.0302	0.0353	0.0384	0.0869	0.1221	0.0614
V	0.1464	0.0430	0.0260	0.0681	0.0845	0.0198	0.0197	0.5876	0.0733	0.2238
W	0.0040	0.0019	0.0036	0.0074	0.0440	0.0033	0.0095	0.0093	0.0099	0.0171
Y	0.0204	0.0092	0.0209	0.0269	0.2502	0.0084	0.1004	0.0359	0.0372	0.0448

residue	M	N	P	Q	R	S	T	V	W	Y	pseudocounts
A	0.1040	0.0908	0.1239	0.1502	0.0784	0.2978	0.1729	0.1747	0.0154	0.0423	0.0296
C	0.0277	0.0186	0.0071	0.0162	0.0120	0.0381	0.0357	0.0406	0.0109	0.0155	0.0089
D	0.0173	0.2266	0.0457	0.1085	0.0438	0.0860	0.0562	0.0154	0.0069	0.0256	0.0197
E	0.0372	0.1165	0.0610	0.3041	0.0920	0.0851	0.0706	0.0356	0.0114	0.0283	0.0219
F	0.1030	0.0224	0.0204	0.0279	0.0207	0.0287	0.0292	0.0626	0.1249	0.3584	0.0164
G	0.0295	0.1318	0.0518	0.0731	0.0541	0.1383	0.0613	0.0306	0.0155	0.0213	0.0314
H	0.0209	0.0854	0.0214	0.1138	0.0606	0.0348	0.0249	0.0112	0.0179	0.0938	0.0092
I	0.2345	0.0350	0.0271	0.0489	0.0321	0.0379	0.0881	0.4754	0.0213	0.0536	0.0233
K	0.0467	0.1337	0.0555	0.2642	0.3582	0.0798	0.0836	0.0373	0.0158	0.0379	0.0207
L	0.4788	0.0457	0.0490	0.1110	0.0721	0.0479	0.0884	0.2749	0.0717	0.1012	0.0344
M	0.2250	0.0193	0.0158	0.0543	0.0246	0.0234	0.0396	0.0715	0.0199	0.0279	0.0091
N	0.0308	0.7723	0.0338	0.1235	0.0775	0.1159	0.0877	0.0195	0.0162	0.0392	0.0157
P	0.0285	0.0396	1.3621	0.0588	0.0408	0.0643	0.0446	0.0285	0.0069	0.0185	0.0165
Q	0.0573	0.0901	0.0373	0.7495	0.1174	0.0623	0.0573	0.0259	0.0216	0.0281	0.0131
R	0.0435	0.0917	0.0438	0.1919	0.9917	0.0668	0.0565	0.0276	0.0270	0.0372	0.0191
S	0.0539	0.1876	0.0916	0.1369	0.0844	0.4823	0.2865	0.0524	0.0191	0.0465	0.0224
T	0.0864	0.1252	0.0600	0.1147	0.0678	0.2594	0.5980	0.1256	0.0129	0.0376	0.0209
V	0.1914	0.0377	0.0478	0.0721	0.0407	0.0581	0.1494	0.6894	0.0254	0.0580	0.0274
W	0.0159	0.0084	0.0033	0.0141	0.0110	0.0061	0.0056	0.0082	0.9543	0.0611	0.0054
Y	0.0397	0.0370	0.0159	0.0375	0.0300	0.0273	0.0240	0.0313	0.1278	0.8340	0.0128

Table 6.6: Substitution matrix and pseudocounts for regularizer using substitution matrix plus scaled counts plus pseudocounts (trained on $|s| = 0, 1, 2, 3$).

The feature alphabets have very few tuning parameters (one per alphabet), and so one would expect them not to do well relative to the pseudocount methods (20 parameters) or the substitution matrices (400 parameters). The excess entropy reported in Table 6.7 show the feature alphabets doing surprisingly well for having so few parameters.

The 8-alphabet set does quite well for the the samples sizes it was tuned for ($|s| = 1, 2$), but degrades rather rapidly for larger sample sizes, doing worse than zero-offsets by $|s| = 4$. The 4-alphabet and 5-alphabet sets do better than pseudocounts for $|s| = 1, 2, 3$, but, like the substitution matrix method, the feature alphabets continue to get further from the optimum regularizer as $|s|$ increases, while the pseudocount methods improve.

The results for hand-created feature alphabet sets is presented in Table 6.8. On the whole, these hand-created feature sets did not do as well as the automatically generated ones.

Although the tiny number of parameters for the feature alphabets makes them aesthetically appealing, their performance is not good enough to justify the effort of implementing them. Perhaps

	excess entropy for n alphabets							
$\|s\|$	1	2	3	4	5	6	7	8
0	0.12527	0.12527	0.12527	0.12527	0.12527	0.12527	0.12527	0.12527
1	0.20207	0.14679	0.12551	0.10224	0.09088	0.08432	0.07842	0.07138
2	0.19704	0.13503	0.11754	0.09424	0.08240	0.07554	0.07132	0.06506
3	0.19054	0.13383	0.12353	0.11017	0.10720	0.10868	0.11143	0.11268
4	0.18429	0.13401	0.12991	0.12637	0.13261	0.14280	0.15241	0.16127
5	0.18040	0.13591	0.13682	0.14165	0.15600	0.17392	0.18952	0.20512
full	0.16902	0.14783	0.17308	0.21292	0.26574	0.32444	0.37478	0.43235

Table 6.7: Excess entropy for feature alphabet regularizers optimized for $|s| = 1, 2$, applied to the full blocks database.

	excess entropy				
$\|s\|$	Smith	Taylor-4	Taylor-10	Taylor	Taylor-58
0	0.12527	0.12527	0.12527	0.12527	0.12527
1	0.12893	0.17180	0.10069	0.11627	0.11019
2	0.10643	0.14352	0.08221	0.09621	0.08696
3	0.13251	0.15269	0.13456	0.13175	0.14194
4	0.16284	0.16989	0.19322	0.17321	0.21009
5	0.19128	0.18937	0.24832	0.21299	0.27880
full	0.29335	0.29341	0.59231	0.44537	1.14868

Table 6.8: Excess entropy for hand-generated feature alphabet regularizers optimized for $|s| = 1, 2$, applied to the full blocks database. The first three alphabet sets are presented in Tables 3.3 through 3.5. The "Taylor" column contains an 11-alphabet set consisting of the fundamental sets in [Tay86], and the "Taylor-58" contains an alphabet for each feature set in [Tay86, Figure 5].

a different feature-based approach could work better.

6.7 Dirichlet mixtures

Dirichlet mixtures are clearly the luxury choice among regularizers. The need for computing Gamma functions in order to evaluate the regularizer makes them much more expensive to use than any of the other regularizers reviewed here. However, the excess entropy results in Tables 6.9 and 6.10 show that the mixtures do perform better than any other regularizer test, and may well be worth the extra computational cost in creating a profile or hidden Markov model.

The regularizers in the table (except for the 9-component one) were created by adding a single component to an initially empty mixture or by adding components to a previously created mixture, optimizing after each addition for $|s| = 1, 2$. The components were added using the greedy strategy described in Section 3.8. The 1-component mixture is just a set of pseudocounts, and so performs almost identically to the pseudocounts optimized for $|s| = 1$ or $|s| = 2$.

The 9-component mixture was provided by Kimmen Sjölander, and was optimized for a different function on the blocks database with all sequence weights equal. The optimization was to provide the best Bayesian prior for the set of observed count vectors [BHK$^+$93]. Sjölander's 9-component mixture is the best we have for $|s| = 5$, but it does fairly poorly for $|s| = 0, 1, 2$.

The overall best regularizer is the 21-component Dirichlet mixture, which gets within 0.027 bits of the best possible regularizer for sample sizes up to 5, and probably never takes more than 0.09 bits more than the optimum regularizer.

	excess entropy for n components							
	1	3	4	6	7	9	10	10
$\|s\|$	1	1+2	1+3	1+2+3	1+2+4	9	1+2+3+4	1+3+6
0	0.02230	0.02488	0.03134	0.02516	0.02694	0.06123	0.02385	0.01774
1	0.13437	0.05235	0.04026	0.02336	0.01972	0.05336	0.01115	0.00661
2	0.13524	0.07353	0.05311	0.03275	0.02678	0.02402	0.02290	0.01610
3	0.12930	0.08302	0.05933	0.03960	0.03301	0.01970	0.03127	0.02520
4	0.12206	0.08657	0.06200	0.04367	0.03715	0.02083	0.03620	0.03105
5	0.11676	0.08886	0.06455	0.04762	0.04119	0.02455	0.04066	0.03624
full	0.08308	0.07986	0.08234	0.08365	0.08837	0.10274	0.08607	0.08992

Table 6.9: Excess entropy for small Dirichlet mixtures regularizers optimized for $|s| = 1, 2$, applied to the full blocks database. The mixtures were built by adding new components to a previous mixture, except for for the nine-component mixture, which was provided by Kimmen Sjölander.

	excess entropy for n components						
	15	15	20	21	28	31	35
$\|s\|$	1+2+3+4+5	1+2+4+8	1+3+6+10	1+2+3+4+5+6	1+2+3+4+5+6+7	1+2+4+8+16	1+3+6+10+15
0	0.01989	0.01040	0.01111	0.00883	0.00832	0.00786	0.00812
1	0.00192	0.00227	0.00169	0.00115	0.00470	0.00198	0.00578
2	0.01137	0.01002	0.01003	0.00757	0.01750	0.00764	0.02100
3	0.01987	0.01958	0.01957	0.01471	0.02740	0.01776	0.03863
4	0.02608	0.02613	0.02653	0.02051	0.03380	0.02479	0.04903
5	0.03186	0.03199	0.03286	0.02636	0.03943	0.03092	0.05650
full	0.08603	0.09155	0.08715	0.08589	0.09357	0.09474	0.10174

Table 6.10: Excess entropy for larger Dirichlet mixtures regularizers optimized for $|s| = 1, 2$, applied to the full blocks database. The mixtures were built by adding new components to a previous mixture, with the history of the additions shown in the name. The 21-component mixture 1+2+3+4+5+6 is the best overall regularizer for the BLOCKS database.

7 Results for separate training and testing

To make sure that the results in Section 6 were not training on noise in the data, but were picking up phenomena that should generalize, regularizers were created using the same methods on a subset of the data, and tested on a disjoint subset.

The blocks database was divided into three disjoint sets, with about 10% of the blocks in set 10a, 10% in 10b, and the remaining 80% in 80c. Regularizers were created separately for each of the 3 sets, and tested on the other two. The ordering of the methods produced by these tests was almost identical to the ordering produced by the self-test presented in Section 6. This separate train-test evaluation lends some extra confidence to the comparative evaluation of the regularizers, but little new information, and so will not be presented in detail here.

8 Conclusions and future research

For applications that can afford the computing cost of the Dirichlet mixture regularizers, they are clearly the best choice. In fact they are so close to the theoretical optimum for regularizers, that there doesn't seem to be much point in looking for better regularizers. The evaluations of regularizers for searches in biological contexts have also found Dirichlet mixtures to be superior [TAK94, HH95], validating the more information-theoretic approach taken here.

Although most applications (such as training hidden Markov models or building profiles from multiple alignments) do not require frequent evaluation of regularizers, there are some applications (such as Gibbs sampling) that require recomputing the regularizers inside an inner loop. For these applications, the substitution matrix plus pseudocounts plus scaled counts is probably the best choice, as it has only about 0.03 bits more excess entropy than the Dirichlet mixtures, but does not require evaluating Gamma functions.

For applications in which there is little data to train a regularizer, the pseudocounts are probably the best choice, as they perform reasonably well with few parameters. If you have enough data to train a substitution matrix technique, then you should have enough data to train a Dirichlet mixture, as they have comparable numbers of parameters.

One weakness of the empirical analysis done in this report is that all the data was taken from the BLOCKS database, which contains only highly conserved blocks. While this leads us to have high confidence in the alignment, it also means that the regularizers do not have to do much work. The appropriate regularizers for more variable columns may look somewhat different, though one would expect the pseudocount and substitution matrix methods to degrade more than the Dirichlet mixtures, which naturally handle high variability. I plan to build regularizers for the HSSP structural alignments [SS91] to check that Dirichlet mixtures are the most effective in that application as well.

To get significantly better performance than a Dirichlet mixture regularizer, we have to step away from using a pure regularizer that only knows about the sample of amino acids seen in the context. There are at least two ways to do this. One uses other information about the column (such as solvent accessibility or secondary structure) and the other uses other information about the sequence (such as a phylogenetic tree relating it to other sequences).

Using extra information about a column could improve the performance of a regularizer up to the "full" row shown in Table 2.1, but no more, since that entropy reflects the best we could do if the extra information uniquely identified the column. There is about 0.6 bits that could be gained by using such information (relative to a sample size of 5), far more than difference between the best regularizer and a crude zero-offset regularizer.

One possible way to use such column information would be to classify each column with one of a small number of labels, and to tune a different regularizer for each label. For this application, pseudocount regularizers are probably most appropriate, both because the labeling will reduce the size of the training set, and because a good labeling should provide fairly pure distributions that shouldn't need the ability of Dirichlet mixtures to match a variety of different distributions. I plan to pursue creating such a collection of regularizers in spring and summer 1995.

Using sequence-specific information may yield even larger gains than using column-specific information. Preliminary investigations at UCSC indicate that there may be a full bit per column to be gained by taking into account phylogenetic tree relationships among sequences in a multiple alignment. Even if phylogenetic tree data is not available, sequence distance information may be useful.

Another way to use sequence-specific information is to use modified regularizers for residues that are in contact, adjusting the probabilities for one amino acid based on what is present in the contacting position. I hope to work on this approach in summer 1995 as well.

Acknowledgements

I'd like to thank Leslie Grate, who wrote an early version of the program used to test regularizers; Kimmen Sjölander for providing the 9-component Dirichlet mixture; Michael Brown for providing an implementation of the Dirichlet mixture regularizer (later replaced with a more efficient implementation); the Henikoffs for providing the BLOCKS database, pre-publication drafts of their papers, and comments on early drafts of this paper; and everyone in the computational biology group at UCSC.

References

[ACL89] Stephen F. Altschul, Raymond J. Carroll, and David J. Lipman. Weights for data related by a tree. *JMB*, 207:647–653, 1989.

[Alt91] Stephen F. Altschul. Amino acid substitution matrices from an information theoretic perspective. *JMB*, 219:555–565, 1991.

[BCHM94] P. Baldi, Y. Chauvin, T. Hunkapillar, and M. McClure. Hidden Markov models of biological primary sequence information. *PNAS*, 91:1059–1063, 1994.

[BHK$^+$93] M. P. Brown, R. Hughey, A. Krogh, I. S. Mian, K. Sjölander, and D. Haussler. Using Dirichlet mixture priors to derive hidden Markov models for protein families. In L. Hunter, D. Searls, and J. Shavlik, editors, *ISMB-93*, pages 47–55, Menlo Park, CA, July 1993. AAAI/MIT Press.

[Cla94] Jean-Michael Claverie. Some useful statistical properties of position-weight matrices. *Computers and Chemistry*, 18(3):287–294, 1994.

[DSO78] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. In *Atlas of Protein Sequence and Structure*, chapter 22, pages 345–358. National Biomedical Research Foundation, Washington, D. C., 1978.

[GME87] Michael Gribskov, Andrew D. McLachlan, and David Eisenberg. Profile analysis: Detection of distantly related proteins. *PNAS*, 84:4355–4358, July 1987.

[GR65] I. S. Gradshteyn and I. M. Ryzhik. *Table of Integrals, Series, and Products.* Academic Press, fourth edition, 1965.

[HH91] Steven Henikoff and Jorja G. Henikoff. Automated assembly of protein blocks for database searching. *NAR*, 19(23):6565–6572, 1991.

[HH92] Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices from protein blocks. *PNAS*, 89:10915–10919, November 1992.

[HH94] Steven Henikoff and Jorja G. Henikoff. Position-based sequence weights. *JMB*, 243(4):574–578, November 1994.

[HH95] Steven Henikoff and Jorja G. Henikoff. Personal communication, January 1995.

[Hig92] Desmond G. Higgins. Sequence ordinations: a multivariate analysis approach to analysing large sequence data sets. *CABIOS*, 8(1):15–22, 1992.

[JTT92] David T. Jones, William R. Taylor, and Janet M. Thornton. The rapid generation of mutation data matrices from protein sequences. *CABIOS*, 8(3):275–282, 1992.

[KBM$^+$94] 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.

[LAB$^+$93] C. E. Lawrence, S. Altschul, M. Boguski, J. Liu, A. Neuwald, and J. Wootton. Detecting subtle sequence signals: A Gibbs sampling strategy for multiple alignment. *Science*, 262:208–214, 1993.

[Mil93] Aleksandar Milosavljević. Discovering sequence similarity by the algorithmic significance method. In *ISMB-93*, pages 284–291, Menlo Park, 1993.

[SS90] Randall F. Smith and Temple F. Smith. Automatic generation of primary sequence patterns form sets of related protein sequences. *PNAS*, 87:118–122, January 1990.

[SS91] C. Sander and R. Schneider. Database of homology-derived protein structures and the structural meaning of sequence alignment. *Proteins*, 9(1):56–68, 1991.

[TAK94] 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.

[Tay86] William Ramsay Taylor. The classification of amino acid conservation. *Journal of Theoretical Biology*, 119:205–218, 1986.

[THG94] Julie D. Thompson, Desmond G. Higgins, and Toby J. Gibson. Improved sensitivity of profile searches through the use of sequence weights and gap excision. *CABIOS*, 10(1):19–29, 1994.

[TJ93] William R. Taylor and David T. Jones. Deriving an amino acid distance matrix. *Journal of Theoretical Biology*, 164:65–83, 1993.

A Partial derivatives for Dirichlet mixtures

The derivative of the Beta function is

$$\frac{\partial \mathrm{B}(z_c)}{\partial z_c(i)} = \mathrm{B}(z_c) \left(\frac{?'(z_c(i))}{?(z_c(i))} - \frac{?'(|z_c|))}{?(|z_c|)} \right) .$$

If we introduce the variable R to be the expression

$$\left(\frac{?'(z_c(n)+s(n))}{?(z_c(n)+s(n))} - \frac{?'(|z_c+s|)}{?(|z_c+s|)} - \frac{?'(z_c(n))}{?(z_c(n))} + \frac{?'(|z_c|)}{?(|z_c|)} \right) ,$$

then the derivative of $X_s(i)$ is

$$\frac{\partial X_s(i)}{\partial z_c(n)} = q_c \frac{\mathrm{B}(z_c+s)}{\mathrm{B}(z_c)} \left((z_c(i)+s(i))R + \delta(i-n) \right) ,$$

where $\delta(i-n)$ is 1 if $i=n$ and 0 otherwise.

Using Stirling's approximation

$$?(x+1) \approx \sqrt{2\pi x}(x/e)^x (1 + 1/12x^{-1} + O(x^{-2}))$$

lets us approximate the derivative of $\ln ?(x)$

$$\frac{?'(x)}{?(x)} \approx \frac{-1}{2x} + \ln x - \frac{1}{12x^2+x} .$$

Since Stirling's approximation is not good for small x, we may have to use

$$?(x) = \frac{?(x+n+1)}{x(x+1)(x+2)\cdots(x+n)}$$

to move the value of the argument up into a region where the approximation is adequate:

$$\frac{?'(x)}{?(x)} \approx -\frac{1}{x} - \frac{1}{x+1} - \frac{1}{x+2} - \cdots - \frac{1}{x+n-1} - \frac{1}{2(x+n)} + \ln(x+n) - \frac{1}{12(x+n)^2+x+n} .$$

Second partial derivatives are easily computed from the first partials.

B Bayesian interpretation of pseudocount regularizers

We can use Bayesian probability techniques to interpret the pseudocount regularizers. To apply these methods we have to view amino acids as being generated by a two-stage random process. First, a 20-dimensional density vector ρ over the amino acids is chosen randomly, then amino acids are chosen randomly with probabilities $P(i) = \rho(i)$. The probability of amino acid i given a sample s is the integral over all possible vectors ρ of the probability of choosing that vector times the probability of choosing i given that vector:

$$P_s(i) = P(i|s) = \int P(\rho|s)\rho(i)\, d\rho .$$

Computing the probability $P(\rho|s)$ requires applying Bayes' rule:

$$P(\rho|s) = P(\rho,s)/P(s) = P(s|\rho)P(\rho)/P(s) ,$$

giving us a new formula for the probability of amino acid i:

$$P_s(i) = \frac{1}{P(s)} \int \rho(i) P(s|\rho) P(\rho)\, d\rho .$$

The probability $P(s|\rho)$ is easily computed for any density vector ρ, but we need to know the prior distribution of ρ in order to compute the integral. The computation for $P(s|\rho)$ is the same as in Section 2.1:

$$P(s|\rho) = |s|! \prod_j \frac{\rho(j)^{s(j)}}{s(j)!} \ .$$

There is an obvious generalization to non-integer $s(j)$ values by replacing the factorial function with the equivalent expression using the Gamma function:

$$P(s|\rho) = ?(|s|+1) \prod_j \frac{\rho(j)^{s(j)}}{?(s(j)+1)} \ .$$

In order to compute the integral, we must choose a model for the the prior distribution of ρ. One choice that allows us to compute the integral is to model the prior as a Dirichlet distribution, that is

$$\hat{P}(\rho) = \prod_j \rho(j)^{z(j)-1}/C \ ,$$

for some parameter vector z, where C is a constant chosen so that $\int \hat{P}(\rho)\, d\rho = 1$.

Showing in detail how to compute the integral is beyond the scope of this paper, but the answer can be derived from the standard definition of the Beta function [GR65, p. 948]

$$\begin{aligned} \mathrm{B}(x,y) &= \int_0^1 t^{x-1}(1-t)^{y-1}\, dt \\ &= \frac{?(x)?(y)}{?(x+y)} \end{aligned}$$

and the combining formula [GR65, p. 285]:

$$\int_0^b t^{x-1}(b-t)^{y-1}\, dt = b^{x+y-1}\mathrm{B}(x,y) \ .$$

By writing the integral over all ρ vectors as a multiple integral over the 20 dimensions of the vector and doing some rearrangement, we can get the solution

$$\begin{aligned} C &= \int \prod_j \rho(j)^{z(j)-1}\, d\rho \\ &= \mathrm{B}(z(1), z(2)+\cdots+z(20))\mathrm{B}(z(2), z(3)+\cdots+z(20))\cdots\mathrm{B}(z(19), z(20) \\ &= \frac{\prod_j ?(z(j))}{?(|z|)} \\ &= \mathrm{B}(z) \ , \end{aligned}$$

where we have introduced the $\mathrm{B}(z)$ notation as an simple generalization of $\mathrm{B}(x,y)$ to the vector argument z.

With this choice of prior distribution for ρ, we can compute

$$\begin{aligned} \hat{P}(\rho, s) &= P(s|\rho)\hat{P}(\rho) \\ &= \frac{?(|s|+1)}{C} \prod_j \frac{\rho(j)^{s(j)+z(j)-1}}{?(s(j)+1)} \\ &= \frac{?(|s|+1|)?(|z|)\prod_j \rho(j)^{s(j)+z(j)-1}}{\prod_j ?(s(j)+1)?(z(j))} \ . \end{aligned}$$

We can now compute the estimated probability of the sample

$$\begin{aligned}
\hat{P}(s) &= \int \hat{P}(\rho, s)\, d\rho \\
&= \int P(s|\rho)\hat{P}(\rho)\, d\rho \\
&= \frac{?(|s|+1|)?(|z|)}{\prod_j ?(s(j)+1)?(z(j))} \int \prod_j \rho(j)^{s(j)+z(j)-1}\, d\rho \\
&= \frac{?(|s|+1|)?(|z|)}{\prod_j ?(s(j)+1)?(z(j))} \frac{\prod_j ?(z(j)+s(j))}{?(|z+s|)} \\
&= \frac{?(|s|+1|)?(|z|)}{?(|z+s|)} \prod_j \frac{?(z(j)+s(j))}{?(s(j)+1)?(z(j))} \\
&= \frac{?(|s|+1|)}{\prod_j ?(s(j)+1)} \frac{\mathrm{B}(z+s)}{\mathrm{B}(z)}\ .
\end{aligned}$$

The integral for estimating the conditional probability of amino acid i given sample s is then

$$\begin{aligned}
\hat{P}_s(i) &= \hat{P}(i|s) \\
&= \hat{P}(i,s)/\hat{P}(s) \\
&= \frac{1}{\hat{P}(s)} \int \hat{P}(i,s,\rho)\, d\rho \\
&= \frac{1}{\hat{P}(s)} \int \rho(i) P(s|\rho)\hat{P}(\rho)\, d\rho \\
&= \frac{\mathrm{B}(z)\prod_j ?(s(j)+1)}{\mathrm{B}(z+s)?(|s|+1|)} \frac{?(|s|+1|)}{\mathrm{B}(z)\prod_j ?(s(j)+1)} \int \rho(i) \prod_j \rho(j)^{s(j)+z(j)-1}\, d\rho \\
&= \frac{\mathrm{B}(z+s+\delta_i)}{\mathrm{B}(z+s)} \\
&= \frac{?(|z+s|)}{\prod_j ?(z(j)+s(j))} \frac{\prod_j ?(z(j)+s(j)+\delta_{i,j})}{?(|z+s|+1)} \\
&= \frac{z(i)+s(i)}{|z+s|}\ .
\end{aligned}$$

Notation: δ_i is used above to mean the vector consisting of a one in the ith position and a zero elsewhere. $\delta_{i,j}$ is one if $i=j$ and zero otherwise.

This rather involved computation finally ends up with the pseudocount method for estimating the probability of an amino acid given a sample of amino acids. The regularizer parameters z can be interpreted as assuming a Dirichlet distribution for the prior probabilities $P(\rho)$. Previous work with pseudocounts has relied heavily on this Bayesian interpretation of the parameters, going so far as to assign $z(i) = \alpha P_0(i)$, which does indeed provide the optimal estimates for $\hat{P}_0(i)$, but which we have seen in Section 3.2 is not the best setting of the parameters for $|s| > 0$.

The *posterior distribution* of ρ after seeing a sample s is $\hat{P}(\rho|s) = P(s|\rho)\hat{P}(\rho)/\hat{P}(s)$. As we can see from the above computations, this posterior distribution is again a Dirichlet distribution, with parameters $s(j)+z(j)$, instead of the prior distribution's parameters $z(j)$. This interpretation of $X_s(j)$ as the parameters of the posterior distribution is what inspired naming them the *posterior counts*. The scaling of X_s does matter for this interpretation, and so not all the posterior counts produced by regularizers can be automatically interpreted as Dirichlet posterior distributions on ρ.

We can extend the Bayesian analysis to compute the posterior distribution of ρ given that we have seen several independent samples: $\hat{P}(\rho|s_1, s_2, \ldots, s_n)$. The computation is fairly straightforward. First we apply Bayes rule:

$$\begin{aligned}
\hat{P}(\rho|s_1, s_2, \ldots, s_n) &= \hat{P}(\rho, s_1, s_2, \ldots, s_n)/\hat{P}(s_1, s_2, \ldots, s_n) \\
&= \hat{P}(\rho)P(s_1, s_2, \ldots, s_n|\rho)/\hat{P}(s_1, s_2, \ldots, s_n) \\
&= \hat{P}(\rho) \prod_{1\leq k\leq n} \left(\frac{P(s_k|\rho)}{\hat{P}(s_k)}\right) .
\end{aligned}$$

Repeating the mathematics for a single sample would be tedious, but we can take a shortcut. Since the posterior distribution after seeing a sample is again a Dirichlet distribution, we can treat it as the prior distribution for adding the next sample. Using this trick, we can see that the final posterior distribution after seeing all n samples is a Dirichlet distribution with parameters $z(i) + s_1(i) + \cdots + s_n(i)$. In other words, we get the same result from observing n independent samples as we would get from adding all the samples together and using the resulting counts as a single sample.

C Bayesian interpretation of Dirichlet mixture regularizers

The Dirichlet mixture regularizers can also be interpreted using Bayes rule, in a manner very similar to that used for interpreting pseudocounts. The main difference comes in how we model the prior distribution of ρ. A mixture distribution consists of a number of component distributions. We can regard the process as adding one more step to the random selection—first we select a component, then a density vector ρ from the component, and finally amino acids from the density.

If we use the letter c to designate a component, we can write a mixture distribution as

$$\hat{P}(\rho) = \sum_c \hat{P}(\rho|c)\hat{P}(c)$$

Simple application of Bayes' rule to the definition of $\hat{P}(s)$ gives us

$$\begin{aligned}
\hat{P}(s) &= \int P(s, \rho)\, d\rho \\
&= \int P(s|\rho)\hat{P}(\rho)\, d\rho \\
&= \sum_c \hat{P}(c) \int P(s|\rho)\hat{P}(\rho|c)\, d\rho .
\end{aligned}$$

If each component of the mixture is modeled as a Dirichlet distribution, we get a *Dirichlet mixture distribution*:

$$\hat{P}(\rho) = \sum_c \frac{q_c}{\mathrm{B}(z_c)} \prod_j \rho(j)^{z(j)_c - 1} ,$$

where the mixture coefficients q_c must sum to one ($q_c = \hat{P}(c)$).

Using our general formulas for mixture distributions, we have

$$\begin{aligned}
\hat{P}(\rho, s) &= \sum_c q_c P(s|\rho)\hat{P}(\rho|c) \\
&= \frac{?(|s|+1|)}{\prod_j ?(s(j)+1)} \sum_c \frac{q_c}{\mathrm{B}(z_c)} \prod_j \rho(j)^{s(j)+z_c(j)-1} \\
\hat{P}(s) &= \int \hat{P}(\rho, s)\, d\rho \\
&= \frac{?(|s|+1)}{\prod_j ?(s(j)+1)} \sum_c q_c \frac{\mathrm{B}(z_c+s)}{\mathrm{B}(z_c)}
\end{aligned}$$

$$\begin{aligned}
\hat{P}_s(i) &= \int P(i|\rho)P(\rho|s)\,d\rho \\
&= \frac{1}{\hat{P}(s)}\sum_c q_c \int P(i|\rho)P(s|\rho)\hat{P}(\rho|c)\,d\rho \\
&= \frac{1}{\hat{P}(s)}\sum_c q_c \frac{?(|s|+1)}{\prod_j ?(s(j)+1)} \frac{\mathrm{B}(z_c+s+\delta_i)}{\mathrm{B}(z_c)} \\
&= \frac{\sum_c q_c \frac{\mathrm{B}(z_c+s+\delta_i)}{\mathrm{B}(z_c)}}{\sum_c q_c \frac{\mathrm{B}(z_c+s)}{\mathrm{B}(z_c)}} \\
&= \frac{\sum_c q_c \frac{z_c(i)+s(i)}{|z_c+s|} \frac{\mathrm{B}(z_c+s)}{\mathrm{B}(z_c)}}{\sum_c q_c \frac{\mathrm{B}(z_c+s)}{\mathrm{B}(z_c)}} \,.
\end{aligned}$$

Unfortunately, unlike the single Dirichlet case, the summations preclude easy cancellation of the ? or Beta functions.

The posterior distribution for ρ is a Dirichlet mixture with components $z_c + s$, but the mixture coefficients change in a rather complex way. The posterior mixture coefficient for the $z_d + s$ component is

$$\frac{q_d \mathrm{B}(z_d+s)}{\mathrm{B}(z_d)} \frac{1}{\sum_c q_c \frac{\mathrm{B}(z_c+s)}{\mathrm{B}(z_c)}} \,.$$

Note that if we just need a set of posterior counts without interpreting them as a posterior Dirichlet mixture distribution (say for computing scores in a profile or hidden Markov model), we can eliminate the scaling factors, getting

$$X_s(i) \leftarrow \sum_{1\leq c\leq k} q_c \frac{\mathrm{B}(z_c+s)}{\mathrm{B}(z_c)} (z_c(i)+s(i)) \,,$$

exactly as used in Section 3.8. The normalization of the counts to make estimated probabilities takes care of whatever scaling we need.

Because the posterior distribution is again a mixture of Dirichlet distributions, we can combine multiple observations in the same way as we do for pseudocounts. The effect is again identical to adding all the observation vectors together, and using the single combined observation vector for s. This is the correct posterior distribution if we assume that one ρ was chosen, and all samples were taken from that ρ. The more interesting assumption is that a separate ρ is chosen from the prior distribution for each sample. With this assumption, the posterior distribution given n samples $s_1, \ldots, s_n$ and an m-component Dirichlet mixture prior is an mn-component Dirichlet mixture, with components $z_c + s_j$ and mixture coefficients

$$\frac{q_c \mathrm{B}(z_c+s_j)}{C\mathrm{B}(z_c)} \,,$$

where C is the appropriate normalizing constant so that the mixture coefficients sum to one.