Stochastic Definite Clause Grammars

Christian Theil Have
Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
cth@ruc.dk

Abstract

This paper introduces Stochastic Definite Clause Grammars, a stochastic variant of the well-known Definite Clause Grammars. The grammar formalism supports parameter learning from annotated or unannotated corpora and provides a mechanism for parse selection by means of statistical inference. Unlike probabilistic context-free grammars, it is a context-sensitive grammar formalism and it has the ability to model cross-serial dependencies in natural language. SDCG also provides some syntax extensions which makes it possible to write more compact grammars and makes it straight-forward to add lexicalization schemes to a grammar.

1 Introduction and background

We describe a stochastic variant of the well-known Definite Clause Grammars [12], which we call *Stochastic Definite Clause Grammars* (SDCG).

Definite Clause Grammars (DCG) is a grammar formalism built on top of Prolog, which was developed by Pereira and Warren [12] and was based the principles from Colmerauers metamorphis grammars [6]. The grammars are expressed as rewrite rules which may include logic variables, like normal Prolog rules. DCG exploit Prologs unification semantics, which assures equality between different instances of the same logic-variable. DCG also allows modeling of cross-serial of dependencies, which is known to be beyond the capability of context-free grammars [4].

In stochastic grammar formalisms such as probabilistic context-free grammars (PCFG), every rewrite rule has an associated probability.

For a particular sentence, a grammar can produce an exponential number of derivations. In parsing, we are usually only interested in *one* derivation which best reflects the intended sentence structure. In stochastic grammars, a statistical inference algorithm can be used to find the most probable derivation, and this is a very successful method for parse disambiguation. This is especially true variants of PCFGs which condition rule expansions on lexical features. Charniak [3] reports that "a vanilla PCFG will get around 75% precision/recall whereas lexicalized models achieve 87-88% precision recall". The reason for the impressive precision/recall of stochastic grammars is that the probabilities governing the likelihood of rule expansions are normally derived from corpora using parameter estimation algorithms. Estimation with complete data, where corpus annotations dictate the derivations, can be done by counting expansions used in the annotations. Estimation with incomplete data can be accomplished using the Expectation-Maximization (EM) algorithm [8].

In stochastic unification grammars, the choice of rules to expand is stochastic and the values assigned to unification variables are determined implicitly by rule selection. This means that in some derivations, instances of the same logic variable may get different values and unification will *fail* as result.

Some of the first attempts to define stochastic unification grammars did not address the issue of how they should be trained. Brew [2] and Eisele [9] tries to address this problem using EM, but their methods have problems handling cases where variables fails to unify. The resulting probability distributions are missing some probability mass and normalization results in non-optimal distributions.

Abney [1] defines a sound theory of unification grammars based on Markov fields and shows how to estimate the parameters of these models using Improved Iterative Scaling (IIS). Abney's proposed solution to the parameter estimation problem depends on sampling and only considers complete data. Riezler[13] decribes the Iterative Maximization algorithm which also work for incomplete data. Finally, Cussens [7] provide an EM algorithm for stochastic logic programs which handles incomplete data and is not dependent on sampling.

SDCG is implemented as a compiler that translates a grammar into a program in the PRISM language. PRISM [16, 19, 15] is an extension of Prolog that allows expression of complex statistical models as logic programs. A PRISM program is a usual Prolog program augmented with random variables. PRISM defines a probability distribution over the possible Herbrand models of a program. It includes efficient implementations of algorithms for parameter learning and probabilistic inference. The execution, or sampling, of a PRISM program is a simulation where values for the random variables is selected stochastically, according to the underlying probability distribution. PRISM programs can have constraints, usually in the form of equality between unified logic variables. Stochastic selection of values for such variables may lead to unification failure and resulting failed derivations must be taken into account in parameter estimation. PRISM achieves this using the fgEM algorithm [17, 20, 18], which is an adaptation of Cussen's Failure-Adjusted Maximization algorithm [7]. A central part

International Conference RANLP 2009 - Borovets, Bulgaria, pages 139–143

of Cussens algorithm is the estimation of the number of times rules are used in failed derivations. PRISM estimates failed derivations using a failure program, derived through a program transformation called First Order Compilation (FOC) [14].

2 Stochastic Definite Clause Grammars

Stochastic Definite Clause Grammars is a stochastic unification based grammar formalism. The grammar syntax is modeled after, and is compatible with, Definite Clause Grammars. To facilitate writing stochastic grammars in DCG notation, a custom DCG compiler has been implemented. The compiler converts a DCG to a PRISM program, which is a stochastic model of the grammar.

Utilizing the functionality of PRISM, the grammar formalism supports parameter learning from annotated or unannotated corpora and provides and mechanism for parse selection through statistical inference. Parameter learning and inference is performed using PRISMs builtin functionality.

SDCG include some extensions to the DCG syntax. It includes a compact way of expressing recursion, inspired by regular expressions. It has expansion macros used for writing template rules which allow compact expression of multiple similar rules. The grammar syntax also adds a new conditioning operator which makes possible to condition rule expansions on previous expansions.

2.1 Grammar syntax

A grammar consist *grammar rules* and possibly some helper Prolog rules and facts. A grammar rule takes the form,

```
H ==> C1,C2,..,Cn.
```

H is called the *head* or left-hand side of the rule and `C1,C2,...,Cn` is called the *body* or right-hand side of the rule. The head is composed of a `name`, followed by an optional parameter list and an optional conditioning clause. It has the form,

```
name(F1,F2,...,Fn) | V1,V2,...,Vn
```

The `name` of the rule is a Prolog atom. The parameter list is a non-empty parenthesized, comma-separated list of features which may be Prolog variables or atoms. The number of features in rules is referred to as its arity. The optional conditioning clause starts with the pipe (included) and is a non-empty, comma-separated list of Prolog variables or atoms, or a combination of the two. The conditioning clause may also contain expansion macros in the case of unexpanded rules.

The *body* of a rule is a comma-separated list of constituents, of which there are four basic types: Rule constituents, embedded Prolog code, symbol lists and expansion macros.

Rule constituents are references to other SDCG grammar rules. They have the same format of Prolog goals, but may not be variables. A rule constituent consists of a `name` which is a Prolog atom, followed by an optional parenthesized, comma-separated list of *features*; `(F1..Fn)`. Features are either Prolog atoms or variables. Rule constituents may additionally have prefix regular expression modifiers. The allowed modifiers are `*` (kleene star) meaning *zero or more* occurrences, `+` meaning *one or more* occurrences and `?` meaning *zero or one* occurrence.

Embedded code takes the form, `{ P }`, where `P` is a block of Prolog goals and control structures. The allowed subset of Prolog corresponds to what is allowed in the body of a Prolog rule, but with the restriction that every goal must return a ground answer and may not be a variable. Also, while admitted by the syntax, meta-programming goals like `call` are not allowed. The goals unify with facts and rules defined outside the embedded Prolog code, but not in other embedded code blocks.

Symbol lists are Prolog lists of either atoms or variables or a combination of the two. The list usually take the form, `[ S1,S2,..,SN ]`, but the list operator `|` may also be used. However, it is required that every variable in the list is ground. A symbol list may not be empty.

Expansion macros have the form,

```
@name(V1,V2,...,Vn)
```

where `name` is an atom and is followed by a non-empty parenthesized, comma-separated list, `V1...Vn`, consisting of atoms or variables or a combination. A macro corresponds have a corresponding goal, `name/n`, which must be defined.

2.2 Procedural semantics

The grammar rules govern the rewriting the head of a rule into the constituents in the body of a rule. A rule is rewritten when all its constituents have been expanded. The order of the constituents in the body are significant and they are expanded in a left-to-right manner. The rewriting process always begins with the start rule and progress in a depth-first manner. A rule constituent in the body of a rule is thus a reference to one or more other rules of the grammar. A grammar rule is said to be matched by a constituent rule if the `name` and arity of are the same and their features unify. A *constituent rule* is expanded by replacing it with the body of some matching rule. Symbol lists are terminals and are not expanded. Embedded Prolog code is expanded to nothing and executed as a side-effect. The expansion terminates when the body only contains symbols or some constituent cannot be expanded (derivation fails).

When a constituent matches more than one rule there might be more than one derivation. The choice of the rule to expand given such a constituent, should be seen in the light of the probabilistic inference being performed. In general, we can assume that only the derivations relevant to the probabilistic query being used are expanded.

2.3 Statistical semantics

A rule $r \in R^{n,a}$ with the distinct name n and arity a has a probability $P(r) \in [0, 1]$ of being expanded in

place of a matching rule constituent.

A rule r_i may have a condition (conditioning clause), in which case the probability of its expansion depend on the probability of the condition $c_i \in C^{n,a}$ being true, $C^{n,a}$ being the set of possible values for condition clauses for rules in $R^{n,a}$. Each distinct condition (clause value) has a separate probability, such that

$$\sum_{i=1}^{|C^{n,a}|} P(c_i) = 1$$

We denote number of rules in $R^{n,a}$ satisfying a particular condition c, $|n, a, c|$.

It holds for the sum of probabilities of such rules $r_i^{n,a} \in R^{n,a}$ that,

$$\sum_{i=1}^{|n,a,c|} P(r_i^{n,a}|c) = 1$$

where the probability of a rule r given a combination of conditions c is their product, $P(r|c) = P(r)P(c)$. If rules with the same head ($R^{n,a}$) occur without conditioning ($C^{n,a} = \emptyset$) then the condition *true* is assumed and $P(true) = 1$.

The probability of a derivation is the product of the probabilities of all rules used in that derivation. The probability of given sentence is the sum of the probabilities for each possible derivation of the sentence. A derivation may be unsuccessful due to failure of variable unification. The probability of all possible derivations, successful and unsuccessful sums to unity, given by the relation, $P_{success} = 1 - P_{failure}$.

2.4 The translated SDCG

The compiler behaves similar to a usual DCG compiler, by transforming rules in a DCG syntax to Prolog rules with difference lists. In addition to these normal Prolog rules, which we call *implementation rules*, special *selection rules* are used to control the stochastic derivation process. Each rule head with the same number and arity in the original DCG grammar are grouped together and managed by one *selection* rule. The selection rule has the same name and number of features as the of the original rule, but any ground atoms in the original rule are replaced by variables in the selection rule. Consider the two rules in the example below,

```
np(Number) ==> det(Number), noun(Number).
np(Number) ==> noun(Number).
```

The generated selection rule for the two rules is shown below:

```
np(Number, In, Out) :-
    msw(np(1), RuleIdentifier),
    np_impl(RuleIdentifer, Number, In, Out).
```

The `msw` goal is a special PRISM primitive which implements simulation of a random variable, which here stochastically unifies `RuleIndentifier` to a value given the name of the random variable. The name of the random variable is assigned according to the name of the nonterminal and its arity. For instance, since `np` has an arity of 1, the corresponding random variable is named `np(1)`. The possible outcomes of this particular random variable are `np_1_1` and `np_1_2`.

The first parameter of the implementation rules uniquely identifies them and this name corresponds to an outcome of the random variable used by the selection rule. The implementation rules for the above grammar is shown below:

```
np_impl(np_1_1, Number, In, Out) :-
    det(Number, In, InOut1),
    noun(Number, InOut1, Out).
np_impl(np_1_2, Number, In, Out) :-
    noun(Number, In, Out).
```

2.5 Grammar extensions

Regular expression operators, expansion macros and conditioning clauses, which are extensions of the usual DCG syntax, makes it possible to express aspects of the grammar more compactly. These operators are implemented in a preprocessing step which expands the compacted grammar.

2.5.1 Regular expression modifiers

Regular expression operators is a way of expressing recursion in a more convenient manner. An example grammar rule containing all the allowed regular expression operators is shown below:

```
name ==> ?(title), *(firstname), +(lastname).
```

The regular expression operators are implemented by generating some additional rules and replacing the original constituent (`orig_const`), which the operator is applied to, with another constituent (`new_const`). All regular expression operators can be implemented generating a subset of the following rules:

```
1) new_const ==> []
2) new_const ==> orig_const
3) new_const ==> new_const, new_const
```

The `?` operator is implemented by adding rules 1-2. The `+` operator is implemented adding rules 2-3 and the `*` operator is implemented adding all the rules. The name `new_const` is symbolic. The compiler use a naming scheme, which avoids conflicting names: The name of the regular expression modifier is prefixed to the constituent name. For instance `*(firstname)` becomes `sdcg_regex_star_firstname/0`. The compiler only adds the implementation rules for the same regular expression once, even if it is used in multiple rules.

2.5.2 Expansion macros

Macros are special Prolog goals embedded in grammar rules. They may occur in both the head and the body of rules. Grammar rules with macros are *meta grammar rules*; they act as templates for the generation similar rules. The result of macro expansion of a rule is a set of rules, equal in structure to the original rule, but where each macro is replaced with selected parameters from an answer for the goal. The ground

input to the goal is omitted by default. It is possible to explicitly configure which parameters of a goal should be inserted using an `expand_mode` directive. If the goal contains more than one non-ground/answer parameter, the answer parameters are inserted comma-separated. If a rule contains more than one macro, then the set of expanded rules correspond to a cartesian product of the answers for all the macros. When several macros in the same rule use the same name for a variable, this works as a constraint on the answers for the macros. This is exactly as if the goals of the macros were constituents in the body of a Prolog rule.

The original motivation for expansion macros was integration of lexical resources. Suppose that we wish to integrate the lexicon defined by the following simple Prolog program,

```
word(he,sg,masc). word(she,sg,fem).
number(Word,Number) :- word(Word,Number,_).
gender(Word,Gender) :- word(Word,_,Gender).

expand_mode(number(-,+)).
expand_mode(gender(-,+)).

term(@number(Word,N),@gender(Word,G)) ==>
    [ Word ].
```

We select the variables which should be inserted in the resulting rules. A minus (`-`) indicates that the parameter is an *input* parameter and will not appear in place of the substituted macro and a plus (`+`) indicates an output parameter which will appear in place of the macro.

Since the macros in the example share the `Word` variable, it must unify to the same value for all macros. The result of performing macro substitutions on the grammar above is another, macro free, grammar:

```
term(sg,fem)==>[she]. term(sg,masc)==>[he].
```

2.5.3 Conditioning

Conditioning makes it possible to condition an expansion on previous expansions, which is useful for adding lexicalization schemes to the grammar. An example of a rule with a conditioning clause is shown below:

```
n1(A,B,C) | a,b ==> n2, n3.
```

The values of the conditioning clause, (`a,b`), corresponds to values for parameters in the head of the rule. This relation is defined by adding a fact, `conditioning_mode(n1(+,+,-))`, to the grammar. The parameter is a compound term with the same functor as a corresponding nonterminal. The parameters of this term indicate which parameters to grammar rules named by the functor are subject to conditioning. For instance, the conditioning mode in the above example states that the two first parameters of `n1` should be conditioned on (indicated with `+`), but the last one should not (indicated with `-`).

In the simple grammar fragment below, we illustrate a simple conditioning scheme, inspired from [5], where we condition on a single headword:

```
sentence ==>
   np(nohead,NPHead),vp(NPHead,VPHead).
np(ParentHead,Head) | @headword(W) ==>
   det(ParentHead,DetHead),noun(DetHead,Head).
vp(ParentHead,Head) | @headword(W) ==>
   verb(ParentHead,Head).
```

We have not specified conditioning modes for the rules, but in each case the condition corresponds to the first parameter in the head. Assume that the macro `@headword` expands to each of the words (terminals) in the grammar. The headword is propagated from the terminals, so for instance in the sentence rule, the choice of which `vp` rule to expand depends on headword propagated from the preceding `np`. Conditioning a rule on every word implicates that the rule *given that word* will have a *distinct* probability distribution.

More advanced lexicalization schemes can easily be created using the conditioning mechanism. The limitation lies in the order in which variables conditioned on are unified (and thus derivation order). It is not possible to condition on a variable which is not yet ground.

2.5.4 Syntax extensions example

As an illustrative example which applies all the syntax extensions, we demonstrate a part of speech tagger expressed with SDCG. A part of speech tagger is can be implemented as a stochastic regular grammar/Hidden Markov Model (HMM). A HMM based POS tagger can be created in SDCG with a single rule,

```
tag_word(Prev, @tag(Cur),  [CurRest])
        | @tag(_) ==>
     @consume_word(W),
     ?(tag_word(Cur,_,Rest)).
```

This assumes definition of words, tags, a conditioning mode declaration. The grammar rule consumes one word for each time it is expanded. Note that there will be separate rules for each word, because of the `@consume_word` macro, which expands the rule for all the words in the lexicon (enclosing them in square brackets). The next constituent in the body is a recursive reference to the rule itself. It is governed by the regular expression operator `?`, which indicates that the constituent may or may not be matched. If it is not matched, we have termination of the recursion. The model defined by the rule is a fully connected second order HMM model, where the expanded grammar has a rule for each possible transition.

To illustrate the use of the tagger we consider an example from [3], defined here as a simple Prolog lexicon,

```
tag(none). tag(det). tag(noun).
tag(verb). tag(modalverb).
word(the). word(can). word(will). word(rust).
```

We introduce a helper rule to interact with the lexicon and also a start rule,

```
consume_word([Word]) :-  word(Word).
start(TagList) ==> tag_word(none,_,TagList).
```

To train the grammar we feed it with tagged sentences,

```
learn([ start([det,noun,modalverb,verb],
    [the,can,will,rust],[]),
  start([det,noun,modalverb,verb],
    [the,can,can,rust],[]),
  start([det,noun,noun], [the,can,rust],[]),
  start([det,noun],[the,rust],[]),
  start([modalverb,noun,verb],
    [will,rust,rust],[]),
  start([noun,modalverb,verb],
    [will,can,rust],[]),
  start([noun,noun],[the,the],[])  ]).
```

When the grammar/tagger has been trained we can pose a `viterbi` query to find the most likely tag sequence for a sentence,

```
| ?- viterbig(start(T,[the,can,will,rust],[])).
T = [det,noun,modalverb,verb|_4794] ?
yes
```

3 Evaluation

To test the grammar formalism with regard to more realistic grammars, a grammar for a subset of the English language was developed. The grammar consists of about 90 rules, not counting pre-terminal rules, and models various different sentences types. It was originally modeled after the descriptions of context-free grammars for English in [11] and extended with some common agreement features, chosen with the tagset of the Brown corpus in mind.

In a small scale experiment, the grammar was used to parse 4000 select sentences from the Brown corpus [10], between 2 and 60 words in length. Parsing was relatively fast - usually less than 100 milliseconds per sentence excluding the time used to load the grammar and sentences. Training the grammar on the same sentences takes quite a while longer, approximately 4 minutes.

Introducing a lexicalization scheme similar to [5] increases the resulting number of random variables and affects both training time and inference time drastically. Some optimizations are needed to work with such lexicalized grammars in more realistic settings.

A limitation seems to be the first order compilation process in PRISM which takes a lot of time and consumes a lot of memory as the grammar grows larger. With recursion, the process may not complete, which has motivated the addition of an option to limit the depth of the derivation tree.

Precision/Recall was not measured, as the intention was only to measure the performance of the formalism, not the usefulness of the grammar.

4 Conclusion and future work

We introduced Stochastic Definite Clause Grammars, a new stochastic unification-based grammar formalism syntacticly compatible with Definite Clause Grammars. The grammar formalism borrows the expressivity and ability to model natural language phenomena from DCG, but also enjoys the benefits from of statistical models. SDCG extends DCG syntax which allow expression of probabilistic grammars very compactly. This naturally includes probabilistic regular grammars (such as the demonstrated POS tagger) and probabilistic context-free grammars, but also includes context sensitive grammars. It was demonstrated that lexicalization schemes can be compactly expressed in the formalism through conditioning and macros.

Some optimizations are needed in order to utilize large grammars (and training sets) for natural languages. Alternative methods for parameter learning may be explored.

Finally, the success of the grammar formalism depends on the applications that using it. SDCG will evolve with the development of applications using it.

References

[1] S. P. Abney. Stochastic attribute-value grammars. *Computational Linguistics*, 23(4):597–618, 1997.

[2] C. Brew. Stochastic HPSG. In *Proceedings of EACL-95*, February 1995.

[3] E. Charniak. Statistical techniques for natural language parsing. *AI Magazine*, 18(4):33–44, 1997.

[4] N. Chomsky. *Syntactic Structures*. Mouton, The Hague, 1957.

[5] M. J. Collins. *Head-driven statistical models for natural language parsing*. PhD thesis, Penn university, Jan. 01 1999.

[6] A. Colmerauer. Metamorphosis grammars. In L. Bolc, editor, *Natural Language Communication with Computers*. Springer-Verlag, 1978.

[7] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245, 2001.

[8] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39:1–38, 1977.

[9] A. Eisele. Towards probabilistic extensions of constraint-based grammars. *DYANA-2 deliverable R 1.2 B*, 1994.

[10] W. N. Francis and H. Kuçera. Brown corpus manual. 1979.

[11] D. Jurafsky and J. H. Martin. *Speech and language processing*. Prentice Hall, 2000.

[12] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13, 1980.

[13] S. Riezler. Probabilistic constraint logic programming. *CoRR*, cmp-lg/9711001, 1997. informal publication.

[14] T. Sato. First order compiler: A deterministic logic program synthesis algorithm. *Journal of Symbolic Computation*, pages 605–627, 1989.

[15] T. Sato. A glimpse of symbolic-statistical modeling by prism. *Journal of Intelligent Information Systems*, 2008.

[16] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research (JAIR)*, 15:391454, 2001.

[17] T. Sato and Y. Kameya. A dynamic programming approach to parameter learning of generative models with failure. In *Proceedings of ICML Workshop on Statistical Relational Learning and its Connection to the Other Fields (SRL2004)*, 2004.

[18] T. Sato and Y. Kameya. Learning through failure. In L. D. Raedt, T. Dietterich, L. Getoor, and S. H. Muggleton, editors, *Probabilistic, Logical and Relational Learning - Towards a Synthesis*, number 05051 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[19] T. Sato and Y. Kameya. New advances in logic-based probabilistic modeling by prism. *Probabilistic Inductive Logic Programming LNCS 4911, Springer*, page 118155, 2008.

[20] T. Sato, Y. Kameya, and N.-F. Zhou. Generative modeling with failure in prism. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI2005)*, pages 847–852, 2005.