Some examples of generating and sampling language from stochastic context free grammars

Anil Alexander

Swiss Federal Institute of Technology (EPFL)

alexander.anil@epfl.ch

1. Introduction

1.1. Automatic Speech Recognition & the Language Model

Probabilistic language models with n-gram grammars, are widely in use for tasks like automatic speech recognition, part of speech tagging and word sense disambiguation. They are very useful in systems that would require predictive information about words are likely to appear.

One of the main uses of the language model is in *Automatic Speech Recognition*. Speech recognition is the process of converting voice utterances in the form of audio signals, into their corresponding phonetic or text transcriptions. The speech recognition process is described in fig 1.

1.1.1. Signal Processing of the acoustic speech signal

Input for the speech recognition process is created by making recordings of human speech and passing it through a signal processing processing. Typically this recording for the input can be performed with microphones in an environment specially created for such recordings, or by even recordings of poor quality telephone lines. The quality of recording can have an impact on the speech recognition process, and thus must be chosen according to the application domain of the recognizer. For a query system over the telephone for restaurant information, the recordings will have only the quality of a telephone signal, and thus the feature extraction box should have mechanisms to remove noise or distortions particular to the telephone line. The signal processing section converts the acoustic signal received into a set of feature vectors.

1.1.2. Phone Likelihood Estimator

The auditory front end processes the signal and passes a sequence of feature vectors to the *phone likelihood estimator*. The phone likelihood estimator is typically a neural network or an implementation using Hidden Markov Models *HMM* and estimating the likely phone sequences for the acoustic signal that is passed to it. Both the HMM and the artificial neural network have already been trained using acoustic recordings and their phonetic transcriptions before. Now, either of them, or a hybrid combination of the two can perform the classification of the input signal into the probabilities of the phone belonging to a particular phoneme.

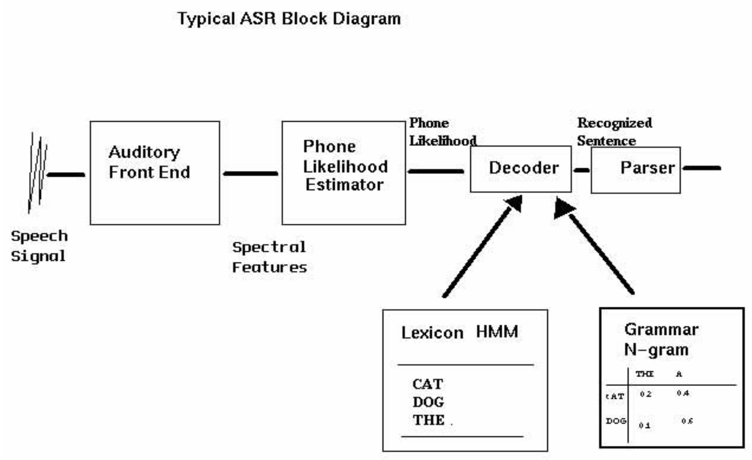

Figure 1: Schema of speech recognition process in an Automatic Speech Recognizer

1.1.3. Decoder

It is in the decoder that the most likely sequence of words, or the sentence corresponding most closely to the input signal is given. In this phase, we have already a vector corresponding to the probability that the phoneme corresponds to the speech feature signal. From this we would like to build the most probable sentence sequence and corresponding to the input vector. For this, the *Viterbi algorithm* is used which uses dynamic programming to find out the most possible sequence of words. In order to compute the most probable sequence of words, it is necessary to know the *transition probability* of each word going to the other *(bigram transition probability)* or the probability of the next word , given the two words preceding that word (*(trigram transition probability)*. The *lexicon* would see that the dynamic programming takes into consideration only *l*egal sequences of letters or correctly spelt words.

1.1.4. Parsing and Understanding

The decoder returns the most probable sentence corresponding to the utterance spoken. At this point it would be necessary to parse the sentence for its syntactic correctness and in order to understand what the import of what has been said. This is of special significance in systems that try to perform a certain duty according to requests uttered by the user. A restaurant query response system like that of the *Berkeley Restaurant Project [4]* would have to make sense of a command like *"I don't want a fancy place, any place that serves good steak will do"* and understand that it should find a medium range restaurant who's specialties include steak. This is not an easy task and would rely heavily on context knowledge and good mapping mechanisms of the sentences to their semantic alternatives.

1.2. The Language Model

The **language model** is a major component of any speech recognition engine and has been briefly described in the following section.

The language model provides the knowledge source for the speech recognition system engine. It is the language model which helps to predict the next word, or the most probable sequence of words which make up the sentence pertaining to the utterance.

The first component of the language model is the **lexicon** which consists of the vocabulary. The vocabulary contains all of the possible words in the voice system which may be encountered. If we require specialized vocabularies such as in many professional systems, a smaller and more focussed lexicon is used which contains terms pertaining to the context required. For instance, a voice recognition based query system for restaurants, will have a lexicon that has more words pertaining to food, music and gastronomy as opposed to sports matters.

The second component of the **language model** is the **grammar**. It defines the structure and format of the text allowed at any point in the utterance. Without grammar, every word in the lexicon would have an equal likelihood of occurring at any point within the utterance, requiring algorithms which are too complex, making continuous speech recognition impossible. These systems use the grammar to constrain the word choice at any point, and to point to the most probable word choice at that point.

Like the acoustic models in the ASR, the grammars have also been built upon *Discrete Markov models*, called **n gram** grammars. In Markov model grammars, the underlying idea is that the probability of a word is a function of the previous $n-1$ words encountered.This assumption allows for the efficient computation of the likelihood for accurate recognition. When we approximate this idea to mean that the probability of the word is a function of just the word which appeared previous to it, we use **bigram grammars**. Similarly, for **trigram grammars** we decide that every word depends most closely on its two predecessors. Thus bigram grammars are the set of Probabilities for all pairs of words within the lexicon and similarly a trigram grammar is the set of probabilities for all triplets of words within the lexicon.

*B*igram and *t*rigram grammars are the most popular types of Markov grammars employed, simply because they seem to be the level of complexity required for embodying the natural constraints of human language

In order to create these grammars we require the system to build probability estimations for all possible word pair groups or groups of three words for the language and context that the system uses. This can be done either statistically or through algorithms that will generate this from a probabilistic grammar.

The **grammar perplexity** is a measure of how constrained the grammar is, based upon the grammar itself

1.3. Motivation for using Probabilistic Language Modeling

Probabilistic language models with n-gram grammars, are widely in use for tasks like automatic speech recognition, part of speech tagging and word sense disambiguation. They are very useful in systems that would require predictive information about words are likely to appear.

On the other hand, these grammars prove difficult to work with because they have, in general, a large number of parameters, and thus a reliable estimation would require a huge training corpus augmented by good smoothing mechanisms. It is difficult to model. These models are not also easily extendible, i.e, if a new word is added to the vocabulary, then it is not possible for the existing n gram model to say anything about the n grams containing the new term.

It is in this situation that, the use of stochastic context free grammars (SCFG) becomes significantly more useful and natural. They are not dependent on a large number of parameters, can capture linguistic generalizations, and can be extended quite simply. They also bring the benefits of being easily understood or written by linguists.

1.4. Motivation for using SCFGs to derive the *n gram* grammar

The SCFG is very good for modeling long distance dependencies and hierarchical structure and they are relatively easy to write and understand. Since many applications such as speech recognition require n gram grammars, it would be convenient to have a method to convert automatically, a given stochastic context free grammar into its corresponding n gram grammar model. Thus it is very convenient, if we have a method to convert an SCFG into a grammar

Using Stolcke's algorithm, we are able to compute the SCFG bigrams, and these give a good coverage of local and lexical dependencies. What is attempted at, in this work is to use the SCFG directly to provide the word transition probabilities for a problem.

1.5. Using the SCFG directly as the Language Model

In a set up that a recognizer is passing a string to the parser, for instance, *I like French* , the parser would parse this prefix string and will come up with all the possible terminations of the string. It is necessary to have a list of possible alternatives, considering *French* as the expansion of some non terminal, say 'desired objects'. This list could be wines, cuisine, women.

Thus there is a necessity to have the probabilities of: $P(w_i|w_1w_2....w_{i-1})$ where w_i is the one of the members of the list *wines, cuisine, countryside* and $w_1w_2....w_{i-1}$ representing the prefix string *I like French*

In order to compute these probabilities, we require the list to produce the probability that a given non terminal expands into a terminal.

If all sentences were unambiguous, it would be sufficient to produce exhaustively, all the transition probabilities. Thus there will be multiple parses for the same prefix string. We thus need to combine the probabilities for non terminals from different parses to create the language model.

1.6. Background on *n gram* Grammar model

An *n gram* grammar is a set of probabilities $P(w_n|w_1w_2...w_{n-1})$ giving the probability that w_n follows a word string $w_1w_2..w_{n-1}$ for each possible combination of the w's.

A SCFG is a set of phrase structure rules, annotated with probabilities of choosing a certain production given the left hand side non terminal. A typical SCFG would be of the following form.

S	$\rightarrow$	NP VP	[1.0]
NP	$\rightarrow$	N	[0.4]
NP	$\rightarrow$	Det N	[0.6]
VP	$\rightarrow$	V NP	[0.7]
VP	$\rightarrow$	V	[0.3]
Det	$\rightarrow$	a	[0.2]
Det	$\rightarrow$	an	[0.2]
Det	$\rightarrow$	the	[0.6]
N	$\rightarrow$	orangutan	[0.1]
N	$\rightarrow$	human	[0.2]
N	$\rightarrow$	child	[0.7]
V	$\rightarrow$	watches	[0.8]
V	$\rightarrow$	beats	[0.2]

In an n-gram grammar, for a vocabulary of size n, the bigram grammar will have n^2 free parameters, and the corresponding trigram grammar would have n^3 such parameters. Thus, if the grammar has a few thousand rules, the number of words in its vocabulary, the free parameters just for the bigrams would be several million, and those for the trigrams would be of the order of billion parameters.

On the other hand, a SCFG would have a much smaller set of free parameters. We can observe a divergence between the 2 models that increases as the size of the vocabulary increases.

1.7. Evaluation of the efficacy of the SCFG as a probabilistic model

It has been argued in literature that SCFGs are in theory, not adequate probabilistic models for use in natural languages. This is because of the conditional independence assumptions that they embody. This is discussed in *Magerman and Marcus, 1991 [6]* and *Briscoe and Carroll 1993 [1]*. Stolcke has suggested that these shortcomings can be overcome by using SCFGs with very specific, semantically oriented categories and rules.

In spite of this, it is to be noted that although a larger and more sophisticated language model would give better results as a probabilistic model, *n grams* are likely to be still important in fields like automatic speech recognition.

In speech recognition, the standard speech decoding techniques which use dynamic programming is based on a first order Markov assumption, which means that a given phone depends only on the phone that appeared directly before it. Now, this first order Markov dependency is satisfied by the bigram models. It is to be noted that this first order Markov assumption is satisfied by the bigram model, and not models that are more sophisticated.

2. Steps in the Conversion of a SCFG into a bigram grammar

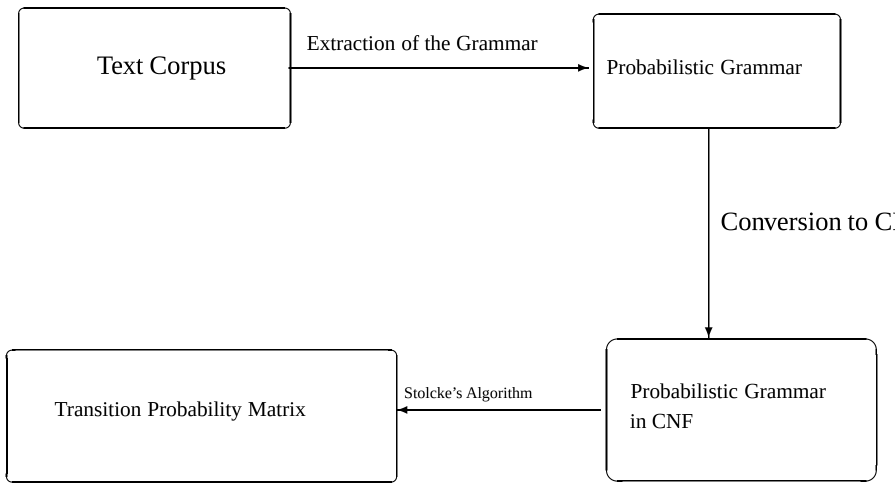
Text Corpus
Extraction of the Grammar
Probabilistic Grammar
Conversion to CNF
Transition Probability Matrix
Stolcke's Algorithm
Probabilistic Grammar in CNF

Figure 2: Schema for Conversion of Stochastic Context Free Grammar into transition Probability Matrix

Now the system developed, follows the schema given in figure 2 above. We start with a text corpus, preferably one that has a good coverage of the target domain of the grammar. For instance, if the grammar is going to be used in a recognition system for stock market queries, then it should depend heavily on corpora dealing with financial matters, for example, *The Wall Street Journal* corpus.

After this, we would require to convert the general grammar thus extracted, into CNF, which is required for the conversion phase. This phase will remove all the unit productions as well as the rule with more than two non terminals on the right hand side.

Now, with the probabilistic CNF grammar as input to the conversion program, the transition probabilities are derived from the grammar. These transition probabilities are output and can then be used in conjunction with the lexicon as the language model.

3. Converting an SCFG into a text corpus

3.1. Algorithm in brief:

Assuming we have the SCFG (in CNF)

1. Take starting symbol.Push into stack.
2. Pop the top of the stack.
3. if terminal put into the output sentence
   else
   expand(Non Term) push expansion into the stack from Left to Right
4. Expansion involves choosing the appropriate rule for expansion For this, random number helps choose which of the rules are better
5. goto 3
6. Termination: Stack is Empty

3.2. Implementation

The program *scfg2txt* has been implemented in C. It expects a grammar file of the format:

S	$\rightarrow$	A B	(1.0)
A	$\rightarrow$	A B	(0.3)
A	$\rightarrow$	B A	(0.1)
A	$\rightarrow$	a	(0.6)
B	$\rightarrow$	b	(1.0)

3.3. Validation

Validation of the output of this program is done by *counter.scfg.pl* which gives the frequencies of the bigrams in the text.

This program samples the number of bigrams for the entire text presented to it. In order to validate the results of *scfg2txt* it would be necessary to bear in mind that when generating the statistics of of scfg2txt, the sampling has to be performed over the aggregate of seperate sentences, and not treating the entire text corpus as one.

3.4. Issues: How to Prevent Infinite Recursion

3.4.1. Why Recursion ?

Recursion will happen because if there is a rule in the grammar such that

$A \rightarrow AB$

Now it is possible that exactly this rule is chosen over and over again without expanding the non terminal A to a terminal. In such a situation we will have infinite recursion happening. If we have a stack based implementation of this program, it is possible that for a badly constructed grammar, the number of recursive calls are indefinitely high and would lead to stack overflow. We would like to handle this sort of a situation.

3.4.2. Methods to counter this

In order to counter this

Naive approach (functional): Impose a limit on the maximum recursion allowed

One of the methods that is suggested is to impose a limit on the maximum recursion allowed. If this approach is followed, the language generated would be a subset of the language generated by the grammar. The probability of all the trees will thus not sum up to one. This approach will not theoretically work absolutely correctly because it is bound to skew the probabilities of the subtree that we are cutting off. In order to reach the best compromise here, we can limit the number of recursive calls to $2n - 1$, where n is the typical maximum length of a sentence generated by the grammar.

The other approach is to:

Let the grammar expand indefinitely, hoping that the probabilities of giving terminals will finally force the grammar to a finite (though large) set of sentences. In practice, if the probabilities for a non terminal going to a terminal are not very small, this process will converge quite soon.

In the program *scfg2txt* implementing this, it was seen that infinite recursion did not happen, for most of the grammars used, which had reasonable probabilities for a non terminal giving a terminal.

Finally, it would be sufficient to check whether the grammar is ’proper’ or ’consistent’ since we have the result that if the grammar(G) is of the form:

S	$\rightarrow$	S S	(p)
S	$\rightarrow$	a	$(1 - p)$

P(L(G)) will be the min(1,(1-p/p))

Thus if we can guarantee that the grammar will be of this form, it is possible for the algorithm to terminate. Then this will converge. However, checking whether a grammar is consistent or not is not very easy.

4. Conversion of a SCFG to Chomsky Normal Form

Conversion of a grammar from a probabilistic grammar to a probabilistic grammar in Chomsky Normal Form. A brief discussion of the Chomsky normal form is given after this section.

The following is the method to convert a generalized grammar into one in Chomsky Normal Form. Direct conversion of a grammar into CNF is fairly straightforward, and the new CNF grammar would have to give exactly the same language as the original grammar. However, what we have to consider here is a general probabilistic grammar, and it has to be converted into CNF, while still conserving the probabilistic characteristics of the language thus created.

It would be necessary to suppress the rules of the following type in the grammar.

1. X $\rightarrow$ a w b
2. X $\rightarrow$ $X_1X_2X_3$
3. X $\rightarrow$ Y

Note that if the grammar has null productions, they have to be removed as well.

1. Transformation 1 **X $\rightarrow$ a w b**

   For rules of this type, where a and b are not null and the probability of production of this rule is 'p', it is necessary to introduce a unique (we shall discuss the production of such a 'unique' non terminal, later on). Q$_w$(r). Replace the rule 'r' with the rule

   X $\rightarrow$ $aQ_w(r)b(p)$

   and introduce a new rule

   $Q_w(r)$ $\rightarrow$ w (1)

   This will distribute the probability mass of the rule *X* $\rightarrow$ *a w b* across a two rules, and will not change the distribution of the language produced by the grammar.

2. Transformation 2: **X** $\rightarrow$ $X_1X_2....X_n$

   where n $>$ 2 and the probability of this rule is p it would be necessary to introduce a collection of non terminals of the type Q(r)i which are all unique.

   Thus a rule of the form X $\rightarrow$ $X_1X_2X_3...X_n$ would be replaced by the following.

   X $\rightarrow$ $Q_1(r)X_n(p)$
   $Q_i(r) \rightarrow Q_{i+1}(r)X_{n-i}$ (1) for $i = 1..n-3$
   $Q_{n-2}(r) \rightarrow X_1X_2(1)$

   also, it would be necessary to suppress or to remove the rule 'r'. Now again, the distribution in the language created by the CNF will not be different from that created by the original grammar. In this case as well, the probability mass of the disobedient rule X $\rightarrow$ $X_1X_2X_3...X_n$ is distributed over several rules.

3. Transformation 3: **X $\rightarrow$ Y**

   Now, replacement of this rule is a little more tricky than the rest. In order to do this, it is necessary to calculate the probability that $X => *Y$. This derivation is discussed in the following section.

Basically a typical grammar that would be encountered would be of the form:

S	$\rightarrow$	A B	(1.0)
A	$\rightarrow$	A B	(0.3)
A	$\rightarrow$	B A	(0.1)

A	$\rightarrow$	a	(0.6)
B	$\rightarrow$	A B B A	(0.6)
B	$\rightarrow$	b	(1.0)
A	$\rightarrow$	B	(0.5)

This grammar contains some irregularities (2 & 3) that do not allow it to conform to the CNF. These are to be removed according to the transformations 2 & 3.

4.1. Calculating $X \Rightarrow *Y$

To explain this, let us take the example of the phrase, **I want Indian** that has already been identified by the recognizer. Let the recognizer pass this string to to the parser, it will produce the following words, ’foods’, ’restaurants’ , ’places’, ’cuisine’ etc. The parser parses the prefix string, and then looks at every non terminal symbol that the parser is predicting next. For each such non terminal, it is necessary now to look at its left corner list- the list of terminal symbols that the Non terminal can create on the left fringe of some parse tree. Look at the following illustration in figure 3.

4.2. Execution of *scfg2cnf.pl*

We have developed a program in Perl [5] in order to perform this conversion.Let us consider that the input grammar to *scfg2cnf.pl* is :

S	$\rightarrow$	A B	(1.0)
A	$\rightarrow$	A B	(0.3)
A	$\rightarrow$	B A	(0.1)
A	$\rightarrow$	a	(0.3)
A	$\rightarrow$	A B B A	(0.1)
B	$\rightarrow$	b	(1.0)
A	$\rightarrow$	B	(0.2)

Notice that the rules A $\rightarrow$ B (0.2) and A $\rightarrow$ A B B A (0.1) are the two that do not allow the grammar to be in CNF, and these should be removed in order to convert them. These are to be removed according to the transformations 2 & 3.

Thus it will be necessary to apply the transformations maintaining the probability distribution of the words in the language of the output grammar thus generated.

The program *scfg2cnf.pl* applies these transformations to the input grammar. And as a result we get the output grammar that is in the Chomsky Normal form.

Then the output grammar is

S	$\rightarrow$	A B	(1.0)
A	$\rightarrow$	A B	(0.3)
A	$\rightarrow$	B A	(0.1)
A	$\rightarrow$	a	(0.3)
B	$\rightarrow$	b	(1.0)
A	$\rightarrow$	b	(0.04)

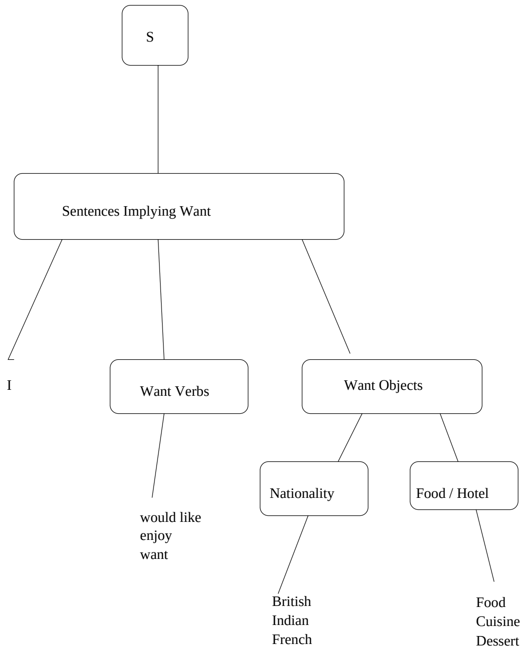

Figure 3: Prefixes and Left corner derivations

A	$\rightarrow ABB\_6\_7$ A	(0.1)
ABB_6_7	$\rightarrow AB\_3\_3$ B	(1)
AB_3_3	$\rightarrow A\_1\_3$ B	(1)

Notice that new rules have been added which have *unique* names for their non terminals and they still maintain the probability distribution across the language that they create, just as the input grammar did.

4.3. Computation of the Probability of Initial Substring generation by Stochastic Context Free Grammars

Calculating $X \Rightarrow *Y_l$

The computation of probability of initial substring generation by Stochastic Context free grammars has been discussed in *Jelinek & Lafferty, 1991*[3]. Speech recognition language models would require this sort of left corner derivation probabilities as well as the right corner probabilities.

The creation of the right corner derivation probability matrix follows exactly the same logic used for computing the left corner derivation probability. We denote the right corner derivation probability matrix is conceptually equal to the left corner derivation matrix if we consider flipping the right hand side of the grammar rules. P$X \Rightarrow *Y_l$ and P$X \Rightarrow *Y_r$ represent the transition left and right corner derivation probabilities.

It is interesting to see whether these derivation matrices actually exist, and formally describe the conditions necessary for their existence.

1. The SCFG is proper iff for all non terminals X, the rule probabilities sum up to one.

2. G is consistent iff it defines a probability distribution over finite strings.

3. G has no useless non terminals iff all non terminals X appear in at least one derivation of some string.

4.3.1. Some definitions

Left Corner Relation $X \rightarrow_L Y$ iff there exists a production for X that has an RHS starting with Y.

$$X \rightarrow_L Y\lambda$$

Probabilistic left corner relation $P_L = P_L(G)$ is the matrix of probabilities $P(X \rightarrow_L Y)$ is defined as the total probability of choosing a production for X that has Y as a left corner.

$$P(X \rightarrow_L Y) = \sum_{X \rightarrow Y\lambda \epsilon G} P(X \rightarrow_L Y\lambda)$$

If these conditions are satisfied then the *matrices R_L and R_U* are known to exist. It has also been observed (Stolcke) that, an SCFG may be inconsistent and still have converging left-corner and unit production matrices. This implies that consistency is a stronger constraint.

4.3.2. Computational Issues

Computation spent in Matrix Inversions

Calculation of the prediction matrix or the right and left substring derivation matrix is defined as a geometric series derived from a matrix P.

$R = I + P + P^2 + P^3 + ... = (I - P)^{-1}$

Both P and R are indexed by the non terminals in the grammar. The matrix P is derived from the SCFG rules and probabilities (either the left-corner relation or the unit production relation)

If we can agree that we have a fixed grammar to start with, the time taken by the precomputation of left-corner and unit production matrices may not affect the computation time, since this can be done off-line, and once and for all.

However, this is not generally the case. Now, even if the matrix P is sparse, the matrix inversion can be prohibitively large for huge numbers of non terminals. Empirically, it can be seen that the inversion of a matrix with rank *n* with a bounded number of *p* non zero entries per row can be inverted in order $0(n^2)$ whereas a full matrix of size nXn would require the time $0(n^3)$.

4.4. Normal form for the SCFGs *Chomsky Normal Form*

A grammar is in *Chomsky Normal Form* if every production is of the form
A $\rightarrow$ BC or A $\rightarrow$ Terminal
Chomsky's grammatical form is particularly useful when one wants to prove certain facts about context free languages. This is because assuming a much more restrictive kind of grammar can often make it easier to prove that the generated language has whatever property you are interested in.

Any context free grammar or stochastic context free grammar can be converted into one in CNF which will generate exactly the same language, and in the case of SCFGs, exactly the same language, with precisely the same distribution of n grams. Thus for any parse of the the original grammar will be reconstructible from a parse in the CNF grammar.

5. Implementation framework: SCFG to ngrams

Stolcke and Segal [7] have developed an algorithm for calculating *n*-grams from stochastic context free grammars. Some of the advantages of their algorithm is that the procedure does away with many of the problems associated with conventional *n*-gram production such as estimation from sparse data and linguistic structure. What this algorithm basically does is to compute substring expectations, and it does so by solving systems of linear equations which are derived from the input grammar.

5.1. Summary of Stolcke's Algorithm

1. Calculate the prefix and suffix (left and right) derivation probabilities for each (non terminal, word) pair.

2. Compute the coefficient matrix and the right hand sides for the systems of linear equations as per the following equations.

   The elements of the coefficient matrix (A) are given by:

$$A[X,U] = \Sigma(X \to YZ)P(X \to YZ)((\delta(Y,U) + \delta(Z,U)) \tag{1}$$

$$\text{where } \delta(X,Y) = 1 \text{ if } X = Y \text{ else } 0$$

$$b[X] = P(X \to W) + \Sigma(X \to YZ)P(X \to YZ)P(Y \Rightarrow_L w_1)P(Z \Rightarrow_r w_2) \tag{2}$$

3. LU decompose the coefficient matrix

4. Compute the unigram expectation for each word in the grammar by solving the LU system for the unigram right hand-sides that had been calculated in step 2.

5. Compute the bigram expectation for each word pair by solving the LU system for the bigram right hand-sides calculated in step 2.

6. Compute each bigram probability $P(w_2|w_1)$ by dividing the bigram expectation $C(w_1w_2|S)$ by the unigram expectation $C(w_1|S)$.

5.2. Stolckes Algorithm Coding

The following describes Stolcke's algorithm in implementable sub steps.

5.2.1. Algorithm Requirements

1. Requires the grammar to be in CNF.

   i.e A→BC
   or A →Terminal.

2. The SCFG has to be *consistent*

Note: Consistency is illustrated with the following grammar:

S	→	S S	(1 - p)
S	→	a	(p)

Find expected freq (C) of a unigram x

$$\begin{aligned} & C = P(S \rightarrow a)+P(S \rightarrow SS)(C+C) \\ \Rightarrow \quad & C = p+2qC \end{aligned}$$

Now for C not to be infinite or negative, it is necessary that $p < 0.5$

5.2.2. Steps of Algorithm

1. **Initialize the grammar rules**

   Load the grammar from input file. Create Matrix of (non-terminal,word) pair, filling it with the probabilities that a non terminal will give the word. if the non terminal gives the word directly, then simply fill up the place noting that it directly gave the word, else calculate and fill up. This step is not a necessary part of the algorithm, but makes the task of calculating the parameters in the next step easier.

2. **Calculating the Parameters required**

   Since we are considering only the special case of the bigrams. The following eqns need to be solved:

   For Bigrams,

   Expectation of word pair w1,w2:

   $$C(w_1, w_2|X) = \sum_{\forall X \rightarrow YZ} P(X \rightarrow w).(c(w_1w_2|Y) + c(w_1w_2|Z))$$
   $$+P(Y \Rightarrow w_1).P(Z \Rightarrow w_2)$$

   For unigrams

   Expectation of w:

   $$C(w|X) = \sum_{\forall X \rightarrow YZ} P(X|w1) + P(X \rightarrow YZ)(c(w1|Y) + c(w1|Z))$$

   It is with these expectation values that the transition probability $P(w2|w1)$ can be calculated.

   In this step, we need to calculate the coefficient matrix A[XU] and the matrix B to solve the eqn $[A - I]c = b$

   Determining *coefficient matrix A* **and the** *matrix b* **in** $[A - I]c = b$

   The elements of the coefficient matrix (A) are given by:

   A[X,U]= Σ (X $\rightarrow$ YZ) P(X $\rightarrow$ YZ) (δ(Y,U)+δ(Z,U)) where δ(X,Y)=1 if X=Y else 0

   The elements of the right hand matrix 'b' are:

   $$b[X] = P(X \rightarrow W) + \Sigma(X \rightarrow YZ)P(X \rightarrow YZ)P(Y \Rightarrow_L w_1)P(Z \Rightarrow_r w_2)$$

   Now, with these, we frame the equations:

   $(A - I)c = B$ Where c gives us the expectation that we require.

   From the expectation we can then compute the bigram probability P(w2|w1)

3. **LU Decomposition**

   To solve the equations calculated in the previous step, it would be useful to use an algorithm suitable for solving large number of linear equations. Here, we have n equations, and n unknowns. We choose the LU decomposition of the LHS of the equation.

   This is quoted from MATLAB help.

   LU factorization

   [L,U] = LU(X) stores an upper triangular matrix in U and a ”psychologically lower triangular matrix” (i.e. a product of lower triangular and permutation matrices) in L, so that X = L*U. X must be square. [L,U,P] = LU(X) returns lower triangular matrix L, upper triangular matrix U, and permutation matrix P so that P*X = L*U.

4. **Calculate Unigram Expectations**

   Get the unigram expectations for each word in the grammar solving LU created in 2.

5. **Calculate Bigram Expectations**

   Similarly compute the bigram expectations for each word pair by solving the LU system for the bigram, right hand sides computed in step 2.

6. **Calculate the bigram Probability** Compute the bigram probability P(w2|w1) by dividing the bigram expectation c(w1w2|S) by the unigram expectation

5.2.3. Function List

1. **LU Decomposition**

   [L,U,P] = LU(X) returns lower triangular matrix L, upper triangular matrix U, and permutation matrix P so that P*X = L*U.

   The use of this functions, is that these L and U matrices can be used to solve systems of equations.

2. **Solving the matrix**

   To solve a system of equations

   Ax=B

   Apply LU decomposition on A such that A=LU, x=$(L.U)^{-1}B$

   Thus $x = U^{-1}L^{-1}B$ and this is solved by forward elimination $(forL^{-1})$ and backward elimination for U^{-1}.

   Once A has been factored into L and U, this two-step procedure can be used repeatedly to solve the system of equations for different values of b.

   Forward Elimination

   The sketch of the function Forward Elimination could be as follows:

```
{Forward elimination with partial pivoting}
  I <- 1
   loop
```

```
{ Determine largest value in column I from row
 I+1 to N.  Call this row Imax. }
{ Switch rows Imax and I. }
{ Do forward elimination.  Multiply row I by
A(J,I)/A(I,I) and subtract from row J.  J varies
from I+1 to N. }
I <- I + 1
exitif (I >= N)
endloop
```

Back Substitution The sketch of Back Substitution could be like this

```
{Backsubstitution}
        I <- N
        loop

{ Determine sum of A(I,J)*X(J) for J between
    I+1 and N }
    X(I) = (B(I) - Sum)/A(I,I)
    I <- I - 1
    exitif (I <= 0)
        endloop
```

3. **Build Coefficient Matrix A**

   We require a function that will give, fairly automatically, the coefficient matrix from the SCFG directly.

   The entry a[X,U] in A corresponds to the X and U non terminals.

   To fill an entry a[X,U] it is enough to get from the grammar, first all the rules that have X on their right and sum the probabilities according to the formula discussed in the algorithm.

4. **Build Matrix b**

   This is matrix is built similarly to according to the formula discussed in the algorithm. Note that, the elements of this matrix correspond to each word, word bigram.

5. **Solve the LU system for Bigram right handsides**

   Call the functions *LUDecompose*,*Forward elimination* and *Backsubstitution*

6. **Solve the LU system for Unigram right hand-sides**

   Just as in the previous step, call the functions *LUDecompose*, *Forward elimination* and *Back-substitution* and solve the equation (I-A)c=b.

7. **Compute the resulting Bigram Probability P(w2|w1)**

   P(w2|w1) is calculated by dividing the bigram expectation vector calculated in the previous steps by the unigram expectation vector.

5.3. Example of Stolcke's Algorithm on a simple SCFG

Let us illustrate Stolcke's Algorithm by the following simple grammar. Let the S.C.F Grammar(G) be:

S	$\rightarrow$	A B	(1.0)
A	$\rightarrow$	A B	(0.4)
A	$\rightarrow$	a	(0.6)
B	$\rightarrow$	b	(1.0)

Note that this grammar is in *Chomsky Normal Form*, and does not have any cyclic dependencies. Intuitively we can see that, there will be no 'aa' or 'ba' string produced by this grammar. On the other hand, 'ab' and 'bb' can be produced.

Now let us apply **Stolckes Algorithm**

1. **Calculating the matrix A[X,U]**

   The formula is:
   A[X,U]= Σ (X $\rightarrow$ YZ) P(X $\rightarrow$ YZ) (δ(Y,U)+δ(Z,U))

   where δ(X,Y)=1 if X=Y else 0
   Thus the coefficient Matrix A[X,U] is

$$\begin{bmatrix} S & A & B & \\ S & 0 & 1.0 & 1.0 \\ A & 0 & 0.4 & 0.4 \\ B & 0 & 0 & 0 \end{bmatrix}$$

2. **To calculate B[X]**

   The formula for B[X] for bigrams is

$$b[X] = P(X \rightarrow W) + \Sigma(X \rightarrow YZ)P(X \rightarrow YZ)P(Y \Rightarrow_L w_1)P(Z \Rightarrow_r w_2) \qquad (3)$$

   Thus for the word pair 'ab'

   B[X] = [.6 0.24 0]

   The reasoning behind this derivation is that
   b[S]= 0+ P(S $\rightarrow$ AB)(A $\rightarrow$ a...) which is .6
   b[A]= 0+ P(A $\rightarrow$ AB)P(A $\rightarrow$ a)P(B $\rightarrow$ b) which is .24
   b[B]= 0+ P(B $\rightarrow$?) which is also 0

   Now, for the word pair 'aa', B[X] = [0 0 0]
   for the word pair 'bb', B[X]= [0.4 .16 0]
   and finally for the word pair 'ba', B[X]=[0 0 0]

3. **Solving the Equations for bigrams** Now we need to solve the equation [I-A]C=B where I is the *identity matrix* and C is the *expectancy matrix* corresponding to the word pair.

   Thus we have the matrix multiplication

   $$\begin{bmatrix} 0 & 1.0 & 1.0 \\ 0 & 0.4 & 0.4 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} C_S \\ C_A \\ C_B \end{bmatrix} = \begin{bmatrix} B_S \\ B_A \\ B_B \end{bmatrix}$$

   Solving for the word pair 'ab'

   we get

   $$C_S - C_A = -0.6$$
   $$0.6C_A - 0.4C_B = -0.24$$
   $$C_B = 0$$

   Thus we get $C_S(ab)$ = -1.0, which is the expectation of 'ab' given this grammar, since S is the starting symbol

   Similarly the expectation for 'aa' being in this grammar is $C_S(aa)$ = 0, for 'bb' C_s(bb) =-1.115 and for word pair 'ba' = 0.

   We note here, that this matches the intuition introduced along with the grammar.

4. **Solving the Equations for Unigrams**

   For unigrams, B[x]= [1.6 0.64 0] for the word *a* and *b*, B[X]= $[1.6667, 0.6667, 1]$

   $$\begin{bmatrix} 1.6 & 1.6667 \\ 0.64 & 0.6667 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} C_S \\ C_A \\ C_B \end{bmatrix} = \begin{bmatrix} B_S \\ B_A \\ B_B \end{bmatrix}$$

   Solving the equations for 'a' (Unigram expectation for 'a')

   $$C_S - C_A = -1.6$$
   $$0.6C_A - 0.4C_B = -0.64$$
   $$C_B = 0$$

   we get C_S =-2.6667=C(a|S) (C(a|S) Unigram expectation for 'a')

   Solving the equations for 'b'

   $$C_S - C_A = 1.6667$$
   $$0.6C_A - 0.4C_B = 0.6667$$
   $$C_B = 1$$

   which give C_S =-4.437= C(b|S) (C(b|S) Unigram expectation for 'b')

5. **Calculating the Transition Probabilities**

   Now for P(w2|w1)= C(w1w2|S)/C(w1)

   P(b|a) = C(ab|S)/C(a|S) = -1/-2.6667 = 0.375
   P(a|b) = C(ba|S)/C(b|S) = 0/-4.4437 =0
   P(b|b) = C(bb|S)/C(b|S) = -1.115/-4.4437 =.2501
   P(a|a) = C(aa|S)/C(a|S) = 0/-2.6667 = 0

6. Comparison : Stolcke's Algorithm and *scfg2txt*

Using Stolcke's Algorithm on the simple grammar given in the previous section, we obtained the transition probabilities as follows:
P(b|a) = 0.375
P(a|b) = 0
P(b|b) = .2501
P(a|a) = 0

Now with running the *scfg2txt* program with the same grammar, with no limit on the number of recursions, a set of 10000 sentences was created. The validation program *count.pl* finds out the number of occurrences of a bigram word word pair. The plot of the variation of bigram probabilities for the word pairs 'ab' and 'bb' are given in *figure 1*. Its results are as follows: For a total number of bigrams 26641:

Word Pair $(w1w2)$	Number of Bigrams	Prob	Predicted Prob (Stolcke)
ab	10000	0.376	0.375
ba	0	0	0
bb	6642	0.249	0.250
aa	0	0	0

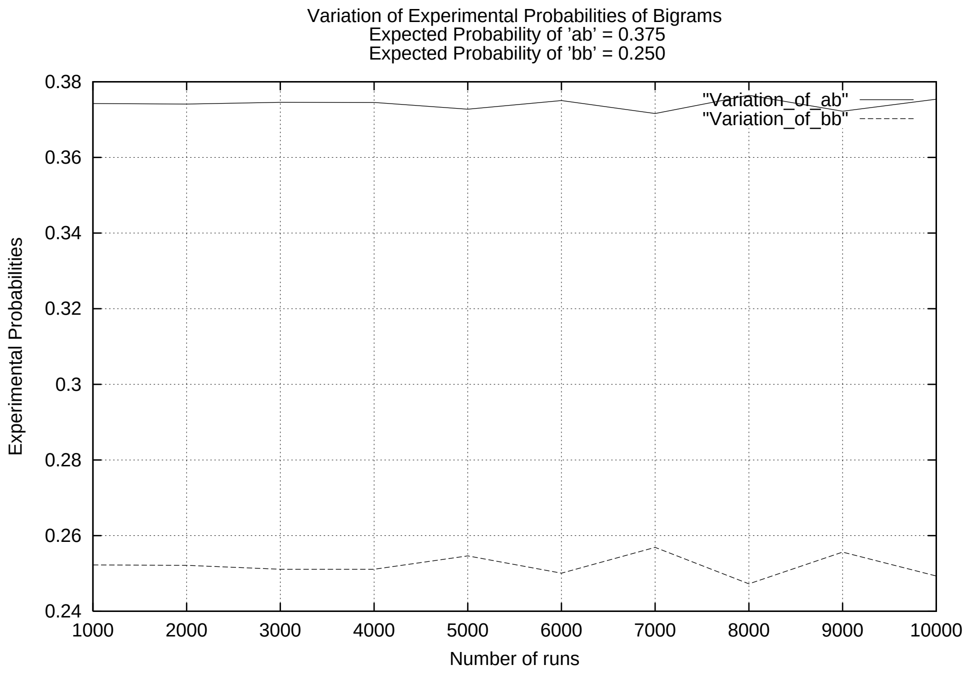

Figure 4: Plot of the variation of Bigram Probabilities

Thus it can be concluded, that the transition probabilities generated by Stolckes algorithm, correspond very well, to the transition probabilities estimated by creating the corpus from the SCFG.

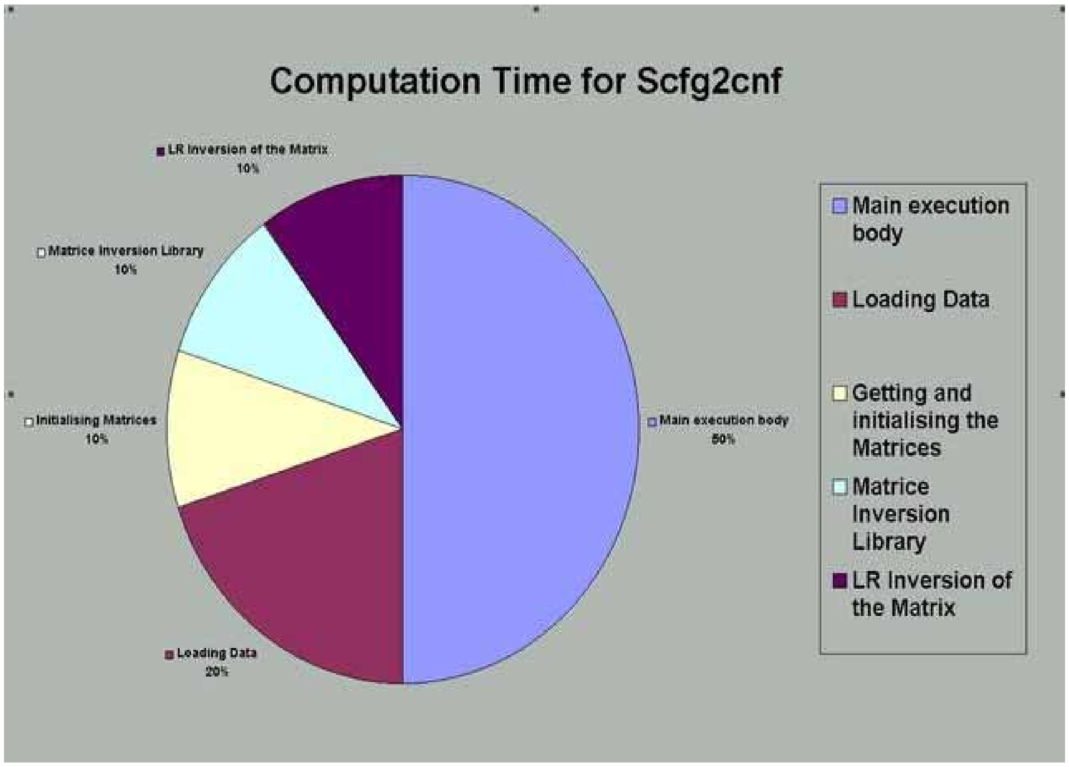

Figure 5: Computational time analysis for program *scfg2cnf.pl*

7. Opinions on estimation of Probabilistic CFGs

It is argued that assigning probabilities to a context free grammar will result in a distribution of words in the language that may not accurately map the distribution of the words in the parent corpus. This improper distribution is because the probability of all finite parse trees is less than one. In other words, the language generated by the grammar may have a probability less than one. This is because the derivation tree may have a probability greater than zero of *n*ever terminating.

Zhiyi Chi & Stuart German '98 [2] discusses that, if production probabilities are estimated from data, then the production probabilities always yield proper distributions. For instance, if we have a parsed corpus, that we treat as a collection of independent samples from a grammar, then it is reasonable to expect that, if the trees in the sample are finite, then the estimation procedure of the production probabilities will produce a system that will assign a zero probability to the remaining parse trees in the infinite set of trees.

8. Insight gained during implementation

When working with grammars of large sizes (typically with several thousand rules and a couple of hundred non terminals), the computation time involved is a matter of concern. Since the creation of a transition probability matrices involves computationally expensive matrix inversions, it is of interest to look at the amount of time each part of the program takes, to gain an understanding of the relative importance and difficulty of each salient part. This is represented in Fig 5.

What we notice from the graph of the relative computation time is that the bulk of time of execution of the program is just in the main execution body, and not in the matrix inversions. The matrix

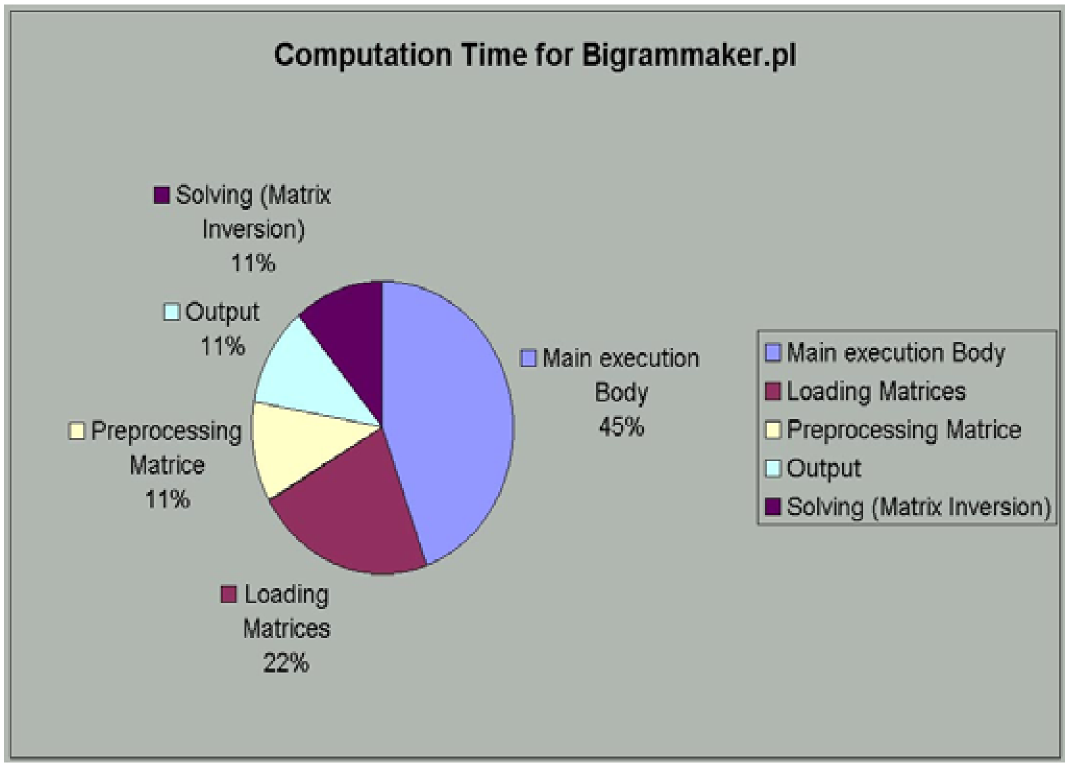

Figure 6: Computational time analysis for program *bigrammaker.pl*

related operations take around 30 to 40% of the time for both *bigrammaker.pl* as well for *scfg2cnf*.

The *scfg2cnf*, does not involve as many matrix inversion operations as *bigrammaker.pl*, as it does not require the solving of linear equations for its computation, and here matrix related computations take 30% of the time. On the other hand *bigrammaker.pl* takes well over 40% of the time in steps including preprocessing the matrices, loading them, and solving the set of linear equations.

It is to be noted that this records only computation time, and that memory usage issues come into play, when the size of the grammar increases. While, this is hardly an issue with smaller test grammars with around 50 rules, the computational time increases sharply as the number of rules in the grammar increases. If this program has to be scaled to real world grammars with several thousand rules, it maybe necessary to augment the data-structures to minimize the memory usage.

9. Future Work

The existing work deals with the creation of bigram transition probability matrix. It would be interesting to extend this its working to include the transition probabilities for n grams, thus making the system truly expandable for problems of any kind, which will require an *n gram* language model.

A graphical user interface *GUI* could be added as a front end, to this suite of programs, in order to make its use more intuitive.

The system currently implemented could be made more memory efficient in order to accommodate large grammars, with several thousand rules. It would be sufficient to modify and optimize the data-structures associated with the grammar.

A. Appendix

A.1. Manual for *scfg2txt*

- **NAME** scfg2txt - Returns the language produced by a stochastic context free grammar (SCFG).

- **SYNOPSIS** ../scfg2txt [OPTION] [FILE]...

- **DESCRIPTION** Generates a subset of the language produced by the SCFG to standard output.

  -n, The number of iterations of the creation, corresponds to the number of sentences of the language.

  -g, The name of the grammar file input

- **EXAMPLE**

  ```
  ./scfg2txt -n300 -g grammarfile
  ```

- **AUTHOR**

  Written by Anil Alexander

- **Notes** : The format of the input grammarfile should be as follows:

  ```
  S → A B :1.0
  A → A B :0.4
  A → a   :0.6
  B → b   :1.0
  ```

  Running this program with $n = 1$ will produce a single sentence corresponding to this grammar. In order to approximate the language, run it with n corresponding to a large number (say 10,000)

A.2. Manual for *counter.scfg.pl*

- **NAME** counter.scfg.pl - Samples the input text (or language) and gives the relative percentages of the bigrams in the text.

- **SYNOPSIS** ../counter.scfg.pl [OPTIONS] [FILE]...

- **DESCRIPTION**

  Samples the input text, and counts the number of bigrams and their relative frequency. It outputs the total number of bigrams in the text as well as their relative frequencies.

- **EXAMPLE**

  ./counter.scfg.pl out.txt

- **AUTHOR**

  Modifications on work done at the University of Minnesota, Duloth by *Ted Pedersen & Satanjeev Banerjee* by Anil Alexander

- **Notes**

  This program samples the number of bigrams for the entire text presented to it. In order to validate the results of *scfg2txt* it would be necessary to bear in mind that when generating the statistics of of scfg2txt, the sampling has to be performed over the aggregate of seperate sentences, and not treating the entire text corpus as one.

A.3. Manual for *scfg2cnf*

- **NAME** scfg2cnf.pl - converts a general stochastic context free grammar into the Chomsky Normal form.

- **SYNOPSIS** ../scfg2cnf.pl [FILE]...

- **DESCRIPTION**

  Converts a general stochastic context free grammar into the Chomsky Normal form, while maintaining the probabilistic distribution of the words in the language generated, just as in case of the parent grammar.

- **EXAMPLE**

  ./scfg2cnf.pl grammarfile

  (On Unix systems which have Perl installed)

- **AUTHOR** Written by Anil Alexander alexander.anil@epfl.ch

- **Notes** Here the expected format of the input grammar is:

$S \rightarrow A\ B$	(1.0)
$A \rightarrow A\ B$	(0.3)
$A \rightarrow B\ A$	(0.1)
$A \rightarrow a$	(0.3)
$A \rightarrow A\ B\ B\ A$	(0.1)
$B \rightarrow b$	(1.0)
$A \rightarrow B$	(0.2)

A.4. Manual for *bigrammaker.pl*

- **NAME** maker.pl - makes the transition probability matrix, if fed a CNF.

- **SYNOPSIS** ../maker.pl [OPTION] [FILE]...

  (On Unix systems which have Perl installed)

- **DESCRIPTION** Generates the transition probability matrix or the bigram probability matrix, given that it is given a context free grammar as input.

  -g, The name of the grammar file input

- **EXAMPLE**

  ./perl maker.pl grammarfile on Windows systems with Perl installed. Alternatively, if you have the binary executable for the program on Windows it is

  ```
  maker.pl.exe grammarfile
  ```

  Its usage is fairly simple on Unix, as almost all flavours of Unix now carry Perl as part of their standard shipping

  The command here would be

  ```
  maker.pl
  ```

- **AUTHOR** Written by Anil Alexander

- **Notes** : The format of the input grammarfile should be as follows:

  ```
  S → A B :1.0
  A → A B :0.4
  A → a   :0.6
  B → b   :1.0
  ```

  The program prints to STDOUT, the transition probabilities, of going from one non terminal to another.

B. References

[1] Ted Briscoe and John Carroll. Generalised probabilistic LR parsing of natural language corpora with unification-based grammars. *Computational Linguistics*, 19(1):61–74, 1993.

[2] Zhiyi Chi and Stuart Geman F. Estimation of probabilistic context-free grammars. *Computational Linguistics*, 24(2):298–305, 1998.

[3] F. Jelinek and J. D. Lafferty. Computation of the probability of initial substring generation. *Computational Linguistics*, 17(3):315–323, 1991.

[4] D. Jurafsky, C. Wooters, G. Tajchman, and J. Segal. The berkeley restaurant project. In *Proc. ICSLP '94*, pages 2139–2142, Yokohama, Japan, 1994.

[5] Tom Christiansen Larry Wall and Jon Orwant. *Programming Perl*. ISBN: 81-7366-265-7. O' Reilly, 3rd edition, 2000.

[6] D. M. Magerman and M. P. Marcus. Pearl: A probabilistic chart parser. In *Proceedings of the European ACL Conference*, pages 40–47, 1991.

[7] Andreas Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. Technical Report TR-93-065, Berkeley, CA, 1993.