Gene Profiling using Suffix Trie Hidden Markov Models

Bernhard K. Bauer
bauerb@in.tum.de

December 6, 2009

Contents

1 Motivation

With the advent of cost-effective high-throughput sequencing methods, the amount of sequenced biopolymer data such as DNA, RNA or amino acid sequences has grown in an exponential way. The EMBL Nucleotide Sequence Database[1] contains over 150 millions of entries for a variety of organisms ranging from viruses and procaryotes to high mammals. However, this abundance of data brings with itself several problems:

First, the huge datasets available to researchers necessitate efficient databases and search methods to find relevant entries among the ever-growing sea of data. Second, the growth in available data *quantity* is often accompanied by a decrease in data *quality*, which manifest itself not only in questionable data, but also in faulty metadata brought about by human error in annotating the data.

As an example, take phylogenetic analysis, which is based on comparing homologuous sequences like rRNA segments or certain genes from different species or strains, creating phylogenetic trees which capture ancestral information. For this, it necessary to find the target sequence in every species involved in the phylogenetic analysis, using for example the EMBL database, which offers genome files for a multitude of organisms in a plain text format. Besides the genome DNA sequence, an EMBL file contains annotations describing "*coding sequences*", which are subsequences coding a gene each. While the genes themselves can be found automatically, the descriptions of the genes and genomes are written by humans, who commit errors or label genes in a non-standard way, therefore making description searches for a specific gene ineffective. Sometimes, annotations are not even present, although it is certain from a biological standpoint that the organism in question possesses the searched for gene. In case a direct metadata search is not successful, more complicated methods can be used to look for the target gene.

The approach described herein constructs a stochastic sequence model from already found gene sequences and uses this model to look for similar sequences either in the whole genome or among the coding sequences defined for it. The model is called a "suffix trie hidden Markov model", utilizing a hidden Markov model based on a suffix trie of the sequences seen so far. The stochastic nature of the model enables it to find a meaningful and stochastically sound evaluation of the relevance of a sequence as well as the reliability of the search results.

[1] `http://www.ebi.ac.uk/embl/`

2 Other sequence search methods

2.1 Sequence alignment methods

The Needleman-Wunsch and Smith-Waterman [15] algorithms calculate optimal global and local alignments, respectively, using a dynamic programming approach. In particular local alignments can be used to search for a subsequence similar to a given sequence in a database. However, these algorithms prove prohibitively expensive for large databases, taking $O(n \cdot m)$ space (and time), where n and m are the length of the two sequences (or rather the length of the search sequence and the combined length of all sequences in the database, respectively).

The BLAST algorithm [2] is a heuristic for calculating local alignments, running much faster, while possibly yielding sub-optimal results.

A big disadvantage of these single sequence alignment methods is that they only deliver results for a single search query. Given multiple variants of a sequence, the search would have to carried out multiple times, with no obvious way of combining the search results. Nevertheless, the statistics used in BLAST turn out to be also applicable to this work.

2.2 HMMER

The HMMER software package[1] implements a profile hidden Markov model [9] constructed from known sequences to search sequence databases.

While the work described here uses the same general approach (albeit with a different model), the biggest drawback of HMMER is its reliance of multiple sequence alignments (MSA) to construct its profiles. Because multiple sequence alignment in general is NP-hard [18], again heuristics have to be employed first to construct the MSA. However, starting with a suboptimal alignment can in turn yield suboptimal search results. On the other hand, constructing a better alignments also takes more time.

[1] `http://hmmer.janelia.org/`

3 Suffix trie HMMs

3.1 Hidden Markov models

A hidden Markov model (HMM) consists of a discrete Markov process, i.e. a sequence of discrete random variables where each variable (the *state*) only depends on the previous one, and a sequence of *observation* or *output* variables that each depend only on the state in the corresponding time step. Figure 3.1 illustrates this with a Bayesian Network.

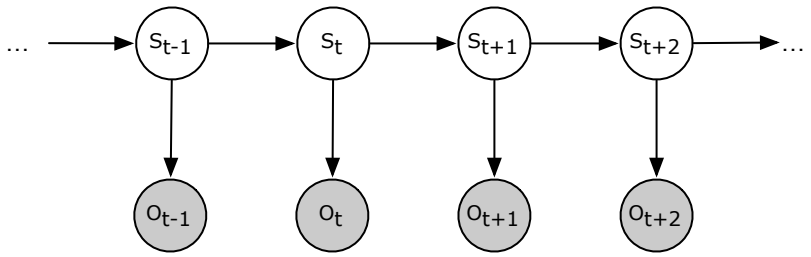

Figure 3.1: A hidden Markov model. Shaded circles indicate observed variables, unshaded circles hidden ones.

Because of the independence properties of an HMM, exact inference can be carried out in an online manner, where every observation needs to be processed only once, thereby taking time linear in the sequence length and constant space for a constant model. To achieve this, we use a vector $\boldsymbol{\alpha}$ called the *forward message*, which is updated for every time step with the respective observation. The vector $\boldsymbol{\alpha}$ represents the probability for being in a certain state and observing the given output character, given the sequence of characters that has been observed so far:

$$\alpha(t)_i := P_{S_t,O_t|\boldsymbol{O}_{1,\ldots,t-1}}(i, o_t|\boldsymbol{o}_{1,\ldots,t-1}), \quad 1 \leq i \leq k, \tag{3.1}$$

where k is the number of states, n is the sequence length, and we define the sequence of output variables as

$$\boldsymbol{O}_{i,\ldots,j} := (O_i, O_{i+1}, \ldots, O_j), \quad 1 \leq i \leq j \leq n$$

and the sequence of actually observed characters as

$$\boldsymbol{o}_{i,\ldots,j} := (o_i, o_{i+1}, \ldots, o_j), \quad 1 \leq i \leq j \leq n.$$

For practical inference, we use a scaled version $\hat{\boldsymbol{\alpha}}(t)$, which represents the conditional probability for each state, obtained by dividing $\boldsymbol{\alpha}$ element-wise by the conditional probability $c(t)$ of the observed output, which is in turn calculated by marginalizing $\boldsymbol{\alpha}$ over all states:

$$c(t) := P_{O_t|\boldsymbol{O}_{1,\dots,t-1}}(o_t|\boldsymbol{o}_{1,\dots,t-1}) = \sum_{j=1}^{k} \alpha(t)_k \tag{3.2}$$

$$\hat{\alpha}(t)_i := P_{S_t|\boldsymbol{O}_{1,\dots,t}}(i|\boldsymbol{o}_{1,\dots,t}) = \frac{\alpha(t)_i}{c(t)} = \frac{\alpha(t)_i}{\sum_{j=1}^{k} \alpha(t)_j}, \quad 1 \leq i \leq k \tag{3.3}$$

Likewise, for backwards inference we use a *backwards message* $\boldsymbol{\beta}(t)$ and its scaled version $\hat{\boldsymbol{\beta}}(t)$:

$$\beta(t)_i := P_{S_t|\boldsymbol{O}_{t,\dots,n}}(i|\boldsymbol{o}_{t,\dots,n}), \quad 1 \leq i \leq k \tag{3.4}$$

If we define the output probability as $b_i(\boldsymbol{o}) := P_{O_t|S_t}(\boldsymbol{o}|i)$ and the state transition probability as $a_{ij} := P_{S_{t+1}|S_t}(i|j)$, we can update $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ as follows:

$$\hat{\alpha}(t+1)_i = \frac{\left(\sum_{j=1}^{k} \hat{\alpha}(t)_j \cdot a_{j,i}\right) \cdot b_i(o_t)}{c(t)}, \tag{3.5}$$

$$\hat{\beta}(t)_i = \frac{\left(\sum_{j=1}^{k} \hat{\beta}(t+1)_j \cdot a_{i,j}\right) \cdot b_i(o_{t+1})}{c(t)}, \quad 1 \leq i \leq k \tag{3.6}$$

The total probability of the output sequence is given by the product of all values of $c(t)$:

$$P_{\boldsymbol{O}}(s) = \prod_{t=1}^{n} c(t) \tag{3.7}$$

A good and deeper introduction to hidden Markov models can be found in [13].

3.2 HMM structure

State transition model. The state transition model for the hidden state is based on a suffix trie [6]. Suffix tries store the prefixes of a string or a set of strings in a tree structure, with every leaf of the tree corresponding to one suffix and every inner node corresponding to a prefix of a suffix, i.e. a substring.

An inner node in a trie, corresponding to a substring v, has links to every substring stored in the trie that is obtained by appending a single character to v. In addition to these forward links, every node has a link to its direct suffix, which is neccessary for efficient trie construction [17] and searching.

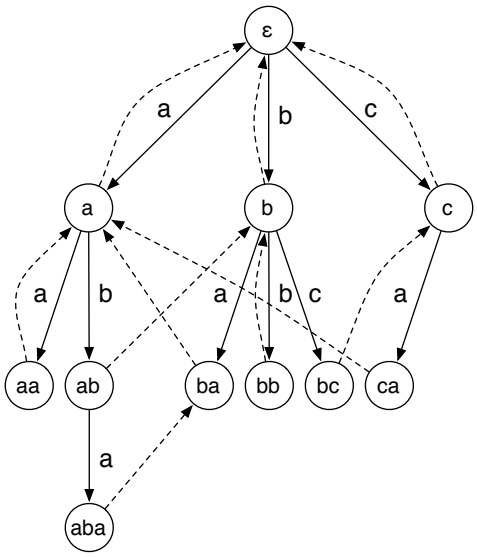

Figure 3.2: A suffix trie.

Based on the trie for the set of read sequences, we construct a state transition model with every node in the trie corresponds to a state, and a transition probability assigned to every outgoing link from a state. There are two variants possible for transitions to a state of lower depth in the trie:

1. Every state has a transition to the root state or
2. Every state has a transition to its direct suffix.

In practice, only the second variant was implemented, because by following the back-links, one can always arrive at the root state, and so intuitively the second model seems to be more powerful than the first one.

In contrast to a classical HMM, we allow an arbitrary number of suffix transitions followed by one forward transition per time step.

Output model. The output model is deterministic, with the output character for every state equal to the last character of its corresponding substring. This determinism is what allows us to keep the number of possible states at each time step small and so allows for efficient inference.

3.3 Inference

Forward pass. Using a dense representation of the state distribution vector, which simply lists the probablity for every state in the model, would make inference intractable.

Therefore we use a sparse representation, storing a list of those states that have a positive probability, along with their corresponding probability. Because of the deterministic output model, we know that at every time step, the only possible states are suffixes of the word read so far (because every other state would have an output probability of zero at some point). This means that the space needed for storing a state distribution is at worst linear in the length of the word read so far and bounded by the maximum depth of the trie.

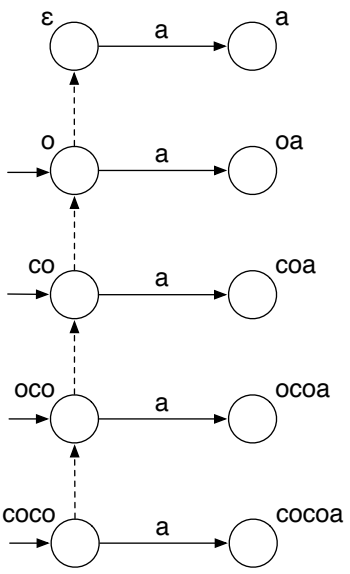

Figure 3.3: State transitions for the forward inference step.

In addition, we can use the same inference algorithm to efficiently calculate all transition probabilities from a state at time t to some state at time $t + 1$ in a single pass over the state list. To do this, we note that the probability for each state is the sum of the probabilities for each incoming transition times the probability for the respective predecessor state:

$$P(s) = \sum_{s'} P(s') \cdot P(s' \to s)$$

As illustrated by figure 3.3, we can calculate the probability including backlinks for each in the left column in turn, multiplying it by the probability for the character transition to get the probabilities for the states in the right column.

Backwards pass. Inference in the backwards pass seems more difficult at first, because while there is only a limited number of transitions *out of* a state, there is a much bigger number of transitions going *into* most states, because of the suffix links and the fact that a transition can encompass more than one of them at a time. Fortunately, it turns out

that we can use the same line of reasoning as for the forward pass to only track prefixes of the string read so far.

3.4 Model construction

3.4.1 Parameter estimation

For constructing the model parameters we use an online version of the standard Baum-Welch algorithm, i.e. one that only looks at each sequence once. Put simply, the classical Baum-Welch algorithm counts the expected state transitions using the forward and backwards messages $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ (see section 3.1) based on a starting set of parameters, and updates the parameters with these counts, setting each transition probability to the relative expected transition frequency. This is then repeated until the transition probabilities converge.

To create an online version of this algorithm, we simply use the intermediate transition counts after each sequence to calculate the probabilities for the next sequence. This is similar to using stochastical instead of batch updating for backpropagation in neural networks.

3.4.2 Suffix trie construction

So far, we have only concerned ourselves with calculating the model parameters for a given trie structure. Now we look at constructing the trie itself. For theoretical purposes, we assume an infinite suffix trie with some transition probabilities set to zero, as seen in figure 3.4. Of course, in practice we only need to materialize nodes that can be reached with a positive probability.

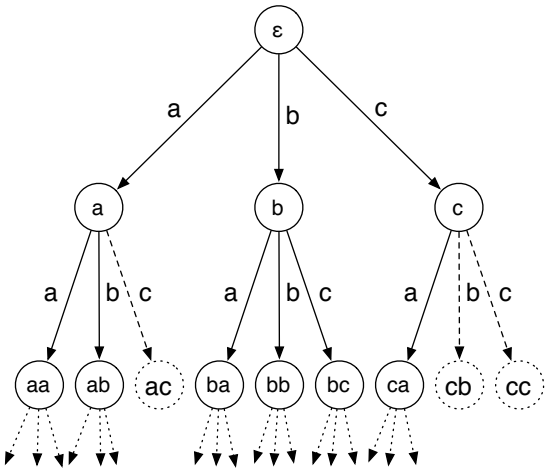

Figure 3.4: An infinite suffix trie.

A big problem with the initial choice of parameters in the Baum-Welch algorithm is that a probability that is zero stays zero after each iteration of the algorithm. This means that a trie consisting only of the root state at the start would stay that way forever. To this end, we use *pseudo-counts* for the transition counts as a form of smoothing. The pseudo-counts for a transition from a state is chosen to be half the probability of that state. The reason for this becomes apparent when we look at the sum of the pseudo-counts at each time step, which is 1. The expected transition counts at each time step also increase by 1 during the update step, which means that we use a simple estimation for the *new* transition count before actually calculating them. Other pseudo-count distributions would also be possible, like a uniform distribution over all transitions.

After the suffix trie has been established this way by learning all sequences, the learning process can be repeated like in the normal Baum-Welch algorithm, that is updating all parameters at the end of each iteration instead of after each sequence, and without pseudo-counts. The first phase of learning is called the "online phase", while this phase is called "batch phase".

3.5 Optimizations for long sequences

Restricting the number of states in the model. To keep the space requirement for the model low, it seems prudent to restrict the number of states, which would otherwise grow as $O(n^2 \cdot m)$. To that end, we define a maximum depth d in the trie, not adding any new states that are deeper than d. This lowers the number of states to $O(d \cdot n \cdot m)$, which is in practice for constant d $o(n \cdot m)$, because of shared prefixes.

Space/time tradeoff for training. For long sequences, the intermediate space needed for training, which grows as $O(d \cdot n)$, can still be prohibitively large. Therefore, we can apply a tradeoff, reducing the space needed in exchange for an increase in run time.

We first formulate the learn step for a single sequence as a recursive function that calculates the β vector from the corresponding α vector, updating the transition counts along the way. Pseudo code for this function is shown in algorithm 1. The functions f and g calculate the α and β updates, respectively.

The idea of the tradeoff is not to store the state distributions for all time steps, instead recalculating some of them as needed. As can be seen in algorithm 2, we calculate the α values for the first half of the training sequence without storing them and then recursively call the learn method again on both halves. Because we only need $\lfloor \log_2 n \rfloor$ calls, the space requirement is reduced from $O(d \cdot n)$ to $O(d \cdot \log n)$, while increasing the time needed from $O(d \cdot n)$ to $O(d \cdot n \cdot \log n)$. This approach is similar to a k-level checkpointing algorithm [5, 16, 19] with $k = \lfloor \log_2 n \rfloor$.

To make better use of available main memory, for small n we can fall back to the linear space (and time) learning algorithm. In the pseudo code, this behavior is governed by the parameter *linearThreshold*.

Algorithm 1 LinearLearn

Parameters: α_t, t, β_n, n
Returns: β_t
  if $t < n$ **then**
    $\alpha_{t+1} \leftarrow f(\alpha_t)$
    $\beta_{t+1} \leftarrow \text{LinearLearn}(\alpha_{t+1}, t+1, \beta_n, n)$
    $\text{Update}(\alpha_t, \beta_{t+1})$
    $\beta_t \leftarrow g(\beta_{t+1})$
    return β_t
  else
    return β_n
  end if

Algorithm 2 DivideAndConquerLearn

Parameters: α_t, t, β_n, n
Returns: β_t
  if $t < n - \mathit{linearThreshold}$ **then**
    $m \leftarrow \lfloor \frac{t+n}{2} \rfloor$
    $\alpha \leftarrow \alpha_t$
    for $i = t+1$ **to** m **do**
      $\alpha \leftarrow f(\alpha)$
    end for
    $\alpha_m \leftarrow \alpha$
    $\beta_m \leftarrow \text{DivideAndConquerLearn}(\alpha_m, m, \beta_n, n)$
    $\beta_t \leftarrow \text{DivideAndConquerLearn}(\alpha_t, t, \beta_m, m)$
    return β_t
  else
    return $\text{LinearLearn}(\alpha_t, t, \beta_n, n)$
  end if

4 Gene searching with suffix trie HMMs

4.1 Base model

To compare the target sequence to other sequences, we first need a model for them, a so-called *null* or *base model*. The simplest base model assigns the same probability to each sequence of the same length. That means that for sequence of length n over an alphabet Σ, the base model is

$$P_{B_\Sigma}(s) = |\Sigma|^{-n} . \tag{4.1}$$

Given a model M and a base model B, we calculate a *log-odds score* S for a sequence s as follows:

$$S = \log \frac{P_M(s)}{P_B(s)} \tag{4.2}$$

Using a log-odds score instead has the advantage that it doesn't suffer from floating point underflow as using the raw probability value would be prone to. Furthermore, it allows us to view the total score as a sum of individual scores, which can be used for searching the whole genome, as will be seen in section 4.3.

The base of the logarithm can be chosen as 2 to get a score in bits, or e for easier calculation of derived values (see sections 4.2 and 4.3.1). In any case, bit scores and natural scores only differ by a constant factor, so it is easy to convert one to the other.

4.2 Coding sequence ranking

For finding the target gene among the coding sequences defined in a genome file, we first assume two things: that the target gene is exactly one of the coding sequences, and that all coding sequences are stochastically independent.

These assumptions are certainly not always the case. For example, in some genome files, the annotations overlap, with one annotation describing an "alternative" subsequence coding the target gene, thus violating both assumptions. Nevertheless, they are necessary for inferring a probability distribution over the coding sequences. How to filter out genomes not containing the target gene at all is discussed in section 4.3.1.

Given a list $\mathbf{g} = (g_1, g_2, \ldots, g_n)$ of coding sequences, the probability that sequence i is the target gene X can be calculated using Bayes' rule:

$$P_{X|G}(i|\mathbf{g}) = \frac{P_{X,G}(i, \mathbf{g})}{P_G(\mathbf{g})} = \frac{P_{G|X}(\mathbf{g}|i) \cdot P_X(i)}{P_G(\mathbf{g})} \tag{4.3}$$

The "evidence probability" $P_G(\mathbf{g}) = \sum_{j=1}^{n} P_{G,X}(\mathbf{g}, j) = \sum_{j=1}^{n} P_{G|X}(\mathbf{g}|j) \cdot P_X(j)$ is independent of i and can thus be subsumed into a normalizing factor α. In practice, one can simply work with the unnormalized values up until the final step and then scale them down so that their sum equals 1.

Furthermore, if we assume that all sequences have the same *a priori* probability $P_X(i)$, the term is also a constant and can be canceled out:

$$P_{X|G}(i|\mathbf{g}) = P_{G|X}(\mathbf{g}|i) \cdot \alpha \tag{4.4}$$

If sequence i is the target gene, it is generated by the model M, while the other sequences are generated by the base model B. Assuming that all sequences are pairwise independent, we get:

$$P_{G|X}(\mathbf{g}|i) = P_M(g_i) \cdot \prod_{j \neq i} P_B(g_j) \cdot \alpha = \frac{P_M(g_i)}{P_B(g_i)} \cdot \prod_{j=1}^{n} P_B(g_j) \cdot \alpha \tag{4.5}$$

The product of the base probabilities $\prod_{j=1}^{n} P_B(g_j)$ is a constant and can also be moved into α. The other term, $\frac{P_M(g_i)}{P_B(g_i)}$, equals the likelihood odds, which we can calculate from the score S_i, resulting in:

$$P_{G|X}(\mathbf{g}|i) = e^{S_i} \cdot \alpha \tag{4.6}$$

Writing out the normalizing factor α, we find that

$$P_{G|X}(\mathbf{g}|i) = \frac{e^{S_i}}{\sum_{j=1}^{n} e^{S_j}}, \tag{4.7}$$

which corresponds to applying the *softmax* function, which is commonly used as an activation function for neural networks, to the sequence scores.

4.3 Whole genome searches

Sometimes the genome file does not contain any annotations besides the genome sequence. In these cases, we would still like to look for the target gene in the genome sequence, assuming that it is surrounded by random sequences.

While assigning a score to a sequence has some similarities to global sequence alignment, in a way aligning a sequence to a model, searching for a subsequence is similar to local alignment.

The probability for a sequence s factors into a product of a conditional probability for each character, depending on the previous ones:

$$P(s) = P(s_1) \cdot P(s_2|s_1) \cdot P(s_3|s_2, s_1) \cdot \ldots \cdot P(s_n|s_{n-1}, \ldots, s_1) \tag{4.8}$$

Likewise, the total score S for a sequence is a sum of individual scores for each character. Therefore, looking for a subsequence of high probability in the genome sequence

amounts to looking for a subsequence of high total score in the sequence of character scores. This is a problem known as the "maximum contiguous subsequence sum" problem, and it can be solved in time linear in the sequence length [3], using an algorithm that can interestingly be seen as a variant of the Smith-Waterman algorithm [15] on a one-dimensional array.

In addition, we can use another linear time algorithm [14] to find not only the highest scoring subsequence, but all subsequences that cannot be lengthed to yield a higher score.

As in the previous section, we can calculate a probability distribution over all subsequences found in this manner. Again, we assume an uniform prior over all found subsequences. While this would not deserve mention by itself, it is important to bring up that we assume a prior probability of zero for all other subsequences. This is justified to some extent because it seems sensible to assume that the target gene is a locally maximal subsequence in terms of score. In addition, calculating scores for all $\frac{n\cdot(n+1)}{2}$ subsequences of a sequence of length n would seem infeasible.

4.3.1 Search hit significance

In long random sequences, high scoring subsequences are expected to occur simply by chance. Therefore, it seems desirable to asses the significance of a search hit in order to make sure that it was actually generated by the target sequence. We do this using *Karlin-Altschul statistics* [1], which are also applied within the BLAST local alignment tool.

The highest scoring match S in a random sequence follows a *Gumbel distribution*, which has the following cumulative distribution function [4, 11]:

$$\mathrm{P}[S \leq x] = \exp\left[-e^{-\lambda(x-\mu)}\right] \tag{4.9}$$

One could expect the chances of a high score in a random sequence to increase with the sequence length. Indeed, as illustrated in figure 4.1, the expected highest score grows with the logarithm of the sequence length [7]:

$$\mathrm{P}[S - \frac{\log n}{\lambda} > x] = 1 - \exp\left[-k \cdot e^{-\lambda x}\right] \tag{4.10}$$

Model calibration. Using these two equations, we can calibrate a given model to give an assessment of a search hit's significance.

We create m random sequences with length n_0 and search for the highest-scoring subsequence in each of them. The resulting scores are saved to x_i, $1 \leq i \leq m$.

Next, we fit the parameters μ and λ of the Gumbel distribution to the obtained scores [11]. The scale parameter λ can be fitted using the empirical variance s^2 of the scores:

$$\hat{\lambda} = \frac{\pi}{\sqrt{6s^2}} \tag{4.11}$$

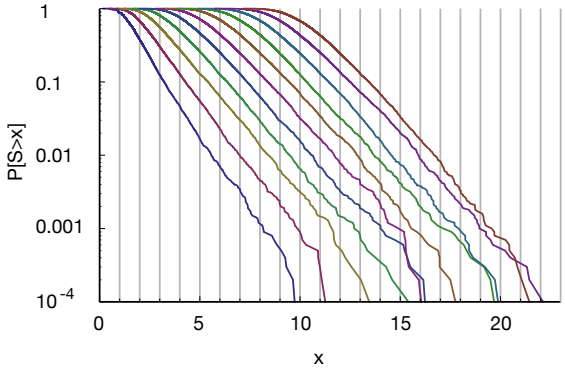

Figure 4.1: Distribution of highest scoring subsequences for 10,000 random sequences of lengths starting at 25 and doubling (analoguous to figure 4 in [4]). According to Karlin-Altschul statistics, in the log-plot the curves are expected to shift right by the same distance with each doubling in sequence length.

A more accurate approach would calculate a maximum-likelihood estimate for λ [11], but empirical evaluation showed that estimating λ from the variance gives sufficient accuracy. Besides, according to [4], for log-odds scores using a natural logarithm, $\lambda = 1$. Experiments gave values of λ that were pretty close to 1, therefore in the built prototype an option is given to either set λ to 1 or fit it to the random distribution using the above-mentioned formula.

The location parameter μ can be calculated from $\hat{\lambda}$ using the following formula:

$$\hat{\mu} = -\frac{1}{\hat{\lambda}} \log \left[\frac{1}{m} \sum_{i=1}^{m} e^{-\hat{\lambda} x_i} \right] \tag{4.12}$$

Combining equations 4.9 and 4.10, we can calculate k from $\hat{\lambda}$ and $\hat{\mu}$:

$$k = \frac{e^{\hat{\lambda}\hat{\mu}}}{n_0} \tag{4.13}$$

Calculating significance measures. Given a search hit score S, we can calculate a *normalized score* S' [8] using the following equation:

$$S' = S\lambda - \log(k \cdot n) \tag{4.14}$$

The normalized score for a random sequence follows a standard Gumbel distribution with $\mu = 0$ and $\lambda = 1$. Now we can calculate the E- and P-values known from BLAST [2]:

The P-value, which is the probability that the highest score in a random sequence is at least x, can be read directly off the cumulative distribution function:

$$P = \mathrm{P}[S' > x] = 1 - \exp(-e^{-x}) \tag{4.15}$$

The E-value is the expected number of hits with score at least x in random sequence. If we assume that the hits in a sequence of sufficient length are indenpendent, the number H of hits with minimum score x follows a Poisson distribution with mean E:

$$P_H(h) = \frac{E^h e^{-E}}{h!} \tag{4.16}$$

For $h = 0$, $\mathrm{P}[S' \leq x] = P_H(0) = e^{-E}$, so

$$E = e^{-S'}. \tag{4.17}$$

If we look at the E-value in terms of the unnormalized score S, we find that $E = n \cdot k \cdot e^{-\lambda S}$, which makes sense intuitively: E is linear in the sequence length and exponential in the score value. The value of λ here serves a scaling factor for the score.

5 Evaluation

5.1 Data sets used

For the following evaluations, two data sets were used: One consisting of several genes from human papillomaviruses (HPV) and one consisting of heat shock protein (HSP70) genes from several organisms.

Human papillomavirus genes. From a gene database of human papillomaviruses, several E1, E2 and L1 genes were extracted. The number of extracted genes can be seen in table 5.1. The reason for the different numbers is that not all genomes contained all of the genes. The gene sequences were randomly split into training and test sets, using aroung 10% of the data set for testing. For the E2 gene, this resulted in a training set consisting of 165 sequences and a test set consisting of 21 sequences.

In order to make better use of the available data, which was of mixed quality, for evaluation of the gene search a cross-validation approach was used. Hereby the data set for each gene was split randomly into 10 groups, with each of these groups constituting a test set for a model trained from the other groups. Because the objective in gene search is to find the target gene in a database containing other genes also, for each test set the corresponding genome files with annotations from the EMBL database were used.

Gene	# of sequences
E1	188
E2	188
L1	198[1]

Table 5.1: Number of HPV gene sequences extracted for evaluation.

Heat shock protein genes. The second data set consisted of 850 genes coding the heat shock protein HSP70, from several organisms. Because of the large size of the full EMBL genome files for these organism, the cross-validation approach was not used here. Instead, from the data set a random sample of 84 gene sequences was extracted for testing, again using the rest of the gene sequences for training and the full EMBL files corresponding to these 84 sequences, together 566 MB in size, for evaluation.

[1] Containing 6 duplicates of other genes, corresponding to 192 genomes.

5.2 Model convergence

The Baum-Welch algorithm seeks to modify the HMM parameters to maximize the likelihood of the observed sequences and guarantees to increase it in every iteration, although it may converge to only a local maximum. Therefore, the first test of the constructed model is to see whether the parameters actually converge, and whether it is able to predict sequences with high probability. We measure this with the *perplexity* in bits, which is the negative base 2 logarithm of the probability assigned to a sequence. The average perplexity is the value of the perplexity divided by the sequence length.

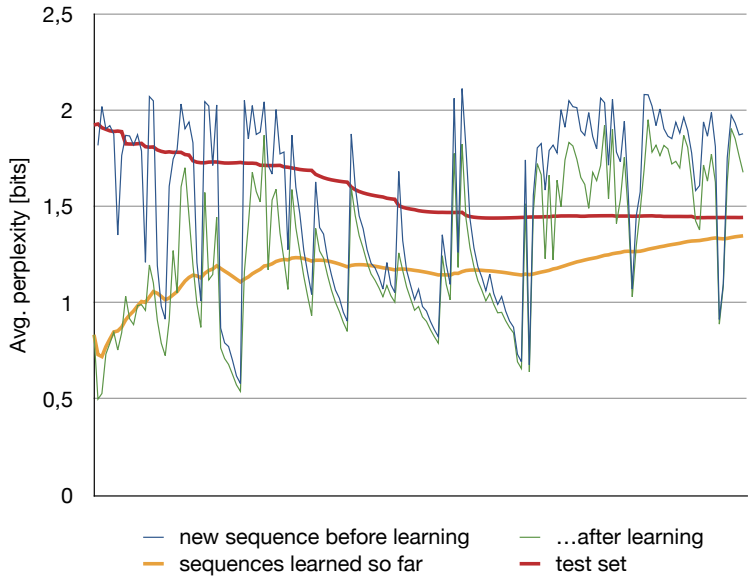

Figure 5.1: Average perplexity of HPV E2 genes during the online learning phase.

As can be seen in figure 5.1, the model already converges during online learning, with the perplexity of the test set at the end not much higher than the perplexity of the training set. Interestingly, the sharp decreases in *test* perplexity are mostly accompanied by spikes in the perplexity of the sequence about to be learned, probably because these sequences are less similar to the ones already learned, and so learning them increases the complexity of the model.

The perplexity can be lowered even further by iterating the learning process in the batch phase. As is apparent from figure 5.2, around 10 iterations are sufficient to get close to the minimum perplexity.

5.3 Effect of maximum depth

Figure 5.3 illustrates the effect of varying the maximum trie depth on the space needed for the model as well as its performance. As could be expected, the number of states

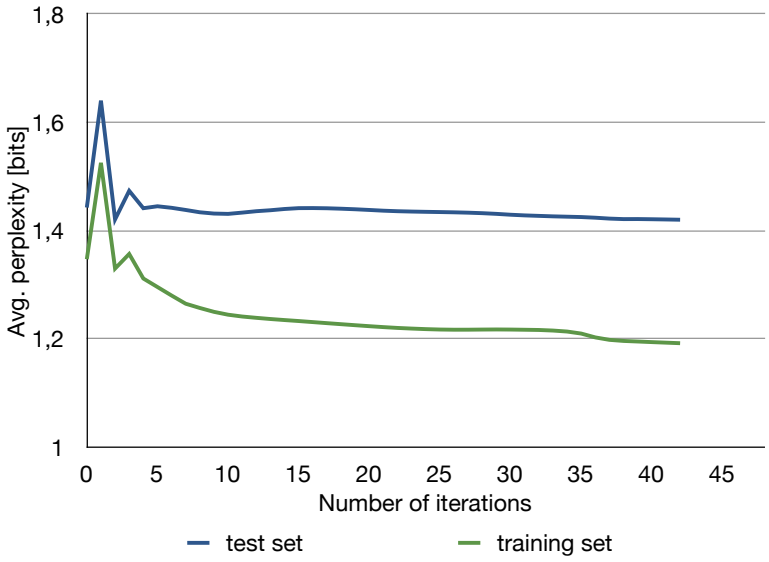

Figure 5.2: Average perplexity of HPV E2 genes during the batch learning phase.

first grows exponentially with the trie depth, than starts to trail off, as not all trie nodes up to the maximum depth are actually created. With the increase in model complexity of course comes better performance, although for trie depths greater than 7 the average perplexities of the training set and the test set start to diverge, hinting to *overfitting* taking place. Nevertheless, even for bigger trie depths, the test perplexity still decreases.

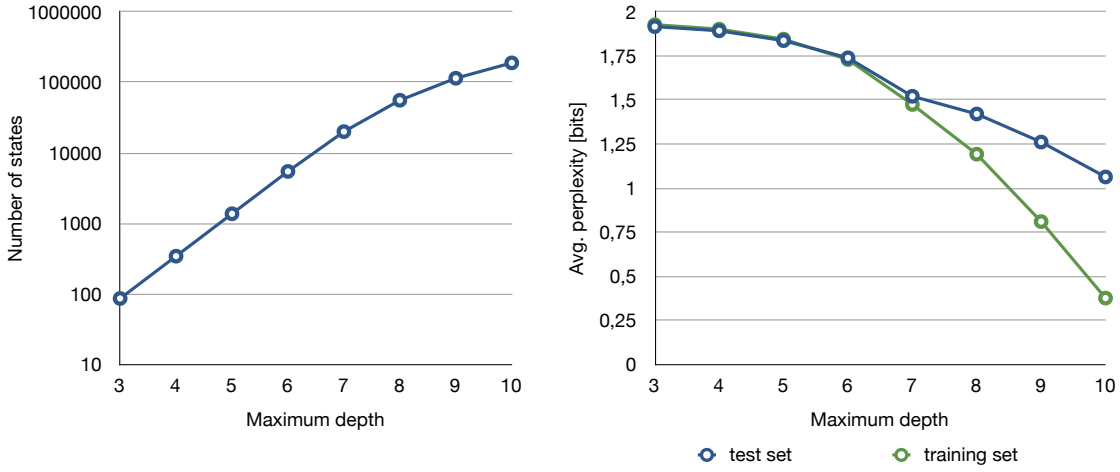

Figure 5.3: Number of states and minimum average perplexity for different maximum trie depths.

5.4 Gene searching

The "real-world" test of the model and the accompanying search method is searching for genes in a genome and gene database. Using the data sets described in section 5.1, the four genes were searched among the coding sequences described by annotations as well as in the whole genome sequences. Table 5.2 summarizes these results. Because the sequence model is position-independent, it cannot capture well the start and end positions of the target gene in the genome sequence, therefore the search hits in the genome sequence are not completely accurate. Therefore, in the table the root mean square error (RMSE) for the start base in the genome is given. Because the prototype does not supported genome search on the complement DNA strand yet and several of the HSP70 genes were lying on it, no value for the RMSE is given for HSP70.

It should be noted that the results could be improved by using a cutoff based on the significance measures described in section 4.3.1, discarding search hits with lower significance and decreasing the chance of false positives.

Gene	Rate	RMSE
HPV E1	98,40%	179
HPV E2	86,17%	553,65
HPV L1	92,93%	159,138
HSP70	73,75%	n/a

Table 5.2: Results of gene searching among the coding sequences and in the genome sequence.

6 Conclusion

6.1 Further work

While the prototype implemented for this work reads EMBL and FASTA files, it could be integrated directly into a sequence database software, like ARB [12]. This would allow for better ease of use in searching for genes, by directly marking the found search hits.

From an algorithmic standpoint, the quality of search results could be improved by employing a more sophisticated base model, for example one that takes the natural abundance of nucleic acids into account.

Last, searching in a whole genome sequence suffers from accuracy problems as seen in section 5.4. To alleviate this problem, a stochastic gene finder model [10] could be used in combination with the suffix trie HMM to increase the accuracy of genome sequence searches.

6.2 Summary

In this work, a way was presented to search in sequence databases using a stochastic model constructed from already known sequences. This allows to search for example in genome files with or without annotations describing coding sequences, ranking the search results as well as allowing an assessment of the accuracy of the search. The stochastic nature of the model here provides a sound foundation for these assessments, while the inference algorithms used scale well with bigger sequence databases and can easily be parallelized.

The model can be used as a tool aiding scientists in looking for homologuous sequences in large databases, complementing simple description searches.

Bibliography

[1] The statistics of sequence similarity scores [online]. Available from: `http://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html`.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, Oct 1990.

[3] J. L. Bentley. *Programming Pearls*. Addison-Wesley, Inc., 1st edition, 1986.

[4] S. R. Eddy. A probabilistic model of local sequence alignment that simplifies statistical significance estimation. *PLoS Computational Biology*, 4(5):e1000069, 05 2008.

[5] J. A. Grice, R. Hughey, and D. Speck. Reduced space sequence alignment. *Computer applications in the biosciences*, 13(1):45–53, Feb 1997.

[6] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Jan 1997.

[7] S. Karlin and S. F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences of the United States of America*, 87(6):2264–8, Mar 1990.

[8] S. Karlin and S. F. Altschul. Applications and statistics for multiple high-scoring segments in molecular sequences. *Proceedings of the National Academy of Sciences of the United States of America*, 90(12):5873–5877, Jun 1993.

[9] A. Krogh. An introduction to hidden Markov models for biological sequences. In S. L. Salzberg, D. B. Searls, and S. Kasif, editors, *Computational Methods in Molecular Biology*, pages 45–63. Elsevier, 1998.

[10] A. Krogh, I. Mian, and D. Haussler. A hidden Markov model that finds genes in E. coli DNA. *Nucleic Acids Research*, 22(22):4768, 1994.

[11] J. Lawless. *Statistical models and methods for lifetime data*. Wiley New York, 1982.

[12] W. Ludwig, O. Strunk, R. Westram, L. Richter, H. Meier, et al. ARB: a software environment for sequence data. *Nucleic Acids Research*, 32(4):1363, 2004.

[13] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Readings in speech recognition*, pages 267–296. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[14] W. L. Ruzzo and M. Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 234–241. AAAI Press, 1999.

[15] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–7, Mar 1981.

[16] C. Tarnas and R. Hughey. Reduced space hidden Markov model training. *Bioinformatics*, 14(5):401–406, Jun 1998.

[17] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[18] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of computational biology*, 1(4):337–348, 1994.

[19] R. Wheeler and R. Hughey. Optimizing reduced-space sequence analysis. *Bioinformatics*, 16(12):1082–1090, Dec 2000.