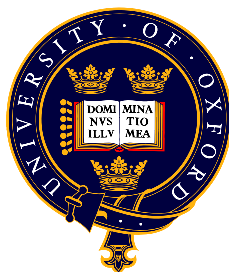
Combining Hidden Markov Models and Stochastic-Context Free Grammars

Joanna Davies
Worcester College



4th Year Dissertation
Honour School of Mathematics and Statistics Part C
University of Oxford

March 2006

Abstract

Currently, gene finding and RNA secondary structure prediction are performed separately. This is only valid if, for any DNA sequence the location of genes and corresponding RNA secondary structure can assumed to be independent. I develop methods that take a hidden Markov model for gene finding and a stochastic context-free grammar for RNA secondary structure prediction and from these, construct a combined model which can make joint predictions. I show that in general, the combined model is a stochastic context-free grammar and compare how an example performs relative to the separate models when data is simulated from the combined model. An investigation of the combined model reveals that apparent dependence can be an artefact of the model rather than dependence present in the data. My results are inconclusive and further work is required to determine whether future implementation of a the joint model is appropriate and feasible on DNA sequence data.

Contents

1 Introduction and Motivation **3**

2 Hidden Markov Models **3**
2.1 Definition and notation . . . 3
2.2 Why are hidden Markov models appropriate for gene finding? . . . 4
2.3 Algorithms associated with Hidden Markov Models . . . 5
2.3.1 Simulation . . . 5
2.3.2 The Viterbi Algorithm . . . 5
2.3.3 The Forward Algorithm and the Backward Algorithm . . . 6
2.4 Parameter estimation for hidden Markov models . . . 6
2.4.1 Maximum Likelihood Estimation . . . 7
2.4.2 Baum-Welch Training . . . 7
2.5 Summary . . . 8

3 Stochastic Context-Free Grammars **9**
3.1 Definitions and notation . . . 9
3.2 Why are stochastic context-free grammars appropriate to model RNA Secondary Structure? . . . 10
3.3 Algorithms associated with Stochastic Context-Free Grammars . . . 10
3.3.1 Simulation . . . 10
3.3.2 The CYK algorithm . . . 11
3.3.3 The Inside Algorithm and the Outside Algorithm . . . 11
3.4 Parameter Estimation for Stochastic Context-Free Grammars . . . 12
3.4.1 Maximum Likelihood Estimation . . . 12
3.4.2 Inside-Outside Training . . . 13
3.4.3 Other training methods . . . 13
3.5 The link between SCFGs and HMMs . . . 13

4 Implementation for the algorithms associated with SCFGs **15**
4.1 The SCFG module . . . 15
4.2 Simulation . . . 15
4.3 The CYK and the Inside algorithm . . . 15
4.4 The Outside Algorithm . . . 16
4.5 Inside-Outside Parameter Training . . . 16

5 Independence **17**

6 Developing a method for combining a general stochastic context-free grammar with a hidden Markov model **21**
6.1 Set-up and notation . . . 21
6.2 Developing a general algorithm for generating the combined grammar . . . 22
6.2.1 Generating the non-terminal alphabet V^* for the combined grammar . . . 22
6.2.2 Generating the production rules P^* for the combined grammar . . . 22
6.3 Assigning parameters to the combined grammar . . . 25
6.4 Combining SCFGs with HMMs with silent states . . . 28

7 Limitations of the Combined Model **31**
7.1 Sensitivity to changes in parameters . . . 31
7.2 Ambiguity . . . 31

8 Results **33**
8.1 Parameter Estimation and Sensitivity . . . 33
8.1.1 Technical Details . . . 35
8.2 Comparing the combined annotation with the HMM and original SCFG annotations . . . 35

9 Conclusion **37**

A The Appendices **39**
A.1 Technical SCFG algorithms 39
A.1.1 Removing epsilon productions from a SCFG which cannot generate the null string 39
A.1.2 Removing replacement productions from a SCFG 39
A.1.3 Collapsing Identical Productions for use of the CYK Algorithm 40
A.2 The SCFG Module 41
A.3 My Extension to the SCFG module 41
A.4 Grammar Files 48
A.4.1 Specification of the HMM as an SCFG 48
A.4.2 Specification of the Simple SCFG 49
A.4.3 Specification of the Random Combined Grammar 50
A.5 Script to test the Inside and the Outside algorithm are consistent 56
A.6 Script to perform parameter training for the combined grammar 57
A.7 Script to calculate sensitivity and specificity from annotations 58

1 Introduction and Motivation

Gene finding is the process of identifying stretches in DNA sequence data which are biologically functional. There are a variety of functional elements including protein coding genes, RNA genes (which produce functional RNA sequences and do not encode proteins) and regulatory regions. The distribution of nucleotides within a coding region is different to that from a non-coding region, and this makes the use of hidden Markov models for gene finding appropriate.

The functional regions of DNA are transcribed, producing RNA sequences via the process of transcription. An RNA sequence is the complement to the DNA sequence from which it is transcribed, but it is single stranded with all thymine nucleotides replaced with uracil nucleotides. It does not bind with another strand (to make it double stranded), instead it folds back on itself such that complementary base pairs and loops are formed. Although the structure is three dimensional it can be reduced to a two dimensional structure according to the base pairs and loops which it forms. This is what is meant by RNA secondary structure. It is considered important because many sequences conserve a secondary structure of base pairings more than they conserve their sequence. Stochastic context-free grammars can model nested, long distance pairwise correlations between nucleotides in a sequence. This makes them appropriate for modelling RNA secondary structure where the complimentary (or even non-complementary) base pairs are correlated with potentially many nucleotides lying between them.

In the majority of previous work in this area, bioinformaticians assume that the location of coding regions within an DNA sequence is independent from the subsequent RNA secondary structure. Intuitively this means that knowledge of one gives no further information about the other. Hence the current frequently used method to annotate a sequence with both its most likely secondary structure and its most likely coding regions is to perform the two annotations separately using a HMM and a SCFG respectively. It is a valid method only if the mechanisms determining secondary structure and the location of coding regions transmit sufficiently strong signals that can be received marginally. There is no biological justification for this assumption and leads naturally to asking several interesting questions:

1. What does it mean for a HMM and a SCFG to be independent and how can this notion be defined?
2. If a sequence is labelled with both coding regions and secondary structure by simply using each algorithm separately is it then possible to establish whether the corresponding HMM and SCFG are independent?
3. How can dependence be established and modelled if appropriate?
4. Is it possible to construct a model that can be used to annotate a sequence with it's predicted coding regions and secondary structure simultaneously?
5. If such a model is possible to construct, how does it perform relative to the separate models?
6. What is the magnitude of the error made if independence is incorrectly assumed?

My project investigates a method of combining a stochastic context-free grammar with a hidden Markov model such that the resulting combined model can impose a joint distribution over possible secondary structure and over possible locations of genes for any given sequence. I hope to achieve this and answer the above questions over the course of the project. The following work is largely theoretical although motivated by the biological problem I have described.

2 Hidden Markov Models

2.1 Definition and notation

A hidden Markov model (HMM) can be fully specified by three components which are explained in the subsequent text:

1. A set of hidden states $\{H\}$
2. A set of symbols emitted from the hidden states $\{T\}$

3. A set of parameters Θ, specifying transition and emission probabilities.

They can be used to model some hidden structure underlying observed sequences of a finite number of symbols (from the set T). The sequence of symbols observed are considered to be emitted from a sequence of underlying unobservable hidden states. The hidden state sequence is called the path and it is described by a simple Markov chain, so the probability of the path being a particular state at one point (in time or position) depends only on the previous state; knowledge of earlier states is redundant.

The starting state can also be given an initial distribution by introducing a silent 'begin' state which does not emit any symbol and only makes a single transition to any one of the other states according to the specified initial distribution. In a similar way the final state of the path can be modelled. The silent 'end' state is introduced such that all other hidden states can make a transition to the end state and the probability of such a transition can be set to zero if necessary. This also imposes a distribution on the length of the sequence generated. The begin and end states are said to be silent states of the HMM because they do not emit any observable symbols. It is possible to build complex models consisting of a mixture of silent and non-silent states and in many cases inserting appropriate silent states can reduce the number a parameters required by the model significantly.

Denote the state path by π, so that π_i denotes the ith state of the path. The state path is described by a simple Markov chain with transition parameters q_{ij} (as shown in figure 2), where q_{ij} denotes the probability of making a transition from state i to state j. These parameters can be conveniently stored in a square transition matrix (denoted Q) with dimension equal to the cardinality of the set H.

The symbols are decoupled from the states, but the distribution of the symbol emitted is state dependent. Another set of parameters are used to specify these distributions. They are referred to as emission probabilities and are denoted $e_k(b)$ for $k \in H$ and $b \in T$ where $e_k(b)$ is the probability that symbol b is emitted when the Markov chain or equivalently the hidden path is in state k.

2.2 Why are hidden Markov models appropriate for gene finding?

Hidden Markov models are frequently used to locate coding regions within DNA sequences. They are appropriate because only a sequence of nucleotides can be observed, the 'coding state' of each nucleotide is unknown (i.e is it part of a coding region or not?). The distribution of nucleotides within coding regions differ to that of non-coding regions, this information can be incorporated into the model via the allocation of the state dependent emission probabilities ($\mathrm{e}_{coding}, \mathrm{e}_{non-coding}$). The initial distribution and the distribution of the length of coding and non-coding regions can be imposed via the allocation of transition probabilities (Q) between the hidden states. For example, a very simple model can specified by

1. H={non-coding (0), coding (1), $begin$, end}, The $begin$ and end states are silent states.
2. $T=\{a, c, g, t\}$
3. $\Theta=\{\theta_0, \theta_1, Q\}$

More complex models can be constructed using a mixture of non-silent and silent states. The construction of such models is well researched and they vary according to the types of DNA sequences being modelled. My project is theoretical, so I consider simple models and simulate data accordingly.

My ultimate project aim is to combine a HMM and a SCFG, so it is important to understand how all the associated algorithms work. I proceed to introduce the algorithms associated with HMMs by considering a simple HMM denoted M_1 which is specified in figure 1 and interpreted in figure 2.

Figure 1 Specification of the two state two symbol HMM; a simplified gene finding HMM where state zero corresponds to the non-coding state and state 1 corresponds to the coding state.

1. H={0, 1, $begin$, end}
2. T={a,b}
3. $\Theta=\{\theta_0, \theta_1, Q\}$

2.3 Algorithms associated with Hidden Markov Models

2.3.1 Simulation

It is necessary to be able to simulate sequences generated by a fully parameterised hidden Markov model when the models are purely theoretical and too simplistic to model DNA sequences. This is the case for the models I consider, therefore the data I use will take the form of sequences simulated from the appropriate HMM.

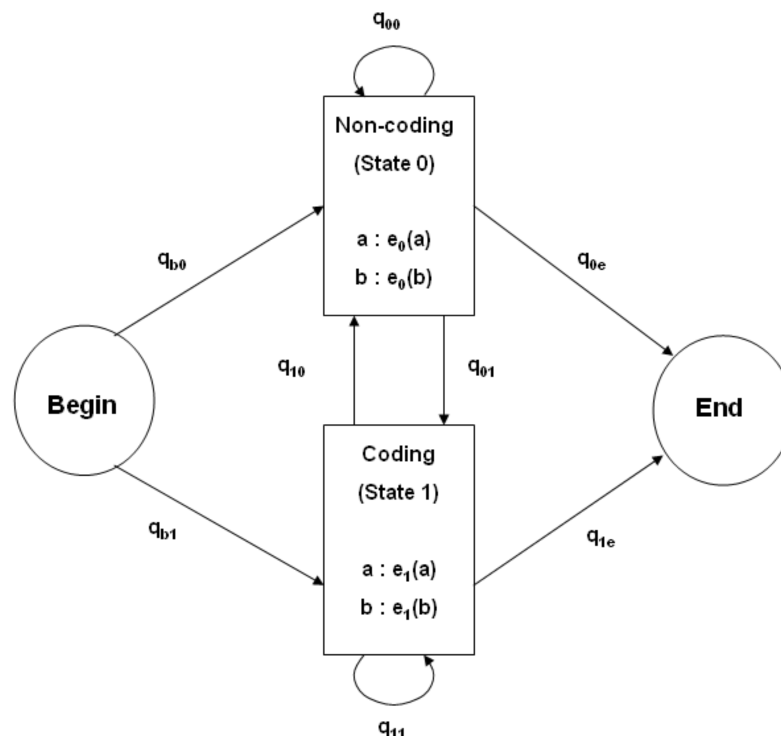
2.3.2 The Viterbi Algorithm

The Viterbi algorithm is used to predict the underlying hidden state path which emitted a sequence of observed symbols. It predicts the unknown path by finding the most probable path given the emission and transition probabilities for the HMM. The algorithm finds the most probable path (out of all possible paths) recursively. It is a dynamic programming algorithm with four stages; initialisation, recursion, termination and traceback.

It works by recursively calculating the quantity $v_k(i)$, the probability of the most probable path ending in state k with symbol x_i for each state k, running through every position i in the sequence. The recursion is obtained simply by conditioning on the state emitting the previous symbol. The most probable path up to the ith position given the chain is in state k at the ith position must in our simple case either be the most probable path up to $(i-1)$th position given the $(i-1)$th symbol is in state 1 multiplied by the probability that a transition is made to state k, or the most probable path up to the $(i-1)$th position given the $(i-1)$th symbol is in state 0 multiplied by the probability that a transition is made from state zero to state k. So $v_k(i)$ is the maximum of these two quantities multiplied by the probability of emitting the ith symbol from state k. For details of the algorithm see [1] p. 56 and for my implementation of the algorithm see the Appendix A.3. If silent states other than the begin and end state are introduced it requires modification but for the HMMs I consider the algorithm as presented in [1] suffices.

This alone does not find the most likely path, it finds the probability of the most likely path through the hidden states. To find the path itself, a traceback can be done and this is easy to do by keeping track of the arguments

Figure 2 A diagram to show the structure of the simple hidden Markov model M_1 described in the text labelled with its transition and emission parameters.



(i.e the states) which yield the maximum when the v quantities are calculated and then tracing back those which contribute to the most likely path. For further details of the trace back see [1] p. 56.

2.3.3 The Forward Algorithm and the Backward Algorithm

The probability that an HMM emits a particular sequence of symbols can be calculated by summing the probabilities over all possible paths which can emit the sequence. But as the length of the sequence grows the number of possible paths grows exponentially. Hence enumerating the possible paths is not practical. In some cases the likelihood can be approximated by the probability of the most probable path. This assumes that the only path that makes a significant contribution to the likelihood is the most likely one. It provides a surprisingly good approximation in many cases.

The full probability can be calculated using a similar recursion to the Viterbi algorithm which replaces the maximisation steps with sums. The resulting algorithm is the forward algorithm. It recursively calculates the quantities $f_k(i)$ for $k \in H$ and $i \in \{1, 2, .., r\}$ where r is the length of the sequence and $f_k(i)$ denotes the probability of observing sequence $\mathbf{x}$ up to and including the ith symbol requiring that k is the hidden state emitting the ith symbol. It is another dynamic programming algorithm with initialisation, subsequent recursions and termination. For further details of the recursion see [1] p.58.

In a similar way it is also possible to calculate the likelihood of observing a particular sequence using a recursion starting from the end of the sequence which works its way back along the sequence taking sums. The algorithm is very similar and it recursively calculates the quantities $b_k(i)$ (for $k \in H$ and $i \in \{1, 2, .., r\}$) where $b_k(i)$ denotes the probability of observing the sequence $\mathbf{x}$ from the ith position onwards (excluding the ith position itself) with the requirement that the underlying state emitting the ith symbol is k. For further details of the recursion see [1] p.59

Calculation of the forward and backward quantities allow inference to be made about the hidden state of a symbol given the rest of the sequence. The probability that the ith nucleotide in the sequence is emitted from state k with no restrictions on the hidden states of the other nucleotides is $f_k(i) \times b_k(i)$ and since this quantity can be calculated for each nucleotide in the sequence, it can be used as an alternative to the Viterbi algorithm. This method known as posterior decoding is often used when several paths have a similar path to that of the most probable one. Using this method yields the path $\widehat{\pi}$ where $\widehat{\pi}_i = \arg\max_{k \in H}\{P(\pi_i = k|x)\}$. This disadvantage of posterior decoding is that the method predicts each state separately, so the resulting path may not be very likely or even possible.

2.4 Parameter estimation for hidden Markov models

The previous algorithms can be implemented provided that the hidden Markov model modelling the sequences is fully specified, that is the structure and the parameters of the model are given. In the field of bioinformatics model structures are constructed based on prior biological knowledge of the process generating the observed data sequences and it is an area well researched in itself. The model structure is specified by the possible hidden states and possible subsequent emissions from these states. For example, take the simple model M_1 specified in figure 1 for locating coding regions, there are two possible non-silent hidden states 0 and 1 corresponding to coding and non-coding and in each state either an 'a' or a 'b' can be emitted. Although the model structure is frequently assumed to be known and fixed, it is rare that transition and emission parameters are known and it is usually necessary to estimate them from sequence data which can be either real DNA sequences or as I will use, simulated sequences. These sequences are known as training sequences. If the sequences are simulated then knowledge of the true parameters is also available for comparison and evaluation.

There are two different approaches to parameter estimation, supervised and unsupervised. Supervised methods are appropriate only if a set of training sequences is available for which all the hidden states are known, otherwise, if either some or none of the hidden states are known, unsupervised methods must be implemented. The most widely known supervised learning technique is maximum likelihood estimation, while the Baum-Welch algorithm is typically used in the unsupervised case.

2.4.1 Maximum Likelihood Estimation

Given a sequence of symbols which has already been annotated with its hidden states, it is possible to obtain maximum likelihood estimates for the transition and emission probabilities just by taking the corresponding proportion of times a particular emission or transition occurs. Proof that these estimators yield the maximum likelihood is given in [1] p. 319. Using the following notation, closed form expressions for the maximum likelihood estimators can be written down. Let:

- A_{ij} be the number of transitions observed from state i to state j
- $E_i(k)$ be the number of emissions of symbol k from state i

Then,

$$\widehat{q}_{ij} = \frac{A_{ij}}{\sum_l A_{il}} \qquad \widehat{e}_i(k) = \frac{E_i(k)}{\sum_j E_i(j)}$$

If it is the case that some transitions or emissions do not occur these estimates are not always well defined. Introducing pseudo counts (based on prior beliefs about the model) can avoid this problem although the corresponding estimates are not strictly maximum likelihood estimates.

2.4.2 Baum-Welch Training

The Baum-Welch algorithm is an iterative procedure which can be used to estimate parameters when the hidden paths for training sequences are unknown. It is a version of the EM algorithm and the hidden states of the training sequences can be seen as the missing data. The E-step calculates the expected number of times the valid transitions and emissions (from arbitrary initial values of the parameters) occur in the training sequences. The M-step maximises the expected loglikelihood given the expected counts with respect to the unknown transition and emission parameters. The M-step is simple in this context with the new emission and transition parameter estimates just being the appropriate proportion of expected emissions and transitions respectively. The E-step is the more complicated step in the iteration and uses the forward and backward quantities defined previously. For example under M_1, the probability that the transition from state j to state k ($j, k \in \{0, 1\}$) is made after the ith symbol is emitted is

$$P(\pi_i = j, \pi_{i+1} = k|\mathbf{x}, \theta) = \frac{f_j(i) q_{jk} e_k(x_{i+1}) b_k(i+1)}{P(\mathbf{x}|M_1)}$$

where θ denotes initial or current parameters for the HMM and $\mathbf{x}$ denotes the sequence.

By summing over all positions and all training sequences, the expected number of times that a particular transition or emission occurs can be derived. Continuing the example above, this yields the following expressions for the expected number of transitions from j to k (denoted $\tilde{A}_{jk}$) and for the expected number of emissions of symbol v from state j (denoted $\tilde{E}_j(v)$). The sums taken over r are sums taken over the number of training sequences and sums over i are taken over the number of symbols in sequence $\mathbf{x}^r$.

$$A_{jk} = \sum_r \frac{1}{P(\mathbf{x}^r|M_1)} \sum_i f_j^r(i) q_{jk} e_k(\mathbf{x}_{i+1}^r) b_k^r(i+1) \qquad E_j(v) = \sum_r \frac{1}{P(\mathbf{x}^r|M_1)} \sum_{i:\mathbf{x}_i^r = j} f_j^r(i) b_j^r(i)$$

New parameter estimates are then obtained via the M-step which inserts expected frequencies in place of observed frequencies in the Maximum Likelihood Estimators defined in section 2.4.1. The parameters obtained at each iteration do not decrease the likelihood function hence they will converge to a local maximum. For a proof and details of the Expectation Maximisation algorithm in this context see [1] p.324. A maximum number of iterations may be specified or a threshold on the change in log likelihood at each iteration can be set as a stopping rule for the algorithm.

The major problem with the algorithm is that it converges to a local maximum which is not always the global maximum. Determining whether the probabilities to which the algorithm has converged are the correct ones is difficult. It is common practise to run the training on the same set of sequences from different initial starting values and then use the set of parameters which yield the maximum loglikelihood. Although this does not guarantee that the global maximum will be found it does increase the chances of finding it.

To assess how accurately Baum-Welch training estimates parameters a simple test can be constructed. It is described below. It can also be used as a check that the implementation of the algorithm is working correctly. If a simple HMM is used then the number of local maxima is low and one would expect to be able to obtain fairly accurate estimates of the parameters.

1. Simulate sequences according to a HMM with specified and known parameters.
2. Run the Baum-Welch training algorithm on the sequences starting from several different sets of initial parameters (assigned randomly).
3. Set the estimated parameters to those which maximise the loglikelihoods calculated from each of the starting points.
4. Compare the estimated parameters with those of the HMM from which the training sequences were simulated typically via calculation of the mean squared error or by comparing histograms of the absolute values of the errors.

2.5 Summary

Given training sequences it is possible to estimate parameters of the HMM modelling the sequences, such that the unknown path can then be predicted using the Viterbi algorithm. This is particularly useful in the context of determining where coding and non-coding regions are along a sequence. The coding and non-coding states corresponding to the hidden states and the nucleotides corresponding to the symbols emitted.

My implementations of these algorithms together with those in the following section on stochastic context-free grammars are discussed in section 4. The code is displayed in Appendix A.3.

3 Stochastic Context-Free Grammars

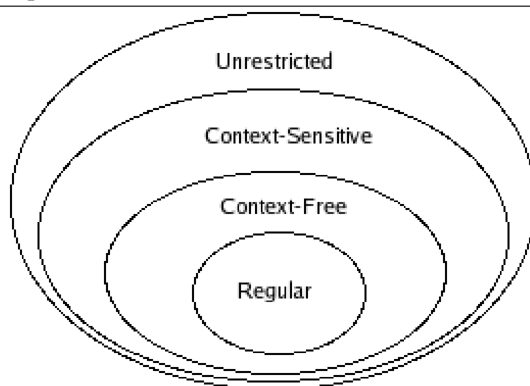
3.1 Definitions and notation

There are four components to any grammar:

1. V a finite set of non-terminal symbols or states, for example {S,W}.
2. T a finite set of terminal symbols, for example in the context of RNA sequences T={a, c, g, u}.
3. P a finite set of production rules.
4. S the initial starting non-terminal, which is always contained in V.

The type of grammar (regular, context-free, context-sensitive or unrestricted) is determined by the nature of the production rules. These grammars form a hierarchy known as Chomsky's hierarchy which can be interpreted easily from figure 3 below.

Figure 3 A diagram to show the Chomsky hierarchy of grammars. As fewer and fewer restrictions are placed on production rules a nested structure is formed. The nestings are proper in the sense that there exist elements in the outer three classes which are not present in the lower classes.



Definitions of the types of grammar given below and Chomsky's hierarchy diagram are taken from [1] p.236. Let:

1. α and γ denote any string of terminals and/or non-terminals $including$ the null string.
2. W denote a generic non-terminal.
3. β denote any string of terminals and/or non-terminals $excluding$ the null string.
4. 'a' denotes a generic terminal.

Then, a grammar can be classified into the appropriate minimal class in the hierarchy by inspection of the production rules according to the following:

- **Regular grammars** allow production rules of the form $W \to aW$ or $W \to a$.
- **Context-free grammars** allow production rules of the form $W \to \beta$, allowing the left hand side of the production rule to consist of a single non-terminal and the right hand side any non-empty string of terminals and non-terminals.
- **Context-sensitive grammars** allow production rules of the form $\gamma_1 W \gamma_2 \to \gamma_1 \beta \gamma_2$.
- **Unrestricted grammars** allow production rules of the form $\alpha_1 W \alpha_2 \to \gamma$

Context-free grammars are particularly useful because they permit rules that allow the grammar to create nested, long-distance pairwise correlations between terminal symbols. Although context-sensitive and unrestricted grammars can also do this, for computational efficiency it is advantageous to use the simplest possible grammar in the hierarchy.

A simple (and soon to be explained relevant) example is that of the palindrome grammar separated by spacers. It is denoted G_1 and defined in figure 4. It describes all palindromes separated by spacers (i.e single symbols

Figure 4 A figure defining the palindrome-spacer grammar over the alphabet $\{a, b\}$

$$\begin{aligned} S &= S \\ V &= \{S\} \\ T &= \{a, b\} \\ P &= \{S \rightarrow aSa|bSb|aS|Sa|bS|Sb|a|b|aa|bb\} \end{aligned}$$

emitted) over the alphabet $\{a, b\}$ excluding the null string. There are technical reasons why it is best, particularly in the context of RNA secondary structure prediction to avoid grammars which are capable of producing the null string. These are discussed in section 4.3.

A grammar is stochastic when probabilities are assigned to the production rules. Stochastic regular grammars are equivalent to Hidden Markov Models in the sense that they can produce/process the same type of languages. This useful result is a direct consequence of the computational result that the class of regular languages is equivalent to the class of languages that can be produced by finite automata (proved in [10]). Stochastic Context-Free Grammars (SCFGs) are more complex than HMMs but have algorithms analogous those described in the previous section.

3.2 Why are stochastic context-free grammars appropriate to model RNA Secondary Structure?

RNA secondary structure can be regarded as a set of nested and potentially long distance pairwise correlations on an RNA sequence. It is like a palindrome language, but the correlated pairs are complementary or non-complementary base pairs instead of identical bases. For a simple model, correlated complementary base pairs fold to form stems and spacers form loops. Context-free grammars cannot model pseudo-knots which are formed by an intertwined dependence structure but they occur rarely and are therefore often neglected. However, it is possible to incorporate the possibility of branching loops and structures which make the grammar more realistic and applicable to real RNA sequences. These grammars are more complex and will not be studied in this project.

Given any sequence of nucleotides and a SCFG (including probabilities assigned to production rules) it is possible to find all possible derivations of the sequence and corresponding probabilities. This process is known as parsing. From any parse, the secondary structure can be derived according to the non-terminals and subsequent productions used, hence a distribution over possible secondary structures can be computed. Each possible structure has a certain probability given by the SCFG. If there is a one-to-one map from parses to secondary structures then finding the most likely secondary structure corresponds to finding the most likely parse. If many parses yield the same secondary structure, finding the most likely secondary structure is more complicated. The probability of a particular secondary structure is found by summing the probabilities of any parse which produces that secondary structure. It is more computationally expensive and it is often assumed that the most likely secondary structure corresponds to that of the most likely parse. I make this assumption and it will be discussed in section 7.2.

3.3 Algorithms associated with Stochastic Context-Free Grammars

For each of the algorithms described in the previous section on HMMs there is an analogous algorithm for SCFGs.

3.3.1 Simulation

The first stage is to simulate strings or sequences of characters generated by a parameterised SCFG. It is necessary and useful for parameter estimation when real sequence data is not available.

3.3.2 The CYK algorithm

The CYK algorithm is analogous to the Viterbi Algorithm. It is used to find the probability of the most likely derivation of a sequence of symbols and the subsequent traceback finds the corresponding path. It is a very similar to the inside algorithm which will be discussed in more detail in the next section. For details of the algorithm and the quantities calculated by the recursion, see [1] p.257

It is possible to use this algorithm to annotate any string from the alphabet $\{a,b\}$ with a secondary structure derived from the most likely parse of the sequence. Both the structure and parse of a sequence can be represented using parentheses as shown by the simple example below.

String: $ababa$
Possible derivation: $S \rightarrow aSa \rightarrow abSba \rightarrow ababa$
Representation of the parse: $S(aS(bS(S(a))b)a)$
Corresponding secondary structure:

$$a - a$$

$$b - b$$

$$a$$

Representation of secondary structure:

$$ababa$$

$$((.))$$

3.3.3 The Inside Algorithm and the Outside Algorithm

Assume that a SCFG is used to model correlations in a sequence, then the likelihood of observing the sequence can be calculated by summing over the probabilities of all possible derivations which can produce that sequence. It is analogous to calculating the likelihood of a sequence generated by a HMM by summing over the probabilities of all possible paths through the sequence. As described in the previous section, this can be done via the forward or the backward algorithm. For SCFGs it can be done using either the inside algorithm or the outside algorithm.

As the sequence length increases, the number of possible derivations increase super exponentially, but it is possible to calculate and sum efficiently over all of them using recursive methods. It relies on the production rules of the grammar being specified in *Chomsky's Normal Form (CNF)*, that is all production rules take one of three possible forms:

1. $W_i \rightarrow W_j W_k$ where W_i, W_j, W_k are non-terminal symbols.
2. $W_i \rightarrow a$ where a is a terminal symbol.
3. $S \rightarrow \varepsilon$ where ε is the empty string.

Rules of type 1, type 2 and type 3 are referred to as non-terminal productions, terminal productions and epsilon productions respectively. The production rules for any SCFG can be reduced to this form. A proof of this fact is demonstrated in [11] by the presentation of an algorithm to convert the rules of any SCFG to an equivalent grammar where the rules are specified in CNF. In particular I apply the algorithm to the simple palindrome grammar G_1 defined in section 3.1. An alternative specification of the production rules in CNF is given in figure 5. The quantities computed by the inside algorithm and other quantities used by the algorithm are introduced using the following notation:

- $t_v(y, z) = P(W_v \rightarrow W_y W_z)$
- $e_v(a) = P(W_v \rightarrow a)$
- $\alpha(i, j, v)$ denotes the probability that the parse subtree for subsequence $x_i, ..., x_j$ is rooted at W_v.

The inside algorithm calculates the probability of observing a sequence by calculating $\alpha(i, j, v)$ for all i, j and v recursively. For details of the dynamic programming algorithm see [1] p.254. The probability of observing an entire sequence of length n, given the parameterised SCFG grammar is simply $\alpha(1, n, 0)$, taking $S = W_0$.

Figure 5 A figure to present the specification of G_1 with production rules in CNF

$$
\begin{aligned}
S &= S \\
V &= \{S, W_1, W_2, W_3, W_4\} \\
T &= \{a, b\} \\
P &= \{S \to W_1W_1|W_1W_2|W_3W_4|W_3W_3|W_1S|SW_1|SW_3|W_3S|a|b, \\
&\quad W_2 \to SW_1,\ W_4 \to SW_3,\ W_1 \to a,\ W_3 \to b\} \qquad (1)
\end{aligned}
$$

The outside algorithm recursively calculates the quantities $\beta(i,j,v)$ where $\beta(i,j,v)$ denotes the probability of a complete parse tree rooted at the start non-terminal for the complete sequence x, excluding parse subtrees rooted at non-terminal W_v for the subsequence $x_i, ..., x_j$. The calculation of this quantity requires the $\alpha(i,j,v)$ quantities computed via the inside algorithm. It starts from the largest excluded sequence and works inwards (in contrast to the inside which starts with the smallest and works outwards).

Implementations of the inside and outside algorithms can be tested by running both algorithms on the same sequence with the same probabilities assigned to production rules. If the code is working correctly they should both output the same probability of observing the sequence.

These algorithms can take any sequence of a's and b's (in the case of G_1) and calculate the probability with which the fully specified SCFG can produce it, or more generally the α and β matrices can be computed. These algorithms can only be performed if the production rules and parameters are known. In the context of RNA secondary structure prediction this is rarely the case and it is necessary to estimate parameters using another algorithm.

3.4 Parameter Estimation for Stochastic Context-Free Grammars

If a set of training strings are available with corresponding derivations then supervised learning techniques can be used. Again maximum likelihood estimation is widely used in this setting but if annotated strings are not available unsupervised learning techniques must be implemented. The EM algorithm is commonly used and in the context of SCFG parameter estimation it is known as inside-outside training. The non-terminals and the production rules generating the sequence can be interpreted as the missing data.

3.4.1 Maximum Likelihood Estimation

If there are training sequences available for which the grammar derivations are known, then the maximum likelihood estimators for $t_v(y,z)$ and $e_v(a)$ have neat closed form formulae which can be expressed using the following notation. Let,

- C_v denote the number of times the non-terminal symbol W_v is used in the derivations of the training sequences.
- $T(v,y,z)$ denote the number of times the rule $W_v \to W_yW_z$ is used in the derivations of the training sequences.
- $E_v(a)$ be the number of times the rule $W_v \to a$ is used in the derivations of the training sequences.

Then the maximum likelihood estimators for $t_v(y,z)$ and $e_v(a)$ are

$$\widehat{t}_v(y,z) = \frac{T(v,y,z)}{C_v} \qquad \widehat{e}_v(a) = \frac{E_v(a)}{C_v}$$

If the grammar is capable of producing the null string, the maximum likelihood estimate of the probability assigned to the rule $S \to \varepsilon$ is given by the number of null sequences observed divided by the total number of training sequences.

They are, as expected the appropriate proportions of the number of uses of a rule relative to all other possible rules which could be used, and the proof is analogous to the derivation of the maximum likelihood estimators for the transition and emission probabilities assigned to a HMM, see [1] p.311.

3.4.2 Inside-Outside Training

Inside-Outside training is the EM algorithm in the context of SCFG parameter estimation. The E-step calculates the expected number of uses of each non-terminal and each production rule. The M-step maximises the expected loglikelihood with respect to the unknown parameters given the expected counts derived from an initial or current set of parameters in the E-step. The M-step reduces to using the maximum likelihood formulae defined in the previous section to obtain new parameter estimates by replacing observed counts with expected counts. The disadvantage of this method is that like the Baum-Welch algorithm, the local maximum to which the loglikelihood converges is dependent on the initial values and the parameters corresponding to the global maximum are not guaranteed to be found even when the algorithm is run several times from different starting points.

3.4.3 Other training methods

Although inside-outside training is widely used for parameter estimation, it is not always accurate, particularly if the parameters to which it converges yield a local maximum rather than a global one. There are already Bayesian alternatives to the Baum-Welch algorithm for hidden Markov model parameter estimation and it is possible to adapt these methods to develop a Bayesian approach to SCFG parameter estimation. These may provide a good alternative although the problem of specifying initial parameters will be replaced by the problem of specifying suitable priors.

3.5 The link between SCFGs and HMMs

The algorithms associated with SCFGs and HMMs are noticeably similar. Table 1 shows the analogous algorithms and their applications.

OUTPUT	HMM ALGORITHM	SCFG ALGORITHM
Most likely Parse	Viterbi	CYK
Likelihood Calculation	Forward	Inside
Likelihood Calculation	Backward	Outside
Parameter Training	Baum-Welch	Inside-Outside

Table 1: A table to compare the analogous algorithms for SCFGs and HMMs and their applications.

Regular grammars are contained within the class of context-free grammars as illustrated by figure 3 in section 3. Consequently it is possible to express a HMM as a SCFG. In particular, consider M_1, the HMM defined in section 2.2. It can be alternatively expressed as the context-free grammar G_2 defined in figure 6.

Figure 6 Specification of M_1 as a SCFG

$$\begin{aligned} V &= \{S, C, C', N, N', E\} \\ T &= \{a, b, \} \\ P &= \{S \to CE|NE,\ C \to C'C|C'N|C'E,\ N \to N'N|N'C|N'E,\ C' \to a|b,\ N' \to a|b,\ E \to \varepsilon\} \\ S &= S \end{aligned}$$

It is possible to assign probabilities to the production rules P in such a way that the resulting SCFG is recognisable as the HMM with two non-silent hidden states corresponding to the 'coding' and 'non-coding' states each emitting symbols from the alphabet $\{a,b\}$. The probabilities assigned are displayed in table 2 in terms of the transition and emission parameters of the HMM (q_{ij} for $i, j \in H$ and $e_k(\alpha)$ for $k \in H$ and $\alpha \in T$ respectively).

RULE	PROBABILITY	NOTES
$S \to CE$	q_{b1}	Probability of making a transition from the silent begin state to the coding state.
$S \to NE$	q_{b0}	Probability of making a transition from the silent begin state to the non-coding state
$C \to C'C$	q_{11}	Probability of remaining in the coding state.
$C \to C'N$	q_{10}	Probability of making a transition from the coding state to the non-coding state.
$C \to C'E$	q_{1e}	Probability of making a transition from the coding state to the silent end state.
$N \to N'C$	q_{01}	Probability of making a transition from the non-coding state to the coding state.
$N \to N'N$	q_{00}	Probability of remaining in the non-coding state.
$N \to N'E$	q_{0e}	Probability of making a transition from the non-coding state to the silent end state.
$C' \to a$	$e_1(a)$	Probability of emitting 'a' from the coding state.
$C' \to b$	$e_1(b)$	Probability of emitting 'b' from the coding state.
$N' \to a$	$e_0(a)$	Probability of emitting 'a' from the non-coding state.
$N' \to b$	$e_0(b)$	Probability of emitting 'b' from the non-coding state.
$E \to \varepsilon$	1	The end non-terminal E emits the null string with probability 1.

Table 2: A table to show how probabilities should be assigned to the production rules of G_2 such that it is recognisable as M_1.

This formulation of a hidden Markov model as a SCFG is particularly useful for implementing the algorithms since it is only necessary to implement the algorithms corresponding to a SCFG. Then implementation of the analogous algorithm for a HMM can be achieved simply by specifying the HMM as a SCFG and running the required algorithm. It is less efficient to run HMM algorithms in this way, but since my work is theoretical and will use relatively small simulated sequences this is not a major issue. I have implemented all of the algorithms described in this section in Python. Details of the implementation are given in the next section together with some checks to establish that they are working correctly. The code is displayed in the Appendices.

4 Implementation for the algorithms associated with SCFGs

4.1 The SCFG module

The module is an Object Oriented program with the main class creating objects of type SCFG. Any SCFG can be read in to the module from a file which contains its terminal alphabet, production rules and corresponding probabilities (although it is not necessary to specify these accurately). A grammar is converted to Chomsky's normal form as it is read in and the process by which this is done is according to the algorithm in [11]. This is particularly useful since the algorithms described in the previous section are implemented when the production rules of the SCFG are specified in this form. They are possible to implement without the production rules in CNF but the way they are presented in [1] and in my implementation they are required to be in CNF.

The module makes the distinction between non-terminal, terminal and epsilon productions which are the three different types of rule permitted when the grammar is specified in CNF (for formal definitions refer back to section 3.3.3). In particular iterators are defined such that one can iterate through all the various types of productions. This is very useful and provides an efficient tool when it comes to implementation of the various algorithms. Other useful methods included in this module allow the probabilities of specific productions to be accessed and reset (which is particularly useful in the context of parameter estimation). The final method to be noted from this module is that of the normalisation method. It means that weights can be assigned to the production rules when it is initially read in from a file which do not have to sum to 1. The normalisation method will calculate appropriate normalising constants and assign each rule a well defined probability.

I have implemented the algorithms described in section 3.3 to extend this module and they are briefly discussed in this section. Annotated code is presented in the Appendices.

4.2 Simulation

The module SCFG contains a method named 'sample' which can act on any grammar to output a sequence simulated from it. The method 'training_sequences' written in my extension of the SCFG module takes a single additional argument, namely the number of sequences required and it acts on a grammar to output the specified number of sequences simulated from that grammar. The sample function can also output the derivation of the sequence if the user specifies this as an additional argument. The form of this derivation takes the same form as the traceback returned by the CYK algorithm. This is particularly useful for comparing predicted annotations with the true annotations.

4.3 The CYK and the Inside algorithm

The presence of epsilon or null string productions (denoted by ε) complicate parsing. More specifically, elements in the recursions of the CYK and inside (-outside) algorithms can get into self-referencing loops. For example, let U and V denote any arbitrary non-terminals; if $V \rightarrow \varepsilon$ and $U \rightarrow VU$ then a string s can be generated from U either by $U \rightarrow s$ or $U \rightarrow VU \rightarrow U \rightarrow s$ and a loop has been constructed where the probability of generating the string s from U depends on the probability of generating s from V. This is a phenomena that is best avoided and indeed can be avoided provided the grammar of concern is incapable of producing the null string.

Context-free grammars that are not capable of producing the null string are known as ε-free grammars. This does not necessarily mean that the null production is not present in any of the production rules, however it can be shown, see [10], that the production rules for an ε-free grammar can be rewritten to eliminate any instance of the null production. An algorithm for doing this written by Rune Lyngsø and is found in Appendix A.1.1. It pushes any epsilon production up through the grammar eliminating it by introducing new production rules. This algorithm may introduce multiple copies of the same production rule but it is important to retain each copy at least initially because their presence will affect each of the algorithms in a different way.

The quantity computed by the CYK algorithm is the probability of the most likely derivation (and this derivation itself), so the copy of the rule used must be the one with the highest probability assigned to it. It is possible to collapse the rules into one by simply neglecting the rules with lower probabilities and retaining the copy with

the highest probability. This is formalised by algorithm 8 in Appendix A.1. Note that the resulting SCFG is not strictly well defined because the probabilities assigned to rules from non-terminals which have multiple copies of rules, sum to a quantity less than one. This is not a problem for implementation of the CYK algorithm.

If the inside or outside algorithm is to be implemented, each instance of a production rule is important and it is necessary to sum the probabilities assigned to each copy of a single rule since the algorithms sum over all possible ways of deriving the same sequence. This is formalised by algorithm 9 in Appendix A.1.

Self-referencing loops can also be a consequence of replacement productions. They are rules where a single non-terminal is replaced by another. They can cause problems in the same way as ε productions and can be eliminated using a similar algorithm. Algorithms 6 and 7 which both work to eliminate replacement productions, are presented in appendix A.1.

The inside and the CYK algorithm are very similar and consequently I have written one method which can either perform the CYK algorithm or the inside algorithm on a sequence according to the arguments it is given. The default is to perform the inside algorithm on the sequence. If the CYK algorithm is run the output of the method is a traceback of the most likely parse of the sequence, otherwise the inside algorithm is run and the matrix of α quantities defined in the section 3.3.3 is returned. The code is presented in Appendix A.3.

4.4 The Outside Algorithm

The outside algorithm takes the sequence as an argument. The method returns the matrix of β quantities defined in section 3.3.3.

A simple check that these algorithms are working consistently is to compute the probability of a sequence being generated by a fully specified grammar in two ways:

1. Run the inside algorithm on the sequence. From the returned matrix extract the probability with which the SCFG can generate such a sequence. This corresponds to simply extracting the element $\alpha(0, n, 0)$
2. Run the outside algorithm on the sequence. From the returned matrix extract the probability with which the SCFG can generate such a sequence. This requires a simple additional calculation;

If the algorithms are both working correctly then the probabilities calculated by each of the two methods should be identical. I have written a program to perform the above for my implementation to check it is working correctly. The script for this program is included in Appendix A.5. The output confirmed that my implementations are consistent. In addition, I performed the inside algorithm by hand for a very simple sequence to check that I also agreed with the program output. It is important that these algorithms are working correctly because they are used repeatedly within parameter training.

4.5 Inside-Outside Parameter Training

The parameter training algorithm requires the specification of initial parameters for the SCFG of interest in addition to a set of training sequences whose structure is being modelled by the SCFG. The algorithm should be run several times from different initial starting values due to the local maxima problem. This is done by writing the training algorithm to start with random initial parameters and running it a number of times (typically 5 in my case) in attempt to find the global maximum.

Irrespective of the starting points chosen to initiate the EM algorithm, each subsequent iteration increases the loglikelihood of the data given the new model. The convergence of the loglikelihood to a local maximum can be seen by monitoring the change in loglikelihood. I impose a threshold of 0.0005 such that the algorithm continues the iterations until the change in loglikelihood is less than this threshold. At this point I assume that the parameters of the grammar have converged sufficiently and output the values. Results and plots of the loglikelihood using these implementations will be presented in section 8. The code for the parameter training can be found in Appendix A.3 and the script used is included in Appendix A.6.

5 Independence

A HMM can be used to model the location of coding and non-coding regions on a DNA sequence and a SCFG can be used to model its secondary RNA structure. Using these models it is possible to annotate a sequence with its most likely gene regions and secondary structure respectively. The method widely used to predict gene locations and secondary structure is to consider both of these models separately. Parameters of a gene finding HMM are estimated via Baum-Welch training and the Viterbi Algorithm is then applied with these estimated parameters to annotate a test sequence with the most probable location of coding regions. The same training sequences are then used to estimate the parameters of a SCFG modelling secondary structure and the CYK Algorithm is then applied to predict the most likely secondary structure.

Informally, secondary structure and location of coding regions within a given sequence are independent if knowledge of the secondary structure is irrelevant to the location of coding regions and vice versa. At the opposite end of the scale (and not biologically correct!) suppose that base pairs only occur in secondary structure when the nucleotides are non-coding and that loops occur only where the nucleotides are coding. Then specification of secondary structure automatically specifies coding regions. Although this assumption is not realistic it demonstrates the idea. In reality it is more likely that a particular secondary structure will affect the probabilities of the possible location of coding regions and visa versa. If they affect each other they are considered to be dependent.

More formally, consider a DNA sequence denoted s, and use X_s and Y_s to denote its gene and secondary structure respectively. Then X_s and Y_s can both be represented as random variables which in the simple case take vector values with dimension same as the sequence length. The coding random vector has entries which are either zero or one according to whether the nucleotide is coding or not. The structure random vector in a simple case will be of the same dimension and has entries zero or one according to whether the corresponding nucleotide in the sequence is part of a base pair or not. In this simple case, these random vectors are discrete with only finitely many possibilities.

If X, Y and S are considered as discrete random variables corresponding to gene structure, secondary structure and the DNA sequence respectively, (although strictly the sample spaces for S, X and Y are infinite if the sequences are permitted to grow arbitrarily large), then it is possible to use the elementary definition of independence. That is U and V (where U and V are discrete) are independent if

$$P(U = u, V = v) = P(U = u)P(V = v) \qquad \forall\, u \in \mathcal{U},\ v \in \mathcal{V} \tag{2}$$

The sample spaces of U and V are denoted by $\mathcal{U}$ and $\mathcal{V}$ respectively.

It is clear that X and Y cannot be independent since they can only be well defined conditional on some sequence S. Hence it is appropriate to consider what it means for two random variables, X and Y to be **conditionally independent** given a third random variable Z. Formally discrete random variables X an Y are conditionally independent given Z if

$$L(X|Y, Z) = L(X|Z) \tag{3}$$

When X and Y are discrete, provided $P(Y = y, Z = z) > 0$ for all $y \in \mathcal{Y}$ and $z \in \mathcal{Z}$ this definition is equivalent to condition (4).

$$P(X = x, Y = y|Z = z) = P(X = x|Z = z)P(Y = y|Z = z) \tag{4}$$

In this framework it is sensible and valid to ask if the location of coding regions X and the secondary structure Y are conditionally independent given a sequence $S = s$. If this is the case then it follows that

$$\begin{aligned} P(X = x, Y = y|S = s) &= P(X = x|S = s)P(Y = y|S = s) \\ &= P(X_s = x)P(Y_s = y) \end{aligned} \tag{5}$$

It remains to link these definitions of conditional independence for the structure and coding regions given a sequence with the HMM and SCFG that are used to model these random variables. Specifying M, a fully parameterised HMM to model the location of coding regions provides a way of imposing a probability distribution on X_S. Similarly specifying G a fully parameterised SCFG to model RNA secondary structure provides a way of imposing a probability distribution on Y_S. The distribution of X_S, Y_S imposed by the models have no convenient closed form since they are dependent on two things:

1. The sequence.
2. The structure and parameters of the HMM or the SCFG which may also be complex according to the features of the structure and gene regions which the grammars can model.

One could think of the distribution of the coding locations of a sequence as being parameterised by the HMM and the sequence itself. Similarly, the distribution of the secondary structure of a sequence is parameterised by the SCFG and the sequence of interest.

How does this relate to what Bioinformaticians currently do?

Suppose there is a test DNA sequence s for which both the location of coding regions and the secondary structure is unknown. Then suppose that the bioinformatician wants to predict the location of genes within this sequence with M, a known or estimated parameterised HMM. If Viterbi is used to do this, then taking the predicted location of genes corresponds to estimating X_s by the path (or vector) which is most likely given the HMM. Denote the predicted coding locations by $\tilde{x}_s$. Now suppose that the bioinformatician also wishes to know or predict the secondary structure in addition to the location of coding regions. Typically the process above is repeated but with a known or estimated SCFG (G), then the secondary structure Y_s is estimated by the structure which is most likely under the SCFG model. Denote the predicted most likely structure by $\tilde{y}_s$. i.e,

$$\begin{aligned} \tilde{x}_s &= \arg\max_{x\in\mathcal{X}} P(X = x|s, M) && (6) \\ &= \arg\max_{x\in\mathcal{X}} P(X_s = x|M) \\ \tilde{y}_s &= \arg\max_{y\in\mathcal{Y}} P(Y = y|s, G) && (7) \\ &= \arg\max_{y\in\mathcal{Y}} P(Y_s = y|G) \end{aligned}$$

Then the bioinformatician may assume that $(\tilde{x}_s, \tilde{y}_s)$ is the most likely annotation of both the path and secondary structure. This is true if X and Y are conditionally independent given a sequence as demonstrated by the following derivation but cannot be shown to be true in general.

$$\begin{aligned} P(X = x, Y = y|S = s) &= P(X = x|S = s)(Y = y|S = s) && (8) \\ &= P(X_s = x)P(Y_s = y) \\ \Rightarrow \arg\max_{x,y\in\mathcal{X}\times\mathcal{Y}} P(X_s = x, Y_s = y) &= \arg\max_{x\in\mathcal{X}} P(X_s = x) \times \arg\max_{y\in\mathcal{Y}} P(Y_s = y) \end{aligned}$$

It is under this assumption, a HMM can be used to model $P(X_s = x)$ and a SCFG can be used to model $P(Y_s = y)$. So,

$$\begin{aligned} \arg\max_{x\in\mathcal{X}} P(X_s = x) \times \arg\max_{y\in\mathcal{Y}} P(Y_s = y) &= \arg\max_{x\in\mathcal{X}} P(X_s = x|M) \times \arg\max_{y\in\mathcal{Y}} P(Y_s = y|G) && (9) \\ &= \tilde{x}_s\tilde{y}_s \end{aligned}$$

Note that this assumption of conditional independence implies that the domain of the joint density is the product space $\mathcal{X} \times \mathcal{Y}$, however if this is not a valid assumption, the joint density may only be non-zero on a subset of this domain. The previous extreme example demonstrates this point; there is a 1:1 correspondence such that x_s completely specifies y_s i.e only points (x, x) permitted in the joint density.

In practise, making the first assumption that X and Y are conditionally independent given S is primarily a biological one and one which is difficult to justify. Although a test for determining whether this is appropriate is discussed later in this section. In the above it is assumed that model parameters are already known, but they are usually obtained via training on the same set of training sequences. So in some sense the parameters of one model will contain information about the parameters of the other model although any information is intractable due to the complexity and random initialisation of the algorithms used to derive the parameters.

If conditional independence is not an appropriate assumption, the models M and G are inappropriate, and even if this is a valid assumption, then the grammars are dependent on each other in some way because their parameters are estimated from the same data.

CODING STATE	STRUCTURAL STATE	NO. OBSERVED	NO. EXPECTED
Non-Coding	Base pairing	n_{01}	$\frac{n_{01}+n_{00}}{n} \times \frac{n_{11}+n_{01}}{n} \times n$
Non-Coding	Loop	n_{00}	$\frac{n_{01}+n_{00}}{n} \times \frac{n_{00}+n_{10}}{n} \times n$
Coding	Base pairing	n_{11}	$\frac{n_{10}+n_{11}}{n} \times \frac{n_{00}+n_{10}}{n} \times n$
Coding	Loop	n_{10}	$\frac{n_{11}+n_{10}}{n} \times \frac{n_{11}+n_{01}}{n} \times n$

Table 3: A table to be completed to test for independence

Evaluating the assumption that X and Y are conditionally independent given a sequence is difficult if there are not training sequences available for which the structure and the location of coding regions are known. If this training data is available it is possible to construct a data dependent test for independence of X_s and Y_s. In the simple case, there are only two possible features in the secondary structure so a single base either forms part of a base-pair or it forms part of a loop and likewise, there are only two possible features in the gene structure, either a nucleotide is part of a coding region or it isn't.

The elementary definition of independence is given in terms of events. Two events A and B are independent if the probability of observing event A and observing event B simultaneously is simply the product of the individual probabilities. That is,

$$P(A \cap B) = P(A)P(B)$$

Conditional independence can be defined in an analogous way. Let C be another event. Then event A is conditionally independent of event B given event C if

$$P(A \cap B|C) = P(A|C)P(B|C)$$

Define the following events:

1. Event A to be the event that a nucleotide drawn at random from a sequence s is part of a coding region.
2. Event B to be the event that a nucleotide drawn at random from a sequence s is part of a base pair forming a stem.
3. Event C to be the event that the sequence s is observed with coding regions X_s and secondary structure Y_s

Then given training data C, the probability that a nucleotide chosen at random from any of the training sequences is part of a coding region is the proportion of coding nucleotides in the entire sequence. In a similar way the probability that a nucleotide chosen at random forms part of a stem in the secondary structure is just the proportion of nucleotides in the entire sequence with this property. It can be extended in the obvious way if multiple training sequences are available.

The elementary definition of conditional independence in terms of events can now be applied. If events A and B are conditionally independent given C, then the probability of a nucleotide being part of a coding region (or not) and forming part of a stem (or not) is simply the product of the two proportions. Taking this to be the null hypothesis it is possible to use Fisher's exact test or a chi-squared test to determine whether the events A and B are conditionally independent given the sequence training data. A method formalising this test is presented below. It can only be used if training sequences are available for which the secondary structure and the location of coding regions are known.

1. Suppose there are a total of n training nucleotides. Draw up a two way contingency table for the data and complete table 3.
2. If all entries of the contingency table are large, use the table to calculate Pearson's Chi-Squared test statistic (summing the squared observed minus expected counts) and obtain an approximate p-value for the test. It can be easily obtained using the fact that with sufficient data Pearson's test statistic is approximately chi-squared distributed with 1 degree of freedom under the null hypothesis. Otherwise Fisher's exact test can be used to obtain an exact p-value. Typically a p-value smaller than 0.05 is sufficient evidence to reject the null hypothesis.

It must be noted that this test does not strictly test whether X_s and Y_s are independent. It tests whether particular features of X_s and Y_s are independent however in the simple case it provides a reasonable approximation. If more complex grammars are considered which model more features of the gene structure and RNA secondary structure, then this test will need modifying to include these features.

If the null hypothesis is rejected with this test, it is natural to investigate the nature of the dependence between X_s and Y_s although determining them is not such a simple task.

If it is possible to incorporate dependencies into a single model (which is my ultimate aim) then it would provide a tool to jointly annotate a sequence with secondary and gene structure. In other words I aim to construct and train a new grammar to model the joint distribution and hence estimate (X_s,Y_s). If X_s and Y_s are dependent then one would expect this way of estimating (X_s,Y_s) to be different to the estimates obtained by using the HMM and the SCFG separately.

Having determined that current methods for estimating (X_s,Y_s) assume that X_s and Y_s are independent, I attempt to develop a combined model which can incorporate dependence. It would also be nice to be able to test how a combined model performs relative to existing methods. This would certainly be possible if the probabilities assigned to production rules in the combined grammar could be set in such a way that the combined annotation is the same as that of the two separate annotations. If this is possible then a nested likelihood ratio test could be performed to show whether the combined model fits the data better than the two separate models.

The next section investigates two things, the construction of a joint model via the combination of a simple HMM and a simple SCFG and the possibility of constructing a statistical test for independence based on the specification of parameters in the joint model.

6 Developing a method for combining a general stochastic context-free grammar with a hidden Markov model

The production rules for an ε-free SCFG can be expressed in CNF by terminal and non-terminal productions. Developing a method to combine terminal production rules and non-terminal productions rules with the hidden states of a HMM would therefore provide a general method to combine a general SCFG with a HMM. In addition to generalising the grammar I can also investigate how to develop the method to combine a SCFG with a more general HMM with a mixture of silent and non-silent states. Developing a generalised method provides a tool for bioinformaticians to investigate combining more realistic grammars and HMMs.

It is possible to incorporate the Markov property into a SCFG by changing the probability distribution over the set of rules according to the state of any terminals produced in the previous rule. This will allow for the same terminals to be emitted from different states in the HMM with different probabilities according to the type of production rule. In itself this is not sufficient. The state to which the HMM makes a transition to after emitting a symbol also needs to be accounted for. Although more complicated, the resulting model is another SCFG with more non-terminals and production rules. This is not surprising because the intersection of a regular language (i.e the set of all strings that can be produced by a regular grammar) and a context-free language is a context-free language. To understand why this is the case and how non-terminals and production rules should be introduced, I use the following notation and set-up.

6.1 Set-up and notation

The SCFGs which are of interest in this project are those which can be used to model RNA secondary structure or simple analogues of such grammars. Grammars capable of producing the null string are not useful since we are interested in the derivations or parses of non-null sequences. For this reason it is justified to consider only ε-free SCFG's. Further since all replacement productions can also be eliminated from the production rules, the class of SCFG's considered can be restricted to be ε-free with no replacement productions. However as will be discussed later in section 8.1.1 eliminating replacement productions can introduce more parameters than necessary into the model.

Gene finding HMMs typically contain a mix of hidden and silent states. For technical reasons to be later discussed, at this stage I consider HMMs with only two silent hidden states: the begin and end states. I allow the HMM to have an arbitrary number of non-silent hidden states.

Let G be an ε-free SCFG without replacement productions specified by:

1. Terminal Alphabet T
2. Start Non-Terminal S
3. Non-Terminal Alphabet V (containing the start non-terminal S)
4. Production rules P
5. Parameters $\theta_1, \theta_2, ...\theta_{|V|}$, corresponding to the probability distributions over possible rules from each non-terminal contained in V.

Let M be a hidden Markov model specified by:

1. Non-silent hidden states N={$n_1, n_2, ..., n_k$} that can emit symbols from the terminal alphabet T (corresponding to that in the context free grammar above)
2. Silent hidden states S={$\tilde{b}, \tilde{e}$}, where $\tilde{b}$ is the begin state and $\tilde{e}$ is the end state.
3. Parameters ($\phi_1, \phi_2, .., \phi_k, Q$), where ϕ_i are emission probability distributions corresponding to hidden states $n_1, ..n_k$ and Q is the matrix of transition probabilities.

Let G^* denote the combined context-free grammar produced when M and G are combined. Clearly G^* will have the same terminal alphabet T but I need to develop an algorithm for generating the new combined set of production rules P^* and the new set of non-terminal symbols V^*.

Introduce the following new notation:

- Let W_{ij} be a non-terminal which can emit a terminal symbol (either directly or indirectly) from non-silent state i then make a series of transitions through non-silent states to non-silent state j. Subsequently the first nucleotide of the subsequence derived from the nonterminal W_{ij} will be emitted from state i in the HMM. The subscript j does not restrict the last symbol of the subsequence to be emitted from state j; it means that a transition is made to state j after the last nucleotide of the subsequence is emitted.
- Let $W_{i\tilde{e}}$ be a non-terminal which can produce a terminal symbol from non-silent state i and make a series of transitions with the final one made to the end state.

6.2 Developing a general algorithm for generating the combined grammar

To combine the SCFG denoted G and the hidden Markov model M, it is necessary to go through all non-terminals contained in V and systematically generate the new set of non-terminals V^* and similarly go the production rules of P to generate rules P^* of the combined grammar G^*.

The motivation behind the introduction of subscripted non-terminals stems from the CYK algorithm; the parse of a sequence is built up from parses of substrings. The idea with the combined grammar is to build up the path of hidden states (from the HMM) together with the structure parse via patching substrings together until the complete path is constructed.

6.2.1 Generating the non-terminal alphabet V^* for the combined grammar

A good starting point is to construct the new non-terminal alphabet V^*. Once this is complete it is possible to formulate the production rules from this set together with the framework provided by the production rules of the original SCFG.

The set of non-terminals for the combined grammar must by definition of the grammar contain the start non-terminal S^*. This is always the first element of the non-terminal alphabet. Set $S^* = S_{\tilde{b}\tilde{e}}$ since the first transition is made from the silent begin state state and the last transition is made to the silent end state.

Now consider the case when the start non-terminal $\{S \in V\}$ appears only on the left hand side of any production rule in P. Then, for each non-terminal in $V \setminus S$, a new non-terminal is required in V^* which can emit either directly or indirectly by a series of subsequent production rules a string for which the first terminal is emitted from state i of the HMM and a series of transitions is made through potentially a mixture of silent and non-silent states ending in state j, a non-silent state in M. Note that if the start non-terminal in V appears on the right hand side in any production rules (contained in P) it is necessary to add S_{ij} to V^* for all $i, j \in N \cup \tilde{e}$.

The state from which the first nucleotide is emitted needs to be accounted for. This is specified by rules used from the start non-terminal and can be done by adding non-terminals $S_{i\tilde{e}}$ where i is the first non-silent state reached in the HMM and $\tilde{e}$ is the end state visited by the chain. The notation W_{ij} denotes the event that a transition to be made to state j with no terminal emission from it, so the end state is valid in the second subscript of a non-terminal (just as for the simple case). To allow for all possible first non-silent states the non-terminals $S_{i\tilde{e}}$ must be added for all possible $i \in N$. For consistency it is necessary to add the non-terminals $W_{i\tilde{e}}$ for all $i \in N$ and for all other non-terminals contained in V.

The process of constructing the non-terminal alphabet of the combined grammar described above can be formulated by algorithm 1.

6.2.2 Generating the production rules P^* for the combined grammar

Since G is an ε-free grammar it has production rules of two types: terminal and non-terminal productions.

Without loss of generality assume that $S \in V$ occurs in the right hand side of a production rule contained in P and S_{ij} is added to the non-terminal alphabet. If this is not the case then production rules containing these

Algorithm 1 Generating the non-terminal alphabet V^* for the combined grammar

Set V^* = $\{S_{\tilde{b}\tilde{e}}\}$
if S appears on the right hand side of any production rule $\in P$ **then**
  for $X \in V$ **do**
    for $i \in N$ **do**
      for $j \in N \cup \{\tilde{e}\}$ **do**
        $V^* \leftarrow V^* \cup \{X_{ij}\}$
else
  for $X \in V$ **do**
    for $i \in N$ **do**
      $V^* \leftarrow V^* \cup \{X_{i\tilde{e}}\}$
for $X \in \{V \setminus S\}$ **do**
  for $i \in N$ **do**
    for $j \in N$ **do**
      $V^* \leftarrow V^* \cup \{X_{ij}\}$

non-terminals (S_{ij}) should not be included in the final set of production rules P^*, but they can be eliminated from the non-terminal alphabet after applying algorithm 2.

First go through the terminal productions of P, for any non-terminal $W \in V$ there exists $W_{ij} \in V^*$ (for all $i \in N$ and $j \in N \cup \{\tilde{e}\}$). If $W \rightarrow \alpha \in P$ it is necessary to add the production rules $W_{ij} \rightarrow \alpha$ provided that there a transition from state i to state j, where $i \in N$ and $j \in N \cup \{\tilde{e}\}$.

Now consider non-terminal production rules. If the rule $W \rightarrow XY$, $(W, X, Y \in V)$ is present in P, there are many possible production rules that need to be added to form P^*. To understand which rules are valid additions to P^* recall that for non-terminal $W \in V$ there exists $W_{ij} \in V^*$ for all $i \in N$ and $j \in \{N \cup \tilde{e}\}$. This notation means that if the nonterminal W_{ij} is the root of a subsequence, the first symbol of this subsequence is emitted from state i then a series of subsequent transitions are made through silent and non-silent states ending with a transition (but no emission) to state j which may either be a non-silent state or the silent end state. Then it must be the case that any non-terminal production rule replacing W_{ij} with two non-terminals in V^* has i as the first subscript of the first non-terminal and j as the second subscript of the second non-terminal. The remaining subscripts must be identical to ensure continuity (since two substrings are patched together by use of this production rule). Using this reasoning it is clear that it is necessary to add rules of the form $W_{ij} \rightarrow X_{ik}Y_{kj}$ for $i, k \in N$ and $j \in N \cup \{\tilde{e}\}$.

Algorithm 2 Generating the production rules P^* for the combined grammar

Set P^* = $\{\emptyset\}$
for $X \in V$ **do**
  for $Y \in V$ **do**
    for $Z \in V$ **do**
      for $i \in N$ **do**
        for $k \in N$ **do**
          for $j \in N \cup \{\tilde{e}\}$ **do**
            if $X \rightarrow YZ \in P$ **then**
              $P^* \leftarrow \{P^*\} \cup \{X_{ij} \rightarrow Y_{ik}Z_{kj}\}$
for $X \in V$ **do**
  for $Y \in V$ **do**
    if $S \rightarrow XY \in P$ **then**
      for $i \in N \setminus B$ **do**
        for $k \in N$ **do**
          $P^* \leftarrow \{P^*\} \cup \{S_{\tilde{b}\tilde{e}} \rightarrow X_{ik}Y_{k\tilde{e}}\}$

How to incorporate the begin and end states with the start non-terminal is non-trivial. The start non-terminal of the combined grammar is $S^* = S_{\tilde{b}\tilde{e}}$. The possible productions on the right hand side need to specify the first non-silent state and end state for example, $S_{\tilde{b}\tilde{e}} \rightarrow S_{i\tilde{e}}$ where i is the first non-silent state and $\tilde{e}$ is the end state in

the chain. However this is a replacement production and rules of this type are not permitted in Chomsky's normal form, but they can be eliminated using algorithm A.1.2 in Appendix A.1. This corresponds to adding rules of the form $S_{\tilde{b}\tilde{e}} \rightarrow X_{ik}Y_{k\tilde{e}}$ for all i where a transition exists from $\tilde{b}$ to i, and $k \in N$.

Applying algorithm A.1.2 from appendix A.1 can introduce complications a few special cases. For example, if the sequence S consists of a single nucleotide. The nucleotide will necessarily be produced by $S_{be} \rightarrow a|b$, neither of these rules specify from which state of the HMM the terminal is emitted. To resolve this problem it is necessary to retain multiple copies of identical rules and the probabilities assigned to each one. As discussed in section 4.3, the information required from the two copies is different according to the algorithm being implemented. For simulation, the inside and the outside algorithm the sum of these two probabilities will be required whereas if the CYK algorithm is being implemented copy of the rule assigned the maximum probability will always be the copy used.

In theory eliminating replacement productions seems like a valid way to proceed and is the method I used in the example presented in figure 7, but later investigation and inspection of my results revealed that elimination of replacement rules introduces unnecessary additional parameters to the model. This will be discussed with examples in section 8.1.1.

This method of generating the production rules for the combined grammar can be formalised by algorithm 2. The set B is used to denote the set containing all non-silent states k for which there does not exist a path from the begin state to state k which does not pass through any other non-silent hidden state. I have not yet implemented these algorithms but if future investigation into other techniques for parameter estimation are successful and the CYK predictions perform better relative to the true annotation, then the combined model can be of practical use and implementation of these algorithms will be appropriate. I performed the algorithms by hand to combine M_1 the HMM specified in section 2 in figure 1 with the palindrome spacer grammar G_1 specified in section 3 in figure 4. The resulting grammar G^* is given in figure 7 where $i, k \in \{0, 1\}$, $r \in \{1, 2, 3, 4\}$ and $j \in \{0, 1, \tilde{e}\}$.

Figure 7 Specification of the combined grammar G^* in CNF

$$
\begin{aligned}
T^* &= \{a, b\} \\
S^* &= S_{be} \\
V^* &= \{S_{be},\, S_{ij},\, W_{ij}^r : i \in \{0,1\}, j \in \{0,1,e\}, r \in \{1,2,3,4\}\} \\
P^* &= \{S_{be} \rightarrow W_{ik}^1 W_{ke}^1 | W_{ik}^1 W_{ke}^2 | W_{ik}^3 W_{ke}^3 | W_{ik}^3 W_{ke}^4 | W_{ik}^1 S_{ke} | S_{ik} W_{ke}^1 | S_{ik} W_{ke}^3 | W_{ik}^3 S_{ke} | a | b \\
&\quad\; W_{ij}^1 \rightarrow a \\
&\quad\; W_{ij}^3 \rightarrow b \\
&\quad\; W_{ij}^2 \rightarrow S_{ik} W_{kj}^1 \\
&\quad\; W_{ij}^4 \rightarrow S_{ik} W_{kj}^3 \\
&\quad\; S_{ij} \rightarrow W_{ik}^1 W_{kj}^1 | W_{ik}^1 W_{kj}^2 | W_{ik}^3 W_{kj}^3 | W_{ik}^3 W_{kj}^4 | W_{ik}^1 S_{kj} | S_{ik} W_{kj}^1 | S_{ik} W_{kj}^3 | W_{ik}^3 S_{kj} | a | b\}
\end{aligned}
$$

This method combines the HMM and the SCFG in the sense that each of the original rules in the grammar are present but with different copies (labelled differently) which can be assigned different probabilities according to the hidden state of the HMM a symbol is emitted from, and to which a subsequent transition is made. By adjusting these probabilities it is possible to model dependence between the secondary structure and the location of coding regions along a sequence. The method described above does this in such a way that the resulting combined SCFG is specified with its production rules in CNF.

It is interesting and useful to note that the combined model can be reduced to a SCFG and that it is not necessary to use a context-sensitive grammar despite the fact the same grammar could be represented in this way in a more compact form. This is a nice result since all the algorithms associated with an SCFG can be implemented and run more efficiently than the corresponding algorithms for a context sensitive grammar.

In the example above there is a total of 31 variables in the combined grammar with 149 parameters to be assigned. With so many parameters to estimate, it is likely that a large number of sequences will be necessary for training to

obtain accurate parameter estimates. Following training of the combined SCFG, joint annotation of test sequences with their most likely secondary and gene structure can be done via the CYK algorithm. The secondary structure and coding states of each nucleotide can be extracted from the CYK most likely traceback according to the non-terminals used to derive the sequence.

6.3 Assigning parameters to the combined grammar

If a data dependent test for independence is performed according to the method proposed in section 5 and the null hypothesis (i.e X_s and Y_s are independent) is rejected then it is appropriate to model the joint distribution (X_s,Y_s). Use of the combined grammar provides a way of doing exactly this, provided sufficient data is available to obtain parameter estimates either via supervised or unsupervised learning. However, the independence test can only be performed if training data with known gene and secondary structure is available. In practise this data is unlikely to be available and it is even less likely that there will be sufficient amounts to be able to train the combined SCFG using maximum likelihood methods. In which case it would be better to be able to test and evaluate whether the joint model or the two independent models fit the test data best.

Such a test would certainly be possible if the models are nested. The likelihood of the data and estimates for the structure and coding locations from the two independent models could be obtained by setting the parameters in the joint model appropriately. If this is the case the independent models can be interpreted as a special instance of the joint model. To investigate whether the independent models are nested, given parameters for the original SCFG the HMM from which the combined grammar is derived, it is natural to start by asking whether it is possible to derive probabilities which can be assigned to the combined SCFG production rules in such a way that the joint annotation produces the same result had a sequence been annotated with secondary structure and coding regions independently. This in itself would not guarantee that the corresponding likelihood of these annotations would be the same as under the independent models but it provides a starting point.

The natural way to compute parameters for the combined grammar when the models are independent is to multiply appropriate transition and production rule probabilities. It is not obvious how this should be done or even if it will yield a well defined set of parameters. For this reason, the 'independent parameters' for the combined SCFG are referred to as weights. Consider the combined grammar G^* and recall two things:

1. The non-terminal W_{ij} is used to denote that the subsequence of terminals derived from W_{ij} starts in state i of the HMM and then makes an arbitrary number of transitions through non-silent hidden states until finally the last transition is made to state j but no emission is made from this state.
2. All non-terminals in the combined grammar are subscripted with indices, and all production rules in the combined grammar are derived from a corresponding rule in the original SCFG. For example, the rules $S_{ij} \to a|b$ are derived from the rules $S \to a|b$ in the original SCFG.

The production rules for the combined SCFG are specified in CNF and the weights to be assigned to the two types of rule are considered separately. The first rules are of the form $X_{ij} \to \alpha$. For all rules of this type, the subsequence derived from the non-terminal in question is a single terminal symbol. So it must be the case that if a series of transitions are made through other states in the HMM before making the transition to state j then they must be through silent states. In the restricted set of HMMs I consider, there are no such silent states, so the path must go directly from state i to state j. This means the weight assigned to the production rule should be the probability assigned to the production rule in the original SCFG (from which it is derived) multiplied by probability of emitting the terminal symbol from state i, also multiplied by the probability of making a transition from state i to state j. This is true provided the non-terminal on the left is not the start non-terminal. $X_{ij} = S_{\tilde{b}\tilde{e}}$ is a special case. If a nucleotide is emitted directly from the start non-terminal, then there will be multiple copies of the same rule assigned different weights, one for each non-silent state of the HMM. Then for the state j copy of the rule, the 'independent' weight requires and extra multiplication by the probability of making the transition from the begin state to state j of the HMM.

To formalise this idea for G and M, let ω_{p^*} be the weight assigned to the production rule $p^* \in P^*$, where p^* is derived from production rule $p \in P$ of G. Without loss of generality let the non-terminal on the left hand side of the production rule p^* have subscripts ij and let the terminal on the right hand side of the production rule be α. Then if p^* is of type one, the weight this rule is given in the 'independent' combined grammar can be computed

by formula (10) provided the terminal is not emitted from the start non-terminal.

$$\omega_{p^*} = \mathbb{P}(p) \times q_{ij} \times e_i(\alpha) \qquad i \in N,\, j \in N \cup \{\tilde{e}\} \tag{10}$$

The start non-terminal is a special case and the weights for the k copies of the terminal production rules can be assigned using formula (11), where p_j^* denotes the copy of the production rule corresponding to emission fron state j of the HMM.

$$\omega_{p_j^*} = \mathbb{P}(p) \times q_{\tilde{b}j} \times e_j(\alpha) \times q_{j\tilde{e}} \qquad j \in N \tag{11}$$

Now consider production rules which are of the second type; the non-terminal on the left is replaced by two non-terminals on the right. There is no subsequence of terminals emitted directly from the non-terminal on the left hand side so there are no additional weights to be incorporated from the HMM provided $X_{ij} \neq S_{\tilde{b}\tilde{e}}$. So the weights assigned to the non-terminal production rules for $X_{ij} \neq S_{\tilde{b}\tilde{e}}$ are simply the probabilities assigned to the corresponding rule in the original grammar as specified by formula (12).

$$\omega_{p^*} = \mathbb{P}(p) \tag{12}$$

The start non-terminal is a special case; rules will be of the form $S_{\tilde{b}\tilde{e}} \to X_{jl}Y_{lk}$ and they must be weighted by the probability of staring in state j of the HMM. Then the 'independent weights' for rules of this form can be set by using formula (13).

$$\omega_{p^*} = \mathbb{P}(p) \times q_{\tilde{b}j} \tag{13}$$

To determine whether these weights impose a well defined probability distribution over the rules of the combined SCFG consider the combined grammar constructed from G_1 and M_1 specified in figure 7. Take the production rule $W_{ij}^1 \to a$ from the combined SCFG (for any $i \in \{0, 1\}$, $j \in \{0, 1, e\}$). There are no other production rules with W_{ij}^1 on the left hand side so it must be the case that whenever the non-terminal W_{ij}^1 is used it is replaced the terminal a with probability 1. However the weight calculated according to (10) does not in general equate to one. It is clear that the weights are indeed weights and will not yield well defined probability distributions over the production rules.

If weights for the combined grammar production rules are assigned using the rules above, then it follows that the weight assigned to a joint annotation of a sequence will be the product of the probability of the structure given the original grammar and the probability of the gene annotation given the HMM. From this it follows that the joint annotation corresponding to the maximum weight will be exactly the independent annotations from the original models. It is not clear how the joint annotation will be affected if the weights are normalised to produce a well defined SCFG. To investigate this I consider a specific example where I set the parameters of the HMM and original SCFG.

The parameters of the HMM are specified according to figure 8. In particular, note that the HMM only emits symbols from the state into which it makes its first transition. Once in this state it is equally likely to remain in this state or make a transition to the end state. Symbols are emitted from state 0 with probability 2/3 and from state 1 with probability 1/3. The symbol emitted from state 0 is 'a' with probability 1/3 and 'b' with probability 2/3. The symbol emitted from state 1 is 'a' with probability 2/3 and 'b' with probability 1/3.

For simplicity set parameters some of the parameters in the original grammar to zero such that the remaining rules are

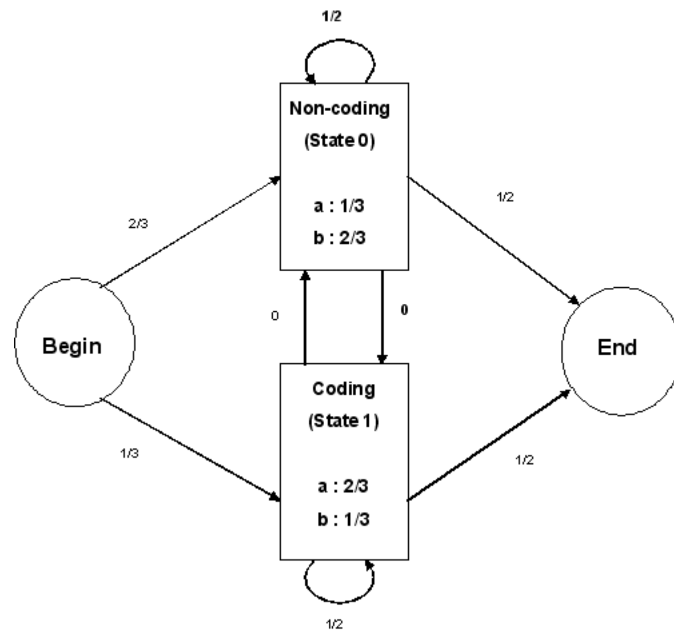
$$S \to aa|aS|a$$

Assigning probabilities and conversion to CNF yields the rules and probabilities displayed in table 4.

The combined grammar can be generated according to the algorithms described previously. Since many of the probabilities in the original grammar have been set to zero, the combined grammar is simpler than that of the full combined grammar specified in figure 7. The rules for this combined grammar with their independent weights and normalised probabilities are displayed in table 5.

Now calculate the probability of observing the sequence 'aa' with the various different models. There are only two possible gene annotations and two possible structure annotations so a total of 4 possible combined annotations. The probabilities and weights are displayed in table 6. Notice that the independent weights agree with the probabilities calculated from the separate models and that under these models the CYK prediction for the structure and coding regions would be that they are coding and not base pairing. Now look at the probabilities of the various

Figure 8 A diagram to show the parameters of the HMM used to show that normalisation of independent weights does not preserve CYK annotations



PRODUCTION RULE	PROBABILITY ASSIGNED
$S \rightarrow WW$	1/7
$S \rightarrow WS$	3/7
$S \rightarrow a$	3/7
$W \rightarrow a$	1

Table 4: A table to display the production rules of the original grammar in CNF and the probabilities assigned for the specific example to demonstrate that independent annotations are not preserved when the 'independent weights' for the combined grammar are normalised.

structures under the normalised combined model. The coding and non-based paired structure is $least$ likely and the CYK prediction is that the nucleotides are non-coding and base pairs. This is the opposite to the prediction under the independent models.

This example demonstrates the fact that normalisation of independent weights in the combined model does not preserve the independent most likely annotation. This is an interesting result and demonstrates that there is no obvious way in which the parameters of the combined grammar should be set such they yield the annotations given from the independent models. This suggests that without further research into non-nested model testing, the only independence test that can be performed is the data dependent method outlined in section 5.

This also raises another interesting point. Suppose that the combined SCFG is used to model (X_s, Y_s) when it is valid to assume that X_s and Y_s are independent. Then by using the combined model it may not be possible to produce the annotations which would be predicted via the use of the independent models. This suggests that using the joint model to estimate (X_s, Y_s) may introduce dependencies which are a feature of the model not the data. This is not a desirable feature of the combined model and it is advisable only to use it when the assumption of independence is not valid.

If it is assumed that the combined model is appropriate, the parameters must be estimated. Since the dimension of the parameter space is high, a relatively large number of training sequences are required to obtain accurate estimates using inside-outside training. The number of sequences required to obtain accurate estimates will be investigated and the results presented in section 8. If expectation maximisation techniques do not perform well,

PRODUCTION RULE	INDEPENDENT WEIGHT	NORMALISED PROBABILITY
$S_{\tilde{b}\tilde{e}} \rightarrow W_{00}W_{0\tilde{e}}$	2/21	1/7
$S_{\tilde{b}\tilde{e}} \rightarrow W_{11}W_{1\tilde{e}}$	1/21	1/14
$S_{\tilde{b}\tilde{e}} \rightarrow W_{00}S_{0\tilde{e}}$	6/21	3/7
$S_{\tilde{b}\tilde{e}} \rightarrow W_{11}S_{1\tilde{e}}$	3/21	3/14
$S_{\tilde{b}\tilde{e}} \rightarrow a$	1/21	1/14
$S_{\tilde{b}\tilde{e}} \rightarrow a$	1/21	1/14
$S_{0\tilde{e}} \rightarrow W_{00}W_{0\tilde{e}}$	1/7	2/9
$S_{0\tilde{e}} \rightarrow W_{00}S_{0\tilde{e}}$	3/7	2/3
$S_{0\tilde{e}} \rightarrow a$	1/14	1/9
$S_{1\tilde{e}} \rightarrow W_{11}W_{1\tilde{e}}$	1/7	1/5
$S_{1\tilde{e}} \rightarrow W_{11}S_{1\tilde{e}}$	3/7	3/5
$S_{1\tilde{e}} \rightarrow a$	1/7	1/5
$W_{00} \rightarrow a$	1/6	1
$W_{11} \rightarrow a$	1/3	1
$W_{0\tilde{e}} \rightarrow a$	1/6	1
$W_{1\tilde{e}} \rightarrow a$	1/3	1

Table 5: A table to display the production rules of the original grammar in CNF and the probabilities assigned for the specific example to demonstrate that independent annotations are not preserved when the 'independent weights' for the combined grammar are normalised.

GRAMMAR	GENE ANNOTATION	STRUCTURE ANNOTATION	WEIGHT/PROBABILITY
Separate models	00	..	1/54 × 9/49= 1/294
	00	()	1/54 × 1/7= 1/378
	11	..	1/27 × 9/49 = 1/147
	11	()	1/27 × 1/7= 1/189
Combined Model	00	..	6/21 × 1/6 × 1/14 = 1/294
(independent weights)	00	()	2/21 × 1/6 × 1/6 = 1/378
	11	..	3/21 × 1/3 × 1/7 = 1/147
	11	()	1/21 × 1/3 × 1/3 = 1/189
Combined Model	00	..	1/7 × 1 × 1 = 1/7
(normalised weights)	00	()	3/7 × 1/9 × 1 = 1/21
	11	..	3/14 × 1/5 × 1 = 3/70
	11	()	1/14 × 1 × 1 = 1/14

Table 6: A table to display the probabilities/weights for the possible structures of the sequence 'aa' under the separate models, the combined model with independent weights assigned to the rules and the combined model with normalised weights. It shows independent CYK annotations are not preserved when the 'independent weights' for the combined grammar are normalised.

it may be necessary to consider another approach. In particular Bayesian methods may perform better if the number of training sequences is relatively small. This will be discussed in the following section together will other potential problems which may be encountered when using the combined model to estimate X_s and Y_s jointly.

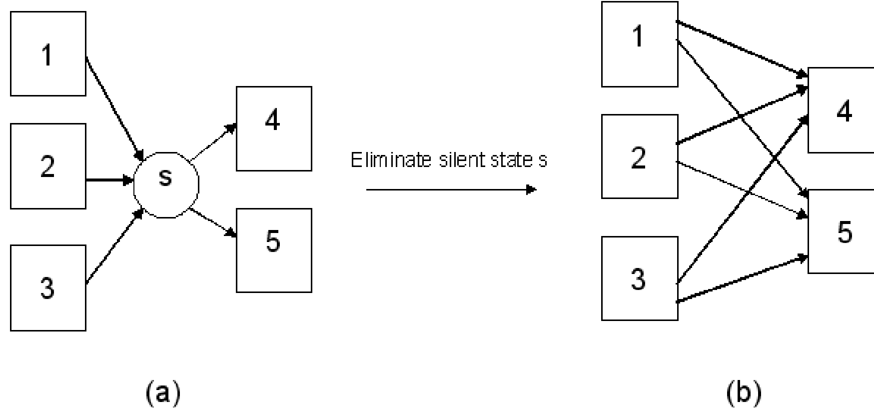
6.4 Combining SCFGs with HMMs with silent states

The construction of the combined grammar described previously can only combine a general SCFG with a HMM containing only non-silent states (excluding the begin and the end state). To account for silent states other than the begin and end state, more details need to be considered. In this section reference to silent states refers to silent

states excluding the begin and end state.

One approach would be to eliminate silent states by inserting direct transitions between non-silent states if there exist a path between two non-silent states which passes through any number of silent states. An example is given by figure 9.

Figure 9 A diagram to show how silent states can be eliminated from a HMM by introducing direct transitions between appropriate non-silent states



Insertion of such transitions requires the parameters of the resulting HMM to be updated (if the transition existed in the original HMM) or newly assigned otherwise. Let the new parameters be denoted q^*_{ij} where i, j are non-silent states in the HMM. Then $q^*_{i,j}$ must satisfy the following system of equations:

$$q^*_{ij} = q_{ij} + \sum_{i \to k : k \, silent} q_{ik} q^*_{kj} \tag{14}$$

The sum is effectively summing over the probabilities of all paths of silent states between i and j, and if a transition already exists between i and j then this must also be included.

These parameters are correct for simulation or implementation of the forward/backward algorithm however, if Viterbi is to be implemented the parameter (and path) stored is computed by taking the maximum (and argmax) rather than the sum:

$$q^*_{ij} = \max_{i \to k : k \, silent} \{q_{ik} q^*_{kj}, q_{ij}\} \tag{15}$$

It is possible to solve these systems numerically and proceed to combine the HMM which no longer has silent states with a SCFG using the method described previously. This is not entirely satisfactory because eliminating silent states introduces more parameters to the HMM than were originally there. This is evident from the example in figure 9. In the original HMM (a) with the silent state included there is only one free transition parameter but after elimination of the silent state (b) there are 3. This is not a problem if the true parameters of the HMM are known since the parameters in the derived HMM (with no silent states) can be set accordingly. However in this context the HMM to be combined has unknown parameters and the combined grammar will inherit the extra parameters.

An alternative approach to combining a HMM with silent states would be to allow the combined grammar to have replacement production rules. My current implementations cannot handle replacement productions, but provided there are no self-referencing cycles of non-terminals they can be modified to handle such rules. The restriction that no self-referencing loops are present in the combined grammar is equivalent to imposing the restriction that the HMM to be combined cannot have cycles of silent states between non-silent states. This is not an unreasonable restriction and many current gene finding HMMs comply to this.

To incorporate silent states of a HMM with a SCFG, introduce non-terminals and rules according to algorithm 2. Then, for all terminal productions of the form $V_{ij} \to \alpha$ where i, j are non-silent states in the HMM and i can make a direct transition to state j with a strictly positive probability add the following rules explained in table 7.

RULE INTRODUCED	NOTES
$V_{ij} \rightarrow \alpha V_{kj}$	Allow for transitions from a non-silent state i to a silent state k. Introduce rules of this form if there is a transition in the HMM from non-silent state i to silent state k.
$V_{kj} \rightarrow V_{lj}$	Allow for transitions through a series of silent states. Introduce rules of this form for silent states k and l and j non-silent if there is a transition from k to l in the HMM.
$V_{kj} \rightarrow \varepsilon$	Allow for transitions from a silent state back to a non-silent state. Introduce rules of this form if there exists a transition from silent state k to non-silent state j in the HMM.
$S_{\tilde{b}\tilde{e}} \rightarrow S_{k\tilde{e}}$	Allow for transitions from the begin state to silent states k provided there is a transition from the begin state to k in the HMM.
$S_{k\tilde{e}} \rightarrow S_{j\tilde{e}}$	Allow for the first transition to a non-silent state j from silent state k provided there is a transition from k to j in the HMM. k may be the begin state.

Table 7: A table to show and explain the rules which must be added to the combined grammar to handle silent states of a HMM

Allowing replacement productions also provides a more efficient way to specify the coding state of the first nucleotide of a sequence. In particular it removes the need to retain multiple copies of identical rules in the combined grammar. Allowing the rules $S_{\tilde{b}\tilde{e}} \rightarrow S_{i,\tilde{e}}$ for i non-silent and where there exists a transition from the begin state to i makes two (or more) copies of the rule $S_{\tilde{b}\tilde{e}} \rightarrow a|b$ unnecessary. It also reduces the number of parameters in the combined grammar.

The introduction of these rules can be done after algorithms 1 and 2 by using algorithm 3. Use the set-up described at the beginning of the section on page 21 with the modification that S is set of silent states such that $S \backslash \{\tilde{b}, \tilde{e}\} \neq \emptyset$.

Algorithm 3 Generating additional production rules for the combined grammar to incorporate silent states (other than the begin and end state) of the HMM being combined

for $i \in N$ **do**
  if begin state can make a transition to non-silent state i **then**
    Set P^*=$P^* \cup \{S_{\tilde{b}\tilde{e}} \rightarrow S_{i\tilde{e}}\}$
for $k \in S$ **do**
  if begin can make a transition to silent state k **then**
    Set $P^* = P^* \cup \{S_{\tilde{b}\tilde{e}} \rightarrow S_{k\tilde{e}}\}$
  for $j \in N$ **do**
    if k can make a transition to j in the HMM **then**
      Set $P^* = P^* \cup \{S_{k\tilde{e}} \rightarrow S_{j\tilde{e}}\}$
for $i \in N$ **do**
  for $j \in N \cup \{\tilde{e}\}$ **do**
    for $\alpha \in T$ **do**
      if $V_{ij} \rightarrow \alpha \in P^*$ **then**
        for $k \in S$ **do**
          if i can make a transition to k in the HMM **then**
            Set $P^* = P^* \cup \{V_{ij} \rightarrow \alpha V_{kj}\}$
for $j \in N \cup \{\tilde{e}\}$ **do**
  for $k, l \in S \backslash \{\tilde{b}, \tilde{e}\}$ **do**
    if k can make a transition to silent state l in the HMM **then**
      Set $P^* = P^* \cup \{V_{kj} \rightarrow V_{lj}\}$
for $j \in N \cup \{\tilde{e}\}$ **do**
  for $k \in S \backslash \{\tilde{b}, \tilde{e}\}$ **do**
    if k can make a transition to non-silent state j in the HMM **then**
      Set $P^* = P^* \cup \{V_{kj} \rightarrow \varepsilon\}$

7 Limitations of the Combined Model

7.1 Sensitivity to changes in parameters

The combined SCFG is more complex than either of the original models with 149 parameters to be assigned, even when the models being combined are simple. It is important to investigate how sensitive the annotation is to small changes in parameters. [3] suggests that sensitivity is high, even with simple grammars. If this is true for the combined SCFG it is important that the parameters can be estimated accurately. Otherwise small inaccuracies in the estimation of parameters may lead to large errors in predicted annotations for coding regions and secondary structure.

This also raises another interesting question: how well does inside-outside training estimate the parameters of a SCFG? This is something that can be investigated by simulating strings generated by a specified grammar and running Inside-Outside training on the sequences to recover estimates for the parameters.

If the combined grammar annotations are highly sensitive to small changes in parameters it is important to obtain accurate estimates. Although inside-outside training is widely used it may be possible to use a Bayesian approach which will estimate parameters more accurately given a relatively small training data set.

For the combined model to work in practise, it is necessary to have training sequences with which to run the parameter training algorithm. It is therefore important to know approximately how many training sequences are necessary to obtain sufficiently accurate parameter estimates. To investigate this, parameter training is run on a varying number of sequences. As the number of sequences used to train the grammar is increased one would expect the error of the estimates to decrease and to see the similarity to the true parameter annotation increase.

These tests are implemented for the combined grammar and results are presented with discussion in section 8.

7.2 Ambiguity

A grammar is said to be ambiguous when there is more than one possible derivation for a given sequence of terminals. In this context different derivations correspond to different possible RNA secondary structures combined with different possible coding regions, in this sense many derivations of the same sequence are expected. However ambiguity becomes a concern if there exists more than one derivation of a sequence which has the same secondary structure and coding regions. The optimal derivation or parse is defined to be the one which generates the sequence with the highest probability. The CYK algorithm finds the optimal derivation but this is only guaranteed to find the optimal structure if there is a one to one correspondence between derivations and secondary and gene structures. In all my previous work it has been assumed that the structure obtained from the optimal derivation is optimal itself. With an ambiguous grammar this is not a valid assumption.

If a grammar is structurally ambiguous then in order to find the probability of a particular secondary structure it is necessary to sum over the probabilities of all derivations consistent with that structure.

Determining ambiguity is an undecidable problem. Although this sounds like bad news it does not mean that it cannot be done. It means that an algorithm cannot be written to do it systematically. Ambiguity of the combined grammar can be established empirically by searching for a sequence for which the same secondary structure and coding regions can be produced in more than one way. Finding such a sequence confirms that multiple parses may produce the same secondary and coding structure hence confirming the ambiguity of the grammar. The combined grammar G^* defined in figure 7 is ambiguous as demonstrated by the following example.

Take the string 'aa' and suppose that they are both non-coding nucleotides which form part of a loop. Then there are two possible derivations of the string:

1. $S_{be} \rightarrow S_{00}W^1_{0e} \rightarrow aa$
2. $S_{be} \rightarrow W^1_{00}S_{0e} \rightarrow aa$

The impact structural ambiguity makes on practical structure prediction can be tested by a reordering experiment as introduced by [3]. They propose sampling from distribution of parses to find N further suboptimal parses. The $N + 1$ parses are then ranked by their probabilities (where the optimal one originally found is ranked 1 by

definition). For each of the $N + 1$ parses the secondary and coding structure is extracted. Then probabilities of each of these (combined) secondary and coding structures are computed by summing over all parses for that structure. These probabilities are ranked and compared with the original parse tree rankings.

If ambiguity does not matter in practise, then very few differences in rankings should be seen between that of the parses and the structure probabilities. Impact varies according to the grammar in question, [3] tests a relatively simple grammar incorporating base pair stacking parameters (similar to the grammar I use) and a more optimal structure is found for 98% of the sequences when sampling is performed deep into the posterior distribution of parse trees (taking $N = 1000$).

This result raises concern and suggests that the impact of structural ambiguity with the combined grammar on structure prediction is likely to be significant. It is possible to investigate the impact of ambiguity of the combined grammar using the methods presented in [3]. These methods are highly computational and they are not guaranteed to find the optimal structure. For this reason, it is recommended the problem is avoided in my case this can be done by ensuring the original grammar to be combined is not structurally ambiguous. The process by which the original grammar is combined with a HMM does not introduce any ambiguities; all the new non-terminals and rules introduced correspond to different coding states and hence a different structure, so if the original grammar for modelling the secondary structure is not ambiguous then the combined grammar will not be ambiguous.

8 Results

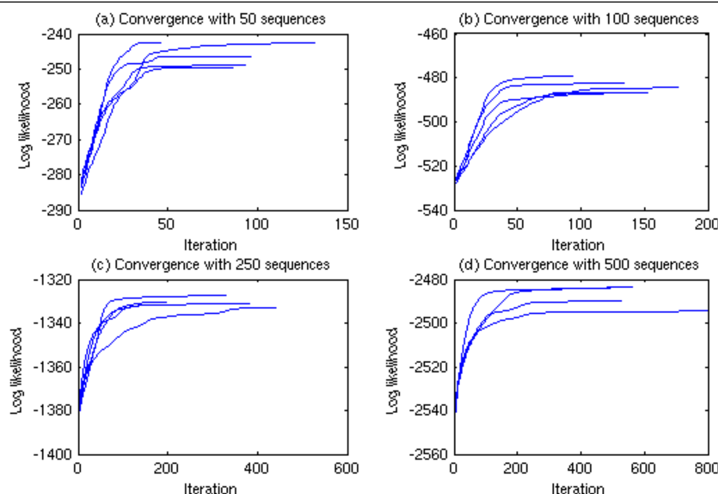
8.1 Parameter Estimation and Sensitivity

Implementing the appropriate algorithms for the combined SCFG appears complicated due to the large number of production rules and non-terminals however using the scfg module as a platform to work from, the algorithms I have written can be used on any ε-free SCFG. Implementation simply requires specifying the combined grammar in the required form such that it can be read in by the scfg module. The grammar files are in Appendix A.4.

To investigate how accurately inside-outside training estimates the parameters for the combined grammar it is necessary to first simulate from a grammar (with arbitrarily chosen parameters) and then run inside-outside training on simulated sequences. It is not clear how the parameters of the grammar from which to simulate should be chosen. There are no natural choices, so they are allocated randomly. Each production rule is assigned a uniform number from the interval $(0, 1)$ and then these weights are normalised to impose well defined probability distributions over rules from each non-terminal. The result is the random combined grammar specified as a file in Appendix A.4.3. This is the combined grammar from which training sequences are simulated.

The Inside-Outside training is performed on the grammar with 50, 100, 250 and 500 training sequences. Note that there are 2 copies of the terminal production rules from the start non-terminal in the combined grammar. Both copies must be retained during training since each copy corresponds to the emission of the terminal from a different state in the HMM. To check that the loglikelihood is increasing after each iteration and converging to a local maximum I plotted the loglikelihood. Figure 10 shows the loglikelihood plots demonstrating convergence for varying number of training sequences. It confirms that the resulting estimates for the parameters of the SCFG are local maxima and highlights that the number of iterations required for convergence increases with the number of sequences used for training. Computationally this means increasing the number of sequences used for training increases the time taken significantly. In particular performing parameter estimation with 500 sequences took a week. This is not ideal and for future use it will need to be reprogrammed more efficiently.

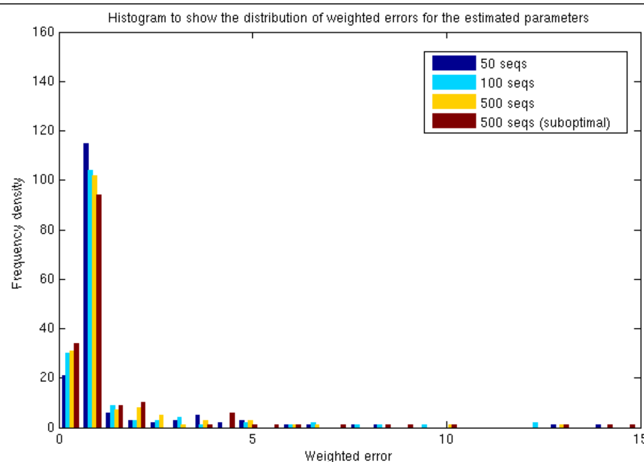
Figure 10 A set of plots to show the convergence of the loglikelihood in the parameter training algorithms with a varying number of sequences.



The 149 estimated parameters are difficult to compare directly with each other, for this reason the actual estimates have been omitted although figure 11 plots the absolute values of the weighted errors in a histogram in different colours according to the number of sequences used for training. The errors have been weighted by the reciprocal of the value of the true parameter since they range from very small values to much larger ones. An error of 0.05 may not be considered large if the true parameter is 0.9 whereas if the true parameter is 0.000001 an error of 0.05 is more significant. The results are quite surprising. The estimates based on the training with 500 sequences do not appear to be significantly more accurate than those based on only 50 sequences. It is noticeable that the first bar of the histogram is larger for the training based on 500 sequences where as the second bar is larger for 50 sequences. This highlights that there are a greater proportion of smaller errors for the grammar estimates from

500 sequences as would be expected.

Figure 11 A histogram to display the absolute value of the errors of the parameter estimates. Different coloured bars correspond to the errors of the estimates obtained when the grammars are trained on a varying number of sequences. The histogram has been truncated to exclude outlying values (corresponding to a very large errors).



I proceed to evaluate how well the estimated grammars perform relative to the true annotation of the sequences. The true annotations can be extracted by retaining the derivation of the sequence in the same form as the trace back of the CYK algorithm. Then the functions extracting secondary and gene structure structure can be applied to give the true annotations.

In the context of gene finding and secondary structure prediction it is more widespread to consider the two types of error separately. The model can predict a coding nucleotide or base pairing where there is not one, or it can fail to identify a coding nucleotide or base pairing where there is one, these type of errors are called false positives and negatives respectively. The (1-false positive rate) is called **specificity** and the true positive rate (1-false negative rate) is called **sensitivity**. Sensitivity gives a measure of how well the model predicts gene/base pairing which are present while specificity gives a measure of how well the model predicts the non-coding nucleotides/loops. There is a trade off between sensitivity and specificity which is easily demonstrated by the following extreme example. Suppose that a model predicts that every nucleotide is coding. Then the sensitivity will be 1 since it cannot fail to predict each gene, but the specificity will be 0 since it never predicts a non-coding region.

The true annotations of test sequences are known, so the false positive and false negative rates can be used to evaluate how well a model performs. The sensitivity rate for the secondary structure is calculated in a different way to the gene finding sensitivity since it is necessary to ensure that annotations are only considered to be the same if and only if they contain exactly the same base pairs. The base pairings in the sequence are extracted from the true structure annotation and from the predicted structure annotation (see Appendix A.7 for the method and code). To compute structure sensitivity take the number of base pairings from the true annotation which are also present in the predicted annotation and divide by the total number of base pairs in the true annotation. Computation of specificity remains unchanged where a negative corresponds to a loop nucleotide. The sensitivity and specificity rates of the estimated grammar annotations and the true grammar annotations are displayed in table 9.

These results are disappointing and I worried that there was a problem with my algorithms. To investigate this I ran more tests and looked at how the true grammar CYK prediction compared to the true annotations of the sequences. The specificity and sensitivity for the true combined grammar annotations are also displayed in table 9 and these although better, are not good either, so it is not surprising that the estimated grammars do not appear to be performing well. It suggests that the most likely annotation is not a good estimate of the true gene and secondary structure. This may be because there many suboptimal annotations which make similar (although smaller) contributions to the likelihood. This would seem reasonable given that there are so many possible annotations for the same sequence.

From these results it is difficult to distinguish whether the grammar obtained from 500 training sequences performs better than models trained on fewer sequences as one would expect.

GRAMMAR FOR CYK ANNOTATION	ANNOTATION TYPE	SPECIFICITY RATE	SENSITIVITY RATE
Estimated grammar based on 50 training sequences	Gene	0.5726	0.4673
Estimated grammar based on 100 training sequences	Gene	0.5240	0.4741
Estimated grammar based on 250 training sequences	Gene	0.4656	0.5379
Estimated grammar based on 500 training sequences	Gene	0.5532	0.5156
True combined grammar	Gene	0.7116	0.5850
Estimated grammar based on 50 training sequences	Structure	0.5830	0.3979
Estimated grammar based on 100 training sequences	Structure	0.5673	0.4774
Estimated grammar based on 250 training sequences	Structure	0.5259	0.5716
Estimated grammar based on 500 training sequences	Structure	0.5374	0.4029
True combined grammar	Structure	0.4362	0.5591

Table 8: A table to evaluate how well the estimated random combined grammar annotations of and the true random combined grammar annotations model perform relative to the true annotation of the 1000 test sequences using specificity and sensitivity rates.

8.1.1 Technical Details

There are two copies of each terminal production from the start non-terminal. My results show that the sum of the estimated probabilities assigned identical copies of the same rule are determined by the proportion of single nucleotide sequences (of a particular type) as expected, however the way in which this probability is split between the two copies of the rule is entirely determined by the values of the initial parameters. This is not desirable and can be avoided by allowing the following replacement production rules (which I eliminated previously!).

$$S_{\tilde{b}\tilde{e}} \rightarrow S_{0\tilde{e}}|S_{1\tilde{e}}$$

In addition, having developed a method for combining a HMM with silent states I realise that eliminating replacement rules introduces more parameters than necessary. The following example demonstrates why this is the case. Let

1. $S = A$
2. $T = \{a, b\}$
3. $V = \{A, B\}$
4. $P = \{A \rightarrow B,\, B \rightarrow a|b\}$

Then eliminating the production rule $A \rightarrow B$ is done by introducing the rules $A \rightarrow a|b$. There are two parameters to assign in the resulting grammar rather than the single parameter of the original grammar. Eliminating replacement rules in this way yields an equivalent grammar only if it is not stochastic. This is not the case for my work and I overlooked this initially.

The combined grammar I implemented is not strictly the combined grammar I should have used for this reason. I introduced more parameters than necessary. For future testing, developing an implementation that can handle replacement productions (which do not enter self-referencing loops) will be required and may improve results.

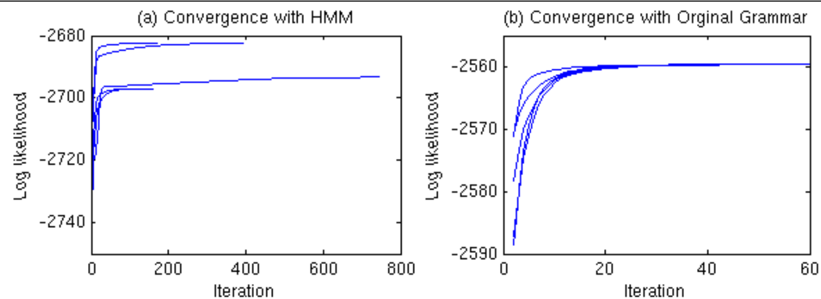
I proceed to investigate how the two independent models perform relative my combined grammar.

8.2 Comparing the combined annotation with the HMM and original SCFG annotations

In this section I compare how the joint annotation of secondary and gene structure via the CYK algorithm with the combined model differs from the two separate model annotations when the assumption of conditional independence and hence use of the latter is invalid. To estimate parameters for the two independent models, a set of 500 sequences are simulated from the combined grammar and then inside-outside training used. It is important that

the set of sequences used to train the combined grammar is used to train both the HMM and the original SCFG because this is exactly what the bioinformatician would do with a set of training DNA sequences. Upon doing this, training for the original grammar converged in a relatively small number, typically less than 50 iterations (relative to the average of over 500 for the combined grammar!) whereas training for the HMM on average took 322 iterations. The convergence of the training for the HMM is very slow considering that only a small number of parameters are being estimated, it may be an indication that a HMM is not appropriate to model the data. The convergence plots are displayed in figure 12. The training is performed several times from different random starting points and the parameters for the estimated grammar are those corresponding to the training which yields the largest loglikelihood of the training sequences.

Figure 12 Graphs to show the convergence of the loglikelihood when the HMM and the original grammar are trained.



To compare how the estimated independent models perform relative to the true grammar, a new set of test sequences are simulated from the combined grammar. The estimated original grammar is used to annotate the test sequences with their most likely secondary structure and the estimated HMM is used to annotate the test sequences with their most likely gene structure under the independence assumption. These annotations are compared with the true annotations retained from the derivation of the sequences. The sensitivity and specificity results are presented in table 9. They are computed using the code in Appendix A.7

GRAMMAR FOR ANNOTATION	ANNOTATION TYPE	SPECIFICITY RATE	SENSITIVITY RATE
True combined grammar	Gene	0.5591	0.5850
Estimated combined grammar	Gene	0.5532	0.5156
Estimated Hidden Markov Model	Gene	0.6058	0.375
True combined grammar	Secondary Structure	0.4362	0.7116
Estimated combined grammar	Secondary Structure	0.5375	0.4029
Estimated original grammar	Secondary Structure	0.4251	0.6916

Table 9: A table displaying specificity and sensitivity rates to evaluate how well the estimated random combined grammar annotation, the true random combined grammar annotation, and the independent model annotation performs relative to the true annotation of the 1000 test sequences.

These results are also poor and it is difficult to draw any conclusions from them. It is apparent that neither of the models perform well relative to the true annotations of the sequences! As discussed in the previous section, this may be because the CYK most likely annotation gives a poor estimate of the true annotation. This is a problem which I did not anticipate. I expected the high dimension of the parameter space to make it difficult to obtain sufficiently accurate parameter estimates but the poor CYK results makes it difficult to determine to what extent this is also a problem.

9 Conclusion

The primary objective of this project was to develop a method to combine an SCFG with an HMM, such that a combined model can be used model the joint density of a RNA secondary structure and gene structure of any given sequence. I have discovered over the course of the project that this is always possible to do by introducing new non-terminals and subsequent productions rules according to the algorithms presented in section 6. In addition, the project raised and answered some interesting questions regarding independence.

Having developed a method for combining a general HMM with a general SCFG (with the restrictions discussed previously), I tested how the combined grammar predicted annotation compared with the annotation obtained via the use of the two separate models when the assumption of conditional independence is invalid. My results are disappointing show that neither of the models perform well relative to the true annotations. All of my results are calculated having used the CYK most likely parse to annotate a sequence with its predicted structure. The results show that this method of prediction does not work well even when the true parameters are used. The reason for this, may in part be due to the true parameters of the combined model. I set them randomly because there was no obvious way to choose them, but in hindsight this was not a good way to assign them because resulting combined grammar did not contain strong signals, so many possible annotations will occur with similar probabilities. Under these circumstances one would not expect the CYK annotation to give good results.

My results may show significant improvement if I could set the parameters to encode strong signals. Training on the 'random' grammar is likely to detect weak or possibly even strong signals where they do not necessarily exist. If training is performed at least twice the resulting estimated grammar may contain stronger signals. So if this estimated grammar is then used as the true combined grammar in subsequent training and testing I may see an improvement in sensitivity and specificity results.

An alternative approach would be to try combining more realistic grammars that could be trained on DNA sequence data with stronger signals. This would involve implementing algorithms which can handle non-self referencing replacement productions, to enable gene finding HMMs with silent states to be combined with SCFGs for RNA secondary structure prediction such as the Knudsen-Hein grammar introduced in [4]. Developing such an implementation will also allow the my existing combined grammar to be specified more efficiently as discussed in section 8.1.1.

I plan to implement both of the above in the future, but if CYK annotation continues to perform badly, alternative methods for estimating the true annotation must be considered and investigated if the combined model is to be a useful tool. Posterior decoding is one such option which allows inference of the structure to be made via imposition of certain restrictions. For example one could ask questions such as 'what is the probability a particular nucleotide is in a coding region and base pairing given the rest of the sequence'. It is possible to construct a complete annotation in this way although it may not be very likely. It would be interesting to see how posterior decoding results compare to my existing results. I would expect inference about specific features of the structure to yield better results because it is performed by summing over all possible annotations possessing that feature; not just the most likely annotation.

The motivation for the project stems from questioning the assumption frequently made by bioinformaticians; that the gene and RNA secondary structure of any given sequence is independent. My investigation and construction of a combined model highlighted several interesting issues; A SCFG and an HMM should only be used to predict secondary structure and coding regions if secondary structure and gene structure can assumed to be independent given a sequence. To determine whether the two separate models or the joint model is appropriate, ideally a data dependent Fisher's exact test should be used, although this is only possible if training data for which the gene and secondary structure is already known is available. Investigating whether the combined model or the two separate models fit the data best is not possible using nested methods since it is not possible to assign parameters to the combined grammar in a convenient way such that results in the same predictions had the individual models been used. Consequently, even if the secondary and gene structure are independent given a sequence, use of the combined model may introduce dependencies which are not a feature of the data, but are a feature of the model. This is not desirable and I hope that future investigation into non-tested tests succeed so that it is possible to establish which model should be used when no annotated training data is available.

Acknowledgements

Many thanks to Jotun Hein, Rune Lyngsø and Thomas Mailund for their guidance and interesting discussions. They have been invaluable and much appreciated. Thanks also to Karen Lees for her patience to help me understand how Object Orientated Programming works!

References

[1] R.Durbin, S.Eddy, A.Krogh & G.Mitchison, *Biological Sequence Analysis* 1998 : Cambridge University Press

[2] J.E.Hopcroft, R.Motwani & J.D.Ullman, *Introduction to Automata Theory, Languages and computation* 2001: Addison Wesley

[3] R.D.Dowell & S.Eddy, *Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction* 2004 : BMC Bioinformatics Vol 5:71

[4] B.Knudsen & J.Hein, *RNA secondary structure prediction using stochastic context-free grammars and evolutionary history* 1999: Bioinformatics, Oxford University Press

[5] J.S.Pederson & J.Hein,*Gene finding with a hidden Markov model of genome structure and evolution* 2003: Bioinformatics, Oxford University Press

[6] J.S Pedersen, I.M.Meyer, R.Forsberg, P.Simmonds & J.Hein *A comparative method for finding and folding RNA secondary structures within protein-coding regions* 2004: Nucleic Acids Research, Vol. 32, No.16 4925-4936

[7] J.A.Sanchez, J.M.Benedi, F.Casacuberta *Comparison between the Inside-Outside Algorithm and the Viterbi Algorithm for Stochastic Context-Free Grammars* 1996: Springer Verlag Lecture Notes in AI. Vol 1121 p.50-59

[8] A.Jagota, R.B.Lyngsø, C.N.S Pedersen *Comparing a Hidden Markov Model and a Stochastic Context Free Grammar* 2001: Springer-Verlag Lecture Notes in Computer Science. Vol 2149 p.69

[9] R.Hughey, A.Krogh *Hidden Markov models for sequence analysis: Extension and analysis of the basic method* 1996: Computer Applications in the Biosciences. Vol 12(2), p.95-107

[10] T.A Sudkamp, *Language and Machines* 1988: Addison Wesley Publishing.

[11] O.Kreylos *Chomsky Normal Form* 2001: Lecture 9 online notes http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/IF/09/chomsky.pdf.

[12] A.Gelman, J.B.Carlin, H.S.Stern, D B.Rubin *Bayesian Data Analysis* 2000: (reprint) Chapman and Hall/CRC

A The Appendices

A.1 Technical SCFG algorithms

The following algorithms are those discussed is section 4.3. They are written by R. Lyngsø

A.1.1 Removing epsilon productions from a SCFG which cannot generate the null string

The removal of ε-productions is formalised by two algorithms. The first removes a single epsilon production and the second removes all epsilon productions from the grammar (provided it is epsilon free).

Algorithm 4 Eliminate $U \to \varepsilon$ from P

/ Remove $U \to \varepsilon$ from P */*
$P = P \setminus \{U \to \varepsilon\}$
/ Push $U \to \varepsilon$ to all productions where U occurs once */*
for every $V \to xUy \in P$, U does not occur in xy **do**
  $P = P \cup \{V \to xy\}$
  $\Pr(V \to xy) = \Pr(U \to \varepsilon)\Pr(V \to xUy)$
  $\Pr(V \to xUy) = (1 - \Pr(U \to \varepsilon))\Pr(V \to xUy)$
/ Push $U \to \varepsilon$ to all productions where U occurs twice */*
for every $V \to UU \in P$ **do**
  $P = P \cup \{V \to U, V \to U, V \to\}$
  $\Pr(V \to U = \Pr(V \to U) = \Pr(U \to \varepsilon)(1 - \Pr(U \to \varepsilon))\Pr(V \to UU))$
  $\Pr(V \to xyz) = \Pr(U \to \varepsilon)^2 \Pr(V \to UU)$
  $\Pr(V \to UU) = (1 - \Pr(U \to \varepsilon))^2 \Pr(V \to UU)$
/ Re-normalise production probabilities for productions for U */*
for every $U \to x \in P$ **do**
  $\Pr(U \to x) = \Pr(U \to x)/(1 - \Pr(U \to \varepsilon))$

Algorithm 5 Eliminate all ε productions for N

if S occurs on right hand side of any productions in P **then**
  / Introduce new start symbol not occurring on right hand side of production */*
  $N = N \cup \{S'\}$
  $P = P \cup \{S' \to S\}$
  $\Pr(S' \to S) = 1$
  $G = (\Sigma, N, P, S')$
for $i = 1$ **to** $|N|$ **do**
  for every old[1] ε production $U \to \varepsilon \in P, U \neq S'$ **do**
    Eliminate $U \to \varepsilon$ using algorithm 4
if $\exists U \to \varepsilon \in P$ with $U \neq S'$ **then**
  There is a cycle along which ε productions can be pushed

[1] We only push ε productions predating this iteration of the outer loop – otherwise the inner loop may become infinite

A.1.2 Removing replacement productions from a SCFG

The removal of replacement productions is done in a similar way via two algorithms. Algorithm 6 removes a single replacement production and algorithm 7 eliminates all replacement productions in a SCFG.

Algorithm 6 Eliminate $U \to V$ from P

/ Remove $U \to V$ from P */*
$P = P \setminus \{U \to V\}$
/ Push $U \to V$ to all productions where U occurs once */*
for every $W \to xUy \in P$, U does not occur in xy **do**
  $P = P \cup \{W \to xVy\}$
  $\Pr(W \to xVy) = \Pr(U \to V)\Pr(W \to xUy)$
  $\Pr(W \to xUy) = (1 - \Pr(U \to V))\Pr(V \to xUy)$
/ Push $U \to V$ to all productions where U occurs twice */*
for every $W \to UU \in P$ **do**
  $P = P \cup \{W \to UV, W \to VU, W \to VV\}$
  $\Pr(W \to UV) = \Pr(W \to VU) = \Pr(U \to V)(1 - \Pr(U \to V))\Pr(W \to UU)$
  $\Pr(W \to VV) = \Pr(U \to V)^2 \Pr(W \to UU)$
  $\Pr(W \to UU) = (1 - \Pr(U \to V))^2 \Pr(W \to UU)$
/ Re-normalise production probabilities for productions for U */*
for every $U \to x \in P$ **do**
  $\Pr(U \to x) = \Pr(U \to x)/(1 - \Pr(U \to V))$

Algorithm 7 Eliminate all replacement productions from G

if S occurs on right hand side of any productions in P **then**
  / Introduce new start symbol not occurring on right hand side of production */*
  $N = N \cup \{S'\}$
  $P = P \cup \{S' \to S\}$
  $\Pr(S' \to S) = 1$
  $G = (\Sigma, N, P, S')$
for $i = 1$ **to** $|N|$ **do**
  for every old replacement production $U \to V \in P, U \neq S'$ **do**
    Eliminate $U \to V$ using algorithm 6
if $\exists U \to V \in P$ with $U \neq S'$ **then**
  There is a cycle along which replacement productions can be pushed

A.1.3 Collapsing Identical Productions for use of the CYK Algorithm

The removal of replacement productions and epsilon productions may introduce multiple identities copies of the same rule. The way to collapse them is dependent on the algorithm to be subsequently used. Algorithm 8 is used if the CYK is to be used and algorithm 9 is used if the Inside-Outside algorithms are to be used.

Algorithm 8 CYK collapse of identical productions

for every type t of production in P **do**
  $\Pr(t) = 0$
  for every $p \in P$ of type t **do**
    $\Pr(t) = \max\{\Pr(t), \Pr(p)\}$

Algorithm 9 Inside(-outside) collapse of identical productions

for every type t of production in P **do**
  $\Pr(t) = 0$
  for every $p \in P$ of type t **do**
    $\Pr(t) = \Pr(t) + \Pr(p)$

A.2 The SCFG Module

The SCFG module is written by Rune Lyngsø. The code is several pages long, and as not been included here but it can be found at http://www.stats.ox.ac.uk/ lyngsoe/scfg.py

A.3 My Extension to the SCFG module

```
################################################################################
################################################################################
#                     Jo's extension of the SCFG class                        #
################################################################################
################################################################################

# import required packages including the scfg module
from scfg import SCFG
from numarray import array, zeros, Float64, resize
from random import uniform
from copy import deepcopy
from math import log
###############################################################################
class MySCFG(SCFG):
###############################################################################
# a function to extract the transition probabilities from the grammar and put
# them in an array.  They are to be used in the inside and outside algorithms
  def get_transition(self):
    nv=self.count_variables()
    t=zeros((nv,nv,nv), Float64)
    for np in self.nonterminal_productions():
      i = np.get_variable().get_index()
      j = np.get_left().get_index()
      k = np.get_right().get_index()
      t[i][j][k] = np.get_probability()
    return t
###############################################################################
# A function to extract emission probabilities from the grammar & put them
# in an array. They will also be used in the inside & outside algorithms
  def get_emission(self, al=0):
   # 0=inside ,1=cyk
    nv=self.count_variables()
    nt=self.count_terminal()
    e = zeros ((nv,nt),Float64)
    for np in self.terminal_productions():
      i = np.get_variable().get_index()
      j = np.get_symbol()
      if (al == 0):                   # handle multiple copies of identical rules
        e[i][j] += np.get_probability()# for inside algorithm required to sum
      elif (al == 1):                 # for cyk algorithm required to take max
        e[i][j] = max(e[i][j], np.get_probability())
    return e
###############################################################################
# This function gets the emission probabilities for a particular sequence
# according to the characters in the sequence it is necessary to use if the
# probability of observing a sequence is to be computed from the oustide
# quantities.  This is to be done as a check.
  def get_observed_emission(self,s):
```

```
    alp=self.get_alphabet()
    ns=len(s)
    nv=self.count_variables()
    nt=self.count_terminal()
    e= self.get_emission()
    x = range(ns)
    for i in range(ns):
      x[i]=alp.index(s[i])
    obs_e=zeros((nv,ns),Float64)
    for i in range(ns):
      for j in range(nt):
        if x[i]==j:
          for k in range(nv):
            obs_e[k][i]= e[k][j]
    return obs_e
##############################################################################
# A function written to perform either the CYK or the Inside algorithm for a
# grammar inputted using the SCFG module.  The function is a method with
# additional paramters (i) the sequence (ii) the algorithm required.
  def cyk_inside(self,s,al=0):
    # 0=inside ,1=cyk
    # M is the array to hold the required quantities
    # require the dimensions of M before creating it
    nv=self.count_variables()   # total number of non-terminals
    nt=self.count_terminal()    # total number of terminals
    ns=len(s)                   # the length of the sequence
    alp=self.get_alphabet()     # alp is the terminal alphabet of the grammar
    # enumerate the terminals and rewrite the sequence accordingly. Call it x
    x = range(ns)
    for i in range(ns):
      x[i]= alp.index(s[i])
    # create transition & emission arrays from functions defined in scgf class
    t=self.get_transition()
    e=self.get_emission()
    # create M
    M=zeros((ns,ns,nv),Float64)
    if (al==1):
      tb=[[[None for i in range(nv)] for i in range(ns)] for i in range(ns)]
    # initialise M:
    for i in range(ns):
      k=x[i]
      for j in range(nv):
        M[i][i][j]= e[j][k]
        if (al==1):
          tb[i][i][j]= (j,s[i])
    # Do a recursion to define all the entries of M
    for i in range((ns-1),-1,-1):
      for j in range(i+1,ns):
        for p in self.nonterminal_productions():
          v=p.get_variable().get_index()
          y=p.get_left().get_index()
          z=p.get_right().get_index()
          for k in range(i,j):
            if (al==0):
              M[i][j][v]+=M[i][k][y]*M[k+1][j][z]*p.get_probability()
            else:
```

```
            temp=M[i][k][y]*M[k+1][j][z]*p.get_probability()
            if temp > M[i][j][v]:
              M[i][j][v]=temp
              tb[i][j][v]=(v, tb[i][k][y], tb[k+1][j][z])
    if (al==0):
      return M
    else:
      return tb[0][ns-1][0]
###############################################################################
# A function to perform the outside algorithm.  It returns matrix of outside
# (beta) quantities which are defined in the text.
  def outside(self,s):
    a=self.cyk_inside(s,0)
    nv=self.count_variables()    # total number of non-terminals
    nt=self.count_terminal()     # total number of terminals
    ns=len(s)                    # the length of the sequence
    alp=self.get_alphabet()      # alp is the terminal alphabet
    x = range(ns)
    for i in range(ns):
      x[i]=alp.index(s[i])
    t=self.get_transition()
    e=self.get_emission()
    obs_e= self.get_observed_emission(s)
    # calculate the outside probabilities hold them in matrix called B.
    B=zeros((ns,ns,nv), Float64)
    # initialise
    B[0][ns-1][0]=1
    for r in range(1,nv):
      B[0][(ns-1)][r]= 0
    # recursion
    for i in range(0,ns):
      for j in range((ns-1),(i-1),-1):
        if i == 0 and j == (ns-1):
          # Skip already initialised entries
          continue
        for v in range(nv):
          for p in self.nonterminal_productions():
            if p.get_right().get_index()==v:
              y=p.get_variable().get_index()
              z=p.get_left().get_index()
              for k in range(i):
                B[i][j][v]+=((B[k][j][y])*(a[k][i-1][z])*(t[y][z][v]))
              if t[y][v][z]>0:
                for w in range(j+1,ns):
                  B[i][j][v]+=(a[j+1][w][z])*(B[i][w][y])*(t[y][v][z])
            if p.get_left().get_index()==v:
              y=p.get_variable().get_index()
              z=p.get_right().get_index()
              if t[y][z][v]==0:
                for w in range(j+1,ns):
                  B[i][j][v]+=(a[j+1][w][z])*(B[i][w][y])*(t[y][v][z])
    return B
###############################################################################
# A function written to set the paramters of the grammar randomly when there
# is no natural choice.
  def set_random_parameters(self):
```

```
    for p in self.productions():
      p.set_probability(uniform(0,1))
    self.normalise()
    return self
################################################################################
# This function I use if the initial grammar from which subsequent iterations
# are made is to be the uniform distribution. Not used in practise since
# required to start from several different starting values which are set by
# the function above.
  def set_initial_grammar(self):
    for p in self.productions():
      p.set_probability(1)
    self.normalise()
    return self
################################################################################
# This function returns nsim simulated sequences in vector
  def training_sequences(self,nsim):
    tr_seq = range(nsim)
    for i in range(nsim):
      tr_seq[i] = self.sample()
    return tr_seq
################################################################################
# This is an implementation of the inside-outside training algorithm Note
# that the program doesn't follow Durbin's specification exactly because I
# have used a couple of tricks to increase efficiency.
  def training(self,training_sequences,maxit=10):
    nsim=len(training_sequences)
    IG= deepcopy(self)
    IG=IG.set_random_parameters()
    nv = self.count_variables()      # total number of non-terminals
    nt = self.count_terminal()       # total number of terminals
    alp = self.get_alphabet()        # alp is the terminal alphabet of the grammar
    # x is a vector of sequences simulated from the original grammar G
    # start iterations-counter = q:
    new_loglike=0.0
    q=0   # always do one iteration
    while q <= 1 or new_loglike-current_loglike >0.0005 and q<maxit:
      current_loglike=new_loglike
      new_loglike=0.0
      print 'starting iteration', (q+1)
      tc= zeros((nv),Float64)
      tg={}
      td={}
      tg= {}
      for np in IG.terminal_productions():
        tg[np] = 0
      for np in IG.nonterminal_productions():
        td[np] = 0
      for f in range(nsim):         # for each of the simulated sequences
        s=training_sequences[f]
        l=len(s)
        x = range(l)
        for j in range(l):
          x[j]=alp.index(s[j])
        a=IG.cyk_inside(s) # run inside algorithm
        b=IG.outside(s)    # run outside algorithm (with initial grammar)
```

```
  prob=a[0][l-1][0]
  # now define a vector (dim nv) conisting of the expected number of times
  # each non-terminal is used during the production of the sequence.
  c=zeros((nv), Float64)
  for r in range(nv):
    for y in range(1,l+1):
      for j in range(y,l+1):
        c[r]+=a[y-1][j-1][r]*b[y-1][j-1][r]
    c[r]=c[r]/(prob)
    tc[r]+=c[r]        # add to the total count
  d={}
  for np in IG.nonterminal_productions():
    d[np] = 0
  for np in IG.nonterminal_productions():
    r = np.get_variable().get_index()
    z = np.get_left().get_index()
    y = np.get_right().get_index()
    for w in range(1,l):
      for j in range(w+1,l+1):
        for k in range(w,j):
          d[np]+= b[w-1][j-1][r]*a[w-1][k-1][z]*a[k][j-1][y]*(np.get_probability()
    d[np]= d[np]/(prob)
    td[np]+=d[np]
  g = {}
  for np in IG.terminal_productions():
    g[np] = 0
  for np in IG.terminal_productions():
    v = np.get_variable().get_index()
    y = np.get_symbol()
    for j in range(l):
      if x[j]==y:
          g[np]+=b[j][j][v]*(np.get_probability())
    g[np]=g[np]/(prob)
    tg[np]+= g[np]
############################## DEBUG TEST ####################################
#for v in IG.variables():                                                     #
#  test=0.0                                                                   #
#  for np in IG.nonterminal_productions():                                    #
#  if np.get_variable()==v:                                                   #
#     r = np.get_variable().get_index()                                       #
#     z = np.get_left().get_index()                                           #
#     y = np.get_right().get_index()                                          #
#     test+=td[np]                                                            #
#  for np in IG.terminal_productions():                                       #
#     if np.get_variable()==v:                                                #
#     test+=tg[np]                                                            #
#  print '<', v.get_index(),'>  ',test,'as computed by total count:',         #
#                                                   tc[(v.get_index())]       #
#                                                                             #
#for v in IG.variables():                                                     #
#  if v.get_index()==0:                                                       #
#    for p in v.terminal_productions():                                       #
#        print '<',p.get_variable().get_index(),'>  ',                        #
#                                  self.alphabet[p.get_symbol()], p.get_probability()#
###############################################################################
# now outside of the loop going through each sequence simulated (but
```

```
        # inside iteration loop), use the total expected counts to update
        # parameter estimates. Note that any initial value of zero will remain
        # zero. This means the grammar keeps its structure of permitted rules.
        # This step automatically updates the grammar
        for np in IG.nonterminal_productions():
          r = np.get_variable().get_index()
          y = np.get_left().get_index()
          z = np.get_right().get_index()
          if np.get_probability()==0:
            pass
          elif np.get_probability()==1:
            pass
          else:
            np.set_probability(td[np]/(tc[r]))
        for np in IG.terminal_productions():
          v = np.get_variable().get_index()
          y = np.get_symbol()
          if np.get_probability()==0:
            pass
          elif np.get_probability()==1:
            pass
          else:
            np.set_probability((tg[np]/(tc[v])))
        # calculate new loglikelihood
        for i in range(nsim):
          a=IG.cyk_inside(training_sequences[i])
          l=len(training_sequences[i])
          new_loglike+=log(a[0][l-1][0])
        q+=1    # update iteration counter
        ### output updated probabilities ###
        print 'change in loglikelihood after ',q,'iterations'
        print new_loglike-current_loglike
        print 'new loglikelihood:',new_loglike
        print '----------------------------------------------------------------'
      print 'convergence after',q,'iterations yields the following grammar:'
      IG.output_grammar()
      return IG
###############################################################################
###############################################################################
# The following functions are particular to the grammars we are working with.#
# they are not defined in the class for this reason. The structure & coding #
# annotations need to be extracted and this is done according to the way the #
# grammars are specified.                                                     #
###############################################################################
#                           Combined annotations                              #
###############################################################################
# output list of 0's and 1's to represent the coding state of terminals. It is
# a function which acts on a CYK traceback according to the combined grammar
def parse_coding(bt,G):
  z=G.count_productions()
  output=zeros((z),Float64)
  h=0
  for p in G.productions():
    output[h]= p.get_probability()
    h+=1
  if (len(bt)==3):
```

```
    return parse_coding(bt[1],G) + parse_coding(bt[2],G)
  else:
    if (bt[0] in [1,2,5,7,8,11,13,14,17,19,20,23,25,26,29]):
      return [0]
    elif (bt[0])==0:
      if (bt[1]=='a'):
        if (output[0]>= output[1]):
          return [0]
        else:
          return [1]
      else:
        if (output[2]>= output[3]):
          return [0]
        else:
          return [1]
    else:
      return [1]
################################################################################
# This function extracts the seecondary structure from the combined grammar
# traceback of the CYK. It outputs parenthesis to represent base pairing &
# dots for spacers.
def parse_structure(bt):
  if (len(bt)==3):
    jo=bt[1]
    phil=bt[2]
    if (len(jo)==2):
      if (len(phil)==2):
        if len(jo[1])==1 and len(phil[1])==1:
          if jo[0] in [7,8,9,10,11,12,19,20,21,22,23,24] and phil[0] in
                                          [7,8,9,10,11,12,19,20,21,22,23,24]:
            return ['(',')']
          else:
            return ['.','.']
        else:
          print 'error␣1'
      else:          # nb len(phil) must == 3.
        if phil[0] in [25,26,27,28,29,30,13,14,15,16,17,18]:
          return ['('] + parse_structure(phil[1])+[')']
        else:
          if jo[0] in [7,8,9,10,11,12,19,20,21,22,23,24]:
            if phil[0] in [1,2,3,4,5,6]:
              return ['.']+ parse_structure(phil)

    else: # nb len(jo)==3 and since no branching, phil must be of length 2
      if jo[0] in [1,2,3,4,5,6]:
        return parse_structure(jo) + ['.']
      else:
        print 'phil' , phil
        print 'jo' , jo , 'error␣2'
  else:
    if bt[0] in [0,1,2,3,4,5,6]:
      if len(bt[1])==1:
        return ['.']
################################################################################
#                          independent annotatations                           #
################################################################################
```

```
# output parentheses for base pairs and dots for spacers (uncombined structure
# grammar)
def parse_simple_structure(bt):
  if (len(bt)==3):
    if (len(bt[1])==2):
      if (len(bt[2])==2):
        return ['(',')']
      elif (len(bt[2])==3):
        temp= bt[2]
        if temp[0] in [1,3]:
          return ['('] + parse_simple_structure(temp[1])+[')']
        else:
          return['.'] + parse_simple_structure(temp)
    elif (len(bt[1])==3):
      return  parse_simple_structure(bt[1]) + ['.']
  if (len(bt)==2):
    return ['.']
################################################################################
# A function to modify the training sequences to have two dollar signs at the
# end of each sequence to enable parsing & training with SCFG class functions
def add_end_symbols(t):
  for i in range(len(t)):
    t[i]+= '$$'
  return t
################################################################################
# Output 0 and 1 to represent the coding state of each terminal annotated
# according to the simple independent HMM.
def parse_hmm(bt):
  if len(tb)==3:
    return parse_hmm(tb[1]) + parse_hmm(tb[2])
  else:
    if tb[0]==3:
      return [1]
    elif tb[0]==4:
      return [0]
    else:
      return ['*']
################################################################################
```

A.4 Grammar Files

A.4.1 Specification of the HMM as an SCFG

This is the estimated HMM obtained after training on the data simulated from the combined grammar.

```
# Alphabet
ab$

# Number of variables
6

# Variable <0>
# Number of productions
2
# Non-terminal productions
<1><5> 0.319600160296
```

```
<2><5> 0.680399839704

# Variable <1>
# Number of productions
3
# Non-terminal productions
<3><1> 0.473172928438
<3><2> 0.322675574088
<3><5> 0.204151497474

# Variable <2>
# Number of productions
3
# Non-terminal productions
<4><1> 0.147207741988
<4><2> 0.61710614972
<4><5> 0.235686108293

# Variable <3>
# Number of productions
2
# Terminal productions
a 0.999233292777
b 0.000766707222682

# Variable <4>
# Number of productions
2
# Terminal productions
a 0.245281352488
b 0.754718647512

# Variable <5>
# Number of productions
1
# Terminal productions
$ 1.0
```

A.4.2 Specification of the Simple SCFG

This is the estimated original SCFG obtained after training on the data simulated from the combined grammar.

```
# Alphabet
ab

# Number of variables
5

# Variable <0>
# Number of productions
10
# Terminal productions
a 0.066597357212
b 0.0348664828919
# Non-terminal productions
```

```
<2><1> 0.0819719342356
<4><3> 0.115970057845
<2><0> 0.18082996869
<0><2> 0.0909097646924
<4><0> 0.11778103888
<0><4> 0.0934225232075
<2><2> 0.0960563276124
<4><4> 0.121594544733

# Variable <1>
# Number of productions
1
# Non-terminal productions
<0><2> 1.0

# Variable <2>
# Number of productions
1
# Terminal productions
a 1.0

# Variable <3>
# Number of productions
1
# Non-terminal productions
<0><4> 1.0

# Variable <4>
# Number of productions
1
# Terminal productions
b 1.0
```

A.4.3 Specification of the Random Combined Grammar

This is true combined grammar I used for testing. I have not included the estimated combined grammar obtained after training on the same set of simulated sequences as the previous separate grammars, but it can be obtained from the author if required.

```
# Alphabet
ab

# Number of variables
31

# Variable <0>
# Number of productions
36
# Terminal productions
a 0.00809297932194
a 0.0542236108199
b 0.0408824997696
b 0.00959662828415
# Non-terminal productions
<8><12> 0.00305837503072
```

```
<7><11> 0.0101990309186
<10><12> 0.00476603639144
<9><11> 0.0290075697727
<8><18> 0.0130419626751
<7><17> 0.00317706490117
<10><18> 0.0308354174052
<9><17> 0.0378263513662
<20><24> 0.0563567405055
<19><23> 0.0433256183477
<22><24> 0.0076874744704
<21><23> 0.016638337229
<20><30> 0.0353428861004
<19><29> 0.040181606374
<22><30> 0.0359165428103
<21><29> 0.0192821253286
<8><6> 0.0115275728743
<7><5> 0.0471228385448
<10><6> 0.0100846833576
<9><5> 0.0536026556838
<2><12> 0.0499205579645
<1><11> 0.0367581410318
<4><12> 0.0375411519527
<3><11> 0.0258743974538
<20><6> 0.0275100981862
<19><5> 0.034736190513
<22><6> 0.0443379905858
<21><5> 0.0262865163794
<2><24> 0.00683294587083
<1><23> 0.0517786133277
<4><24> 0.0130666274143
<3><23> 0.023580161037

# Variable <1>
# Number of productions
18
# Terminal productions
a 0.0846801684789
b 0.00436847628246
# Non-terminal productions
<7><7> 0.0838108321948
<8><9> 0.0202514201915
<7><13> 0.0758100178031
<8><15> 0.0682277476684
<19><19> 0.0381473911584
<20><21> 0.0343690573604
<19><26> 0.086276524993
<20><27> 0.0828392497431
<7><1> 0.0485684254854
<8><3> 0.0848326420472
<1><7> 0.000782497019513
<2><9> 0.0294326856981
<19><1> 0.0359202715031
<20><3> 0.0630982440501
<1><19> 0.0720778880718
<2><21> 0.0865064602505
```

```
# Variable <2>
# Number of productions
18
# Terminal productions
a 0.125047574247
b 0.0136506693494
# Non-terminal productions
<7><8> 0.051947865072
<8><10> 0.0156424770685
<7><14> 0.0897363122305
<8><16> 0.0118014709492
<19><20> 0.0168477320766
<20><22> 0.0304360067964
<19><26> 0.0129224479805
<20><28> 0.0801928655449
<7><2> 0.082995046376
<8><4> 0.104253528656
<19><2> 0.097935094807
<20><4> 0.071202203577
<1><8> 0.0183019556865
<2><10> 0.00109692919805
<1><20> 0.0875635133938
<2><22> 0.0884263069902

# Variable <3>
# Number of productions
18
# Terminal productions
a 0.0580012716264
b 0.042714673048
# Non-terminal productions
<9><7> 0.0823850540057
<10><9> 0.10302971666
<9><13> 0.0809285565142
<10><15> 0.0695260250931
<21><19> 0.0603760747651
<22><21> 0.0153270919148
<21><25> 0.0889964904497
<22><27> 0.0061210168807
<9><1> 0.020051524884
<10><3> 0.0712319406511
<21><1> 0.0884467607744
<22><3> 0.0990796329501
<3><7> 0.0168908143582
<4><9> 0.0137406742993
<3><19> 0.0221078396819
<4><21> 0.061044841443

# Variable <4>
# Number of productions
18
# Terminal productions
a 0.132922773995
b 0.0346039658385
# Non-terminal productions
<9><8> 0.035054998503
```

```
<10><10> 0.0356663383013
<9><14> 0.0984659373155
<10><16> 0.00838100810389
<21><20> 0.0509034080183
<22><22> 0.119700832282
<21><26> 0.0529623423959
<22><28> 0.0912331248005
<9><2> 0.0175566586237
<10><4> 0.0467959631146
<21><2> 0.0345261600576
<22><4> 0.139921540585
<3><8> 0.0453360709803
<4><10> 0.00183281904604
<3><20> 0.022060787499
<4><22> 0.0320752705388

# Variable <5>
# Number of productions
18
# Terminal productions
a 0.062780572474
b 0.0307497372723
# Non-terminal productions
<7><11> 0.0520403285133
<8><12> 0.0379798068162
<7><17> 0.000102600555969
<8><18> 0.0293374651923
<19><23> 0.0768225036416
<20><24> 0.0363806934991
<19><29> 0.0582103465708
<20><30> 0.103616284563
<7><5> 0.0589631926833
<8><6> 0.0961346519217
<19><5> 0.0620787828102
<20><6> 0.0300373997135
<1><11> 0.0988890299799
<2><12> 0.0479722503561
<1><23> 0.0968558629602
<2><24> 0.0210484904766

# Variable <6>
# Number of productions
18
# Terminal productions
a 0.0526873057663
b 0.0426424250532
# Non-terminal productions
<9><11> 0.101423567618
<10><12> 0.0115731423506
<9><17> 0.0351247999248
<10><18> 0.0588177246284
<21><23> 0.0845210163658
<22><24> 0.0821591972226
<21><29> 0.0203218098033
<22><30> 0.0299756121945
<9><5> 0.0507877342011
```

```
<10><6> 0.0778918550335
<21><5> 0.0102067745607
<22><6> 0.0791246434605
<3><11> 0.104685300738
<4><12> 0.00346167189549
<3><17> 0.0660255153042
<4><18> 0.0885699038798

# Variable <7>
# Number of productions
1
# Terminal productions
a 1.0

# Variable <8>
# Number of productions
1
# Terminal productions
a 1.0

# Variable <9>
# Number of productions
1
# Terminal productions
a 1.0

# Variable <10>
# Number of productions
1
# Terminal productions
a 1.0

# Variable <11>
# Number of productions
1
# Terminal productions
a 1.0

# Variable <12>
# Number of productions
1
# Terminal productions
a 1.0

# Variable <13>
# Number of productions
2
# Non-terminal productions
<1><7> 0.305335199805
<2><9> 0.694664800195

# Variable <14>
# Number of productions
2
# Non-terminal productions
<1><8> 0.644552768762
```

```
<2><10> 0.355447231238

# Variable <15>
# Number of productions
2
# Non-terminal productions
<3><7> 0.667168989614
<4><9> 0.332831010386

# Variable <16>
# Number of productions
2
# Non-terminal productions
<3><8> 0.809239537945
<4><10> 0.190760462055

# Variable <17>
# Number of productions
2
# Non-terminal productions
<1><11> 0.61221488977
<2><12> 0.38778511023

# Variable <18>
# Number of productions
2
# Non-terminal productions
<3><11> 0.502027563796
<4><12> 0.497972436204

# Variable <19>
# Number of productions
1
# Terminal productions
b 1.0

# Variable <20>
# Number of productions
1
# Terminal productions
b 1.0

# Variable <21>
# Number of productions
1
# Terminal productions
b 1.0

# Variable <22>
# Number of productions
1
# Terminal productions
b 1.0

# Variable <23>
# Number of productions
```

```
1
# Terminal productions
b 1.0

# Variable <24>
# Number of productions
1
# Terminal productions
b 1.0

# Variable <25>
# Number of productions
2
# Non-terminal productions
<1><19> 0.557454591451
<2><21> 0.442545408549

# Variable <26>
# Number of productions
2
# Non-terminal productions
<1><20> 0.439317259717
<2><22> 0.560682740283

# Variable <27>
# Number of productions
2
# Non-terminal productions
<3><19> 0.757719914706
<4><21> 0.242280085294

# Variable <28>
# Number of productions
2
# Non-terminal productions
<3><20> 0.655803230352
<4><22> 0.344196769648

# Variable <29>
# Number of productions
2
# Non-terminal productions
<1><23> 0.0186277306557
<2><24> 0.981372269344

# Variable <30>
# Number of productions
2
# Non-terminal productions
<3><23> 0.746737745735
<4><24> 0.253262254265
None
```

A.5 Script to test the Inside and the Outside algorithm are consistent

```
##########################################################################
A simple program written to test that the inside prob = outside prob
by Jo Davies January 10th 2006
##########################################################################

from scfg import SCFG
from this_works import MySCFG
import numarray
from numarray import Float64, zeros, resize, sum

H=MySCFG('random_combined_grammar.scfg')
H.output_grammar()
print '------------------------------------------------------------'
nv = H.count_variables()       # total number of non-terminals
nt = H.count_terminal()        # total number of terminals
s='bbaaaaaaaaabaaaba'          # use a reasonably long example sequence
n=len(s)                       # n is the length of the sequence
w=H.cyk_inside(s,0)            # w is the matrix of alpha quantities
y=H.outside(s)                 # y is the matrix of beta quantities

print 'probability calculated via the inside algorithm'
print w[0][n-1][0]             # extract the required element of the matrix

obe = H.get_observed_emission(s) # nb use the observed emission function here.
temp2=zeros((nv),Float64)
for v in range(nv):
    temp2[v] = y[0][0][v]*obe[v][0]
print 'probability calculated via the outside algorithm'
print sum(temp2)
```

A.6 Script to perform parameter training for the combined grammar

```
from scfg import SCFG
from copy import deepcopy
from this_works import MySCFG,
from math import log
H=MySCFG('random_combined_grammar.scfg')
# y=H.training_sequences(20)
def find_global(H, nstart=10,maxit=100,nsim=20):
# H is original grammar, nstart=number of different initial starting points,
# maxit = maximum # of iterations, nsim= # number of training sequences
# training is performed on each times
    print '##############################################################'
    print 'Original grammar from which training sequences are simulated'
    print '##############################################################'
    H.output_grammar()
    print '############## End of original grammar #####################'
    print
    zx=H.training_sequences(nsim)
    print '############# Simulated Training sequences #################'
    print zx
    print '############## End of training sequences #################'
    candidate_grammar=deepcopy(H)
    best_grammar=deepcopy(H)
    best_grammar=best_grammar.set_random_parameters()
    loglike_best=0.0
```

```
    for i in range(nsim):
        a=best_grammar.cyk_inside(zx[i])
        l=len(zx[i])
        loglike_best+=log(a[0][l-1][0])
    for i in range(nstart):
        candidate_loglike=0.0
        print
        print '-----------------------------------------------------------'
        print 'Training iterations with starting values', i+1
        print '-----------------------------------------------------------'
        candidate_grammar=candidate_grammar.training(zx,maxit)
        for j in range(nsim):
            a=candidate_grammar.cyk_inside(zx[j])
            l=len(zx[j])
            candidate_loglike+=log(a[0][l-1][0])
        if candidate_loglike>loglike_best:
            loglike_best=candidate_loglike
            best_grammar=deepcopy(candidate_grammar)
        print '------ End of training for starting values',i+1,'-------'
    print
    print '############### BEST GRAMMAR ####################################'
    print
    print 'best loglikelihood is : ', loglike_best
    print
    print 'Parameters for the best grammar : '
    best_grammar.output_grammar()
    return best_grammar

for i in [50,100,250,500]:
    print
    print '#################################################################'
    print 'Parameter estimation performed with',i,'training sequences simulated'
    print 'from the random combined grammar'
    print '#################################################################'
    print
    find_global(H,5,500,i)
    print '#################################################################'
    print '########### End of training with',i,'training sequences ############'
    print
```

A.7 Script to calculate sensitivity and specificity from annotations

```
####################################################################
# Functions written to calculate sensitivity and specificity rates #
####################################################################
from scfg import SCFG
from copy import deepcopy
from this_works import MySCFG, parse_coding, parse_structure, parse_hmm,
parse_simple_structure, add_end_symbols, combined_sensitivity,
run_combined_sense, compare_annotation, run_compare_annotation
from math import log
from sets import Set
####################################################################
# This Function takes the structure of a sequence represented as a
# list of parenthesis and dots and outputs the set of basepairings.
# They are specified according to their index in the sequence.
```

```
# For example [(2,8)] corresponds to the nucleotides in
# positions 2 and 8 in the sequence forming a base pair.
def extract_basepairing(s): # s is the structure of seq.
    n=len(s)
    bp=Set()
    stack=[]
    for i in xrange(n):
        if s[i]=='.':
            pass
        elif s[i]== '(':
            stack.append(i)
        elif s[i]== ')':
            x=len(stack)-1
            bp.add((stack[x],i))
            stack.pop()
    return bp
########################################################################
# This function returns the base pairings which are common to the
# two structures t and p
def compare_basepairing(t,p):        # t=true structure, p=predicted structure
    if len(t)!=len(p):
        print 'error: the structures are of different lengths'
    else:
        true_bp=extract_basepairing(t)
        predicted_bp=extract_basepairing(p)
        common=true_bp.intersection(predicted_bp)
        return common
########################################################################
# This function gives idea of how well the prediction for grammar
# G compares with the true secondary & gene structure.
def s_s_s(nsim,CG,G):
    similarity_gene_count=0.0
    similarity_structure_count=0.0
    total_count=0.0          # nb total count is same for gene & structure=n
    t_p_structure_count=0.0
    t_p_gene_count=0.0
    t_n_structure_count=0.0
    t_n_gene_count=0.0
    total_positive_gene_count = 0.0
    total_positive_structure_count = 0.0
    total_negative_gene_count = 0.0
    total_negative_structure_count = 0.0
    for i in xrange(nsim):
        T=CG.sample(derivation=1)
        true_traceback=T[1]
        sequence=T[0]
        #print sequence
        n=len(sequence)
        #print n
        predicted_traceback=G.cyk_inside(sequence, al=1)
        true_gene=parse_coding(true_traceback,G)
        #print true_gene
        true_structure=parse_structure(true_traceback)
        predicted_gene=parse_coding(predicted_traceback,G)
        #print predicted_gene
        predicted_structure=parse_structure(predicted_traceback)
```

```
        for i in xrange(n):
            if true_gene[i]==1:
                total_positive_gene_count+=1
                if predicted_gene[i] == 1:
                    t_p_gene_count+=1
                else:
                    pass
            else:
                total_negative_gene_count += 1
                if predicted_gene[i] == 0:
                    t_n_gene_count += 1
                else:
                    pass
        for i in xrange(n):
            if true_structure[i] == '.':
                total_negative_structure_count += 1
                if predicted_structure[i] == true_structure[i]:
                    t_n_structure_count+=1
                else:
                    pass
        total_positive_structure_count+=len(extract_basepairing(true_structure))
        # print total_positive_structure_count
        t_p_structure_count+=len(compare_basepairing(true_structure,
                                                     predicted_structure))
        for i in xrange(n):
            total_count +=1
            if true_structure[i] == predicted_structure[i]:
                similarity_structure_count +=1
            else:
                pass
            if true_gene[i] == predicted_gene[i]:
                similarity_gene_count +=1
    sensitivity_gene = (t_p_gene_count)/(total_positive_gene_count)
    sensitivity_structure=(t_p_structure_count)/(total_positive_structure_count)
    specificity_gene = t_n_gene_count/total_negative_gene_count
    specificity_structure = t_n_structure_count/total_negative_structure_count
    similarity_gene = similarity_gene_count/total_count
    similarity_structure = similarity_structure_count/total_count
    total_similarity = (similarity_gene + similarity_structure)/2
    print 'gene␣sensitivity␣rate␣=␣',sensitivity_gene
    print 'structure␣sensitivity␣rate␣=␣',sensitivity_structure
    print 'gene␣specificity␣rate␣=␣',specificity_gene
    print 'structure␣specificity␣rate␣=␣',specificity_structure
    print 'similarity␣structure␣=␣', similarity_structure
    print 'similarity␣gene␣=␣', similarity_gene
    print 'total_similarity␣=␣',total_similarity

#########################################################################
# This function computes the specificity and sensitivity for the
# two independent models relative to the true annotations of the sequences.
def s_s_s_independent(nsim,G,H,CG):   # the function takes 3 arguments
                                      # CG = the combined grammar
                                      # G = the simple estimated grammar
    similarity_gene_count=0.0         # H = the estimated HMM
    similarity_structure_count=0.0
    total_count=0.0
```

```
t_p_structure_count=0.0
t_p_gene_count=0.0
t_n_structure_count=0.0
t_n_gene_count=0.0
total_positive_gene_count = 0.0
total_positive_structure_count = 0.0
total_negative_gene_count = 0.0
total_negative_structure_count = 0.0
for i in xrange(nsim):
    T=CG.sample(derivation=1)
    true_traceback=T[1]
    sequence=T[0]
    x=add_end_symbols([sequence])
    y=x[0]
    # print sequence
    n=len(sequence)
    ######## predict genes with the HMM ###############
    predicted_gene_traceback=H.cyk_inside(y,al=1)
    predicted_gene=parse_hmm(predicted_gene_traceback)
    # print predicted_gene
    true_gene=parse_coding(true_traceback,CG)
    ######## predict structure with the SCFG ##########
    predicted_structure_traceback=G.cyk_inside(sequence,al=1)
    true_structure=parse_structure(true_traceback)
    # print true_structure
    predicted_structure=parse_simple_structure(predicted_
                                                 structure_traceback)
    # print predicted_structure
    ################ compare the two now ##############
    for i in xrange(n):
        if true_gene[i]==1:
            total_positive_gene_count+=1
            if predicted_gene[i] == 1:
                t_p_gene_count+=1
            else:
                pass
        else:
            total_negative_gene_count += 1
            if predicted_gene[i] == 0:
                t_n_gene_count += 1
            else:
                pass
    for i in xrange(n):
        if true_structure[i] == '.':
            total_negative_structure_count += 1
            if predicted_structure[i] == true_structure[i]:
                t_n_structure_count+=1
            else:
                pass
    total_positive_structure_count+=len(extract_basepairing(true_structure))
    print total_positive_structure_count
    t_p_structure_count+=len(compare_basepairing(true_structure,
                                                   predicted_structure))

    for i in xrange(n):
        total_count +=1
```

```
            if true_structure[i] == predicted_structure[i]:
                similarity_structure_count +=1
            else:
                pass
            if true_gene[i] == predicted_gene[i]:
                similarity_gene_count +=1
    sensitivity_gene = (t_p_gene_count)/(total_positive_gene_count)
    sensitivity_structure = (t_p_structure_count)/(total_positive_
                                                    structure_count)
    specificity_gene = t_n_gene_count/total_negative_gene_count
    specificity_structure = t_n_structure_count/total_negative_structure_count
    similarity_gene = similarity_gene_count/total_count
    similarity_structure = similarity_structure_count/total_count
    total_similarity = (similarity_gene + similarity_structure)/2
    print 'gene sensitivity rate = ',sensitivity_gene
    print 'structure sensitivity rate = ',sensitivity_structure
    print 'gene specificity rate = ',specificity_gene
    print 'structure specificity rate = ',specificity_structure
    print 'similarity structure = ', similarity_structure
    print 'similarity gene = ', similarity_gene
    print 'total_similarity = ',total_similarity

print 'The true combined grammar predictions : '
s=s_s_s(1000,CG=MySCFG('random_combined_grammar.scfg'),
        G=MySCFG('random_combined_grammar.scfg'))
print '------------------------------------------------------------'
print 'The best estimated grammar with 50 sequences predictions :'
s_s_s(1000,CG=MySCFG('random_combined_grammar.scfg'),
      G=MySCFG('best_combined_grammar_50.scfg'))
print '------------------------------------------------------------'
print 'The best estimated grammar with 100 sequences predictions :'
s_s_s(1000,CG=MySCFG('random_combined_grammar.scfg'),
      G=MySCFG('best_combined_grammar_100.scfg'))
print '------------------------------------------------------------'
print 'The best estimated grammar with 250 sequences predictions :'
s_s_s(1000,CG=MySCFG('random_combined_grammar.scfg'),
      G=MySCFG('best_combined_grammar_250.scfg'))
print '------------------------------------------------------------'
print 'The best estimated grammar with 500 sequences predictions :'
s_s_s(1000,CG=MySCFG('random_combined_grammar.scfg'),
      G=MySCFG('best_combined_grammar_500.scfg'))
print '------------------------------------------------------------------------------------------'
print '------------------------------------------------------------------------------------------'
print 'Sensitivity , Similarity and specificity for the independent trained models'
s_s_s_independent(1000,G=MySCFG('estimated_simple_grammar.scfg'),
                  H=MySCFG('estimated_hmm.scfg'),
                  CG=MySCFG('random_combined_grammar.scfg'))
```