PPL 1.0
Propositional Probabilistic Logic Package
User Manual

André da Costa Teves, Danillo Paulo Couto
Paulo S. de S. Andrade, Fabio G. Cozman

andreteves@gmail.com, danillo_couto@yahoo.com.br
p_s_s_a@yahoo.com.br, fgcozman@usp.br

August 16, 2007

Contents

1 Introduction

1.1 A little bit of history ...

Proposals for the union between logic and the probabilities theory go back to the research of George Boole in the last century [1], and have been discussed frequently in the last decades [2, 3]. The reason for this interest is the potential generality of this unifying language, with potential applications in knowledge representation, objects description in group technology, expert systems for diagnostics, learning from rules, information search and recovery, description of resources distributed in a network [4] and planning under uncertainty.

Nowadays there as two main approaches for the combination of logic and probabilities. The first one basically associates probabilities to general sentences expressed in some logic. The second (and more recent) one starts by restricting the language and assuming independence relations so as to guarantee uniqueness in probabilistic assessments. These two approaches are, in many aspects, complementary. Probabilistic logic offers big flexibility, but typically dispense the representation of independence relations; on the other hand, the "relational probabilistic model" uses independence relations, but offers limited flexibility.

1.2 Background

Propositional logic, given its relatively simple sintax and semantics, offers an useful starting point for knowledge representation [5]. The basic syntatic element in this logic is the concept of *propositional variable* or *atomic formula* formula that can assume one of two values, *true* or *false*. A *literal* is either an atomic formula or its negation. In this text compound formulas are indicated by the greek letters ϕ, ψ and θ with or without indexes. A disjunction of literals is a *clause*. A *Conjunctive Normal Form (CNF)* is a conjunction of clauses, denoted by C1 $\wedge$ . . . Cr where Ci is a clause.

The semantics of any expression in propositional logic depends on a mapping that establishes a correspondence between the variables in the formula to facts in a target domain. A *truth assignment* is a vector assigning either value true or false to each propositional variable of an expression (these assignments are often called *possible worlds* [3]). If we have n propositional variables, there are 2 n truth assignments. A conjunctive formula is true if all its component formulas are true, otherwise it is false. A disjunctive formula is false only if all its component formulas are false, otherwise it is true. A negative formula is true (false) if its component formula is false (true).

A formula ϕ is *satisfiable* if it is true in some possible world ω; then ω is a *model* for ϕ ($\mathcal{M}(\phi, \omega)$). If ϕ has no model it is unsatisfiable. A formula ψ *entails* a formula θ ($\psi \models \theta$) if every model for ψ is a model for θ.

An *inference* $\psi \vdash \theta$ determines whether a premise ψ entails a conclusion θ. *satisfiability problem (SAT)* is: given a CNF ψ with m clauses C1 , . . . Cm , is ψ satisfiable? This question has a strong relationship to logic entailment and logic inference because $\psi \models \theta$ iff $\psi \wedge \neg\theta$ is unsatisfiable. That is, it is possible to approach inference as SAT problem. A particular kind of SAT problem is the *k-SAT*, a SAT which every clause has k literals.

An important limitation of propositional logic, from a point of view of knowledge representation, is its inability to deal with uncertainty. As stressed by Neapolitan [6]: "We also must acknowledge that in some cases the truth of certain premisses may be suggestive of the truth of a conclusion, but not imply it conclusively." To overcome this difficulty, propositional probabilistic logic extends propositional logic by attaching probability assessments to formulas. In this context,

The counterpart of SAT in probabilistic logic is the probabilistic satisfiability problem (*PSAT*) [7]. The PSAT structure is similar to SAT but it poses the following question: is there a probability distribution satifying a set of m assessments that assign probability interval to $P(\phi_i)$ for a set of formulas $\{\phi_i\}_{i=1}^{m}$ over n propositions.

If the assessments are such that no probability distribution p over truth assignments can be specified, the assessments are *inconsistent*. The *consistency problem* of probabilistic satisfiability is: given a set of assessments, determine whether they are inconsistent or not. The *inference problem* of probabilistic satisfiability is: given a set of assessments and a formula ϕ, obtain the infimum of $P(\phi)$ — that is, the infimum value α such that the constraint $P(\phi) = \alpha$ and the assessments are consistent. The infimum is denoted by $\underline{P}(\phi)$ and called the *lower probability* of ϕ. This infimum is attained because probabilitisc assessments on the logical formulas and constraints $\sum_{\omega} P(\omega) = 1$, $P(\omega) \geq 0$ define a bounded polyhedron in the space of probability measures over truth assignments.

Like SAT, PSAT is a NP-complete [7].

1.3 PPL overview

The construction of a probabilistic logic knowledge base is not a simple task. Formulas must be inserted; assessments associated with them; consistency must be checked, and revisions must be made continuously.

Currently the only system that allows interactive development of a probabilistic logic base, to the best of our knowledge, is the Check Coherence (CkC) package. CkC is distributed for noncommercial use, for Windows platforms only, at `http://www.dipmat.unipg.it/~{}upkd/paid/software.html`. The package deals with PSAT and allows conditioning on events of zero probability, a possibility that for the time being we avoid in our software. CkC asks the user to enter each formula and assessment in a sequence of steps, using an graphical interface to guide the process. While the CkC package

is useful and very general in its operation, we find that the manipulation of formulas is excessively rigid and a bit difficult at times

We have thus decided to investigate a different strategy to edit probabilistic logic bases. Our idea was to start from a well known prototyping language, and add features to this language so that it can serve as a convenient, simple and easy-to-learn editor of probabilistic logic bases. We wanted to create a tool that could be easily extended by others; that could be freely distributed; and that could run in a variety of operating systems. After a comparative analysis of several prototyping languages currently available, we settled on the Python language (`http://www.python.org`), as it has a clean syntax, a free implementation and an associated development system.

In the PPL package, the user types in arbitrary propositional formulas, using an intuitive syntax (described in the system manual). The user interacts with the package using the friendly Python editor (the IDLE system), and the user can benefit from all Python facilities such as memory control and string processing. The package can call functions that translate formulas into CNF if so desired. The user can attach either probabilities or probability intervals to formulas, and check consistency at any point in time. To check consistency, the package executes calls to the consistency checker described in the section 7.1.

2 Downloading PPL

To get PPL, you have two options:

1. You can download the gzip/tar file PPL-1.0.tar.gz . You have to use the gunzip and tar utilities to obtain all the files.

2. You can download the zip file PPL-1.0.zip. You have to use one of the many utilities that read the zip format.

You can also download a manual (pdf or postscript).

Downloading and unpacking the PPL distribution should produce several directories and files:

- A `to_CNF` directory, the package responsable for making the conversion to the conjuntive normal form.

- A `geracol` directory, with the inference algorithm (for more information see section 7.1) used by PPL and two important files: `ppl_file.txt` and `saida.txt`. They are responsable for the comunication between PPL and Geracol (see section **??**).

- A `Readme` directory that contains this manual and miscellaneous information, such as list of changes and bugs

- PPL.py

Important: If you download PPL, we ask you to notify us (andreteves@gmail.com or danillo_couto@yahoo.com.br) with a small email message. This software is experimental and will be evolving soon as we test it and kill bugs. We would like to know who has it so that we can send messages indicating patches and new versions. Even if you do not want to receive messages, send us a message indicating that you have the software but you do not want any messages. Thanks.

3 Pre-requisites

Before unpacking PPL you must have the following enviroments already installed in your computer:

ANSI C

You must download an IDE that uses Mingw port of GCC (GNU Compiler Collection) as it's compiler. If you are using Windows we recommend Dev C++ 5 or newer, you can find it at `http://www.bloodshed.net/devcpp.html`.

This is necessesary because the inference algorithm used by PPL (section 7.1) is written in C and should be recompiled in order to adapt it to your computer.

Python

PPL was developed using Python 2.4. You can download it at `http://www.python.org`.

Python is compatible with UNIX, OS X, Windows and Macintosh OS X.

Because PPL do not have an user interface, it works embedded in any Python IDE. We recomend IDLE because it comes with almost all available Python distributions. If you don't have any IDE installed you should download it either.

Linear Programming tool

You can use GLPK (GNU Linear Programming Kit) or ILOG CPLEX.

GLPK is a set of routines written in ANSI C and organized in the form of a callable library. If you are using a Unix system You can download it at `http://www.gnu.org/software/glpk/glpk.html`.

If you are using Windows you can find it at `http://gnuwin32.sourceforge.net/packages/glpk.htm`

If you have acess to the commercial programm ILOG CPLEX it is preferable to use this software, especially because it has proved to be considerably faster than GLPK. You can find more information about it at `http://www.ilog.com/`

PPL presents full compatibility with ILOG CPLEX 10 and GLPK 4.9.

4 Installation

First off all you should install all the programs listed above (section 3), acording to your operational system.

Now, unpack the downloaded file in a folder of your choice.

Creating the environment variables

You will have to create two environment variable: `PYTHONPATH` and `PPLPATH`. The first one will be used by Python to identify a new module folder (it tells Python which folder to search for a specific module when the command `import nameofthemodule` is used) and the second one allows Python to identify where PPL is installed. Probably, if you are using Pyhthon for the first time this two variable will have the same content.

If you are using Linux, to create an environment variable that lives forever, update your `.bash_profile` file:

XXXPATH=/usr/local/XXX/bin export XXXPATH

To check if the variable was correctly created, use `env` in the terminal, this command will list all existent environment variables

If you are using Windows go to:

My Computer → Properties → Advanced → Environment variables

and create the new variable pointing to the desired folder.

Compiling Geracol

with GLPK

The next step is to compile the geracol algorithm, which is written in C. You can find in `PPLPATH/geracol` the folders `CPLEX` and `GLPK`. Among others files, in each folder you will find the source file of Geracol (the inference engine used by PPL - section 7.1) adapted to your favourite linear programming solver.

To compile it in Windows using Dev C++ 5, go to:

Tools → Compiler Options → Compiler → Add the following commands when calling compiler

and add the following line, or something similar(depending on where you have installed GLPK):

C:/GnuWin32/lib/glpk.lib

Go to:

Tools → Compiler Options → Directories → Libraries

and add the path (again, or something similar to that):

C:/GnuWin32/lib

Go to:

Tools → Compiler Options → Directories → C includes

and add the path:

C:/GnuWin32/include

Now you can copile it! I hope these steps will work for you!

Finally, to execute Geracol is necessary to have the file with the extension .dll in the same directory (`PPLPATH/geracol`). For example, glpk49.dll, where the number following the name of the file is related with the version of GLPK.

with CPLEX

To compile Geracol you must have the file `Makefile` in the same directory of the source file and check if the path for the ILOG CPLEX library corresponds to:

/usr/ilog/cplex100/include/ilcplex/cplex.h

Now you just have to type Makefile in the terminal, which will generate the executable Geracol.

To execute the programm it is needed to have the files for the parameters modification in the format PRM: `altparam.prm` and `normparam.prm` in the same directory of the executable geracol.

5 Using PPL

In the following sections we will present all commands you need to know in order to use the PPL package.

5.1 Starting off

First you have to import the module called PPL.py. To do that, in your Python IDE just type:

```
>>> import PPL
```

This will work only if your environment variable PYTHONPATH is correctly configured (see section 4).

5.2 Sintax

There is a specific sintax that must be followed in order to insert a sentence. In general, a sentence is a set of operators and args. The possible operators are:

- Null-ary (no args) op:

  A symbol, representing a variable or constant (e.g. 'a')

- Unary (1 arg) op:

  '∼','-', representing NOT, negation (e.g. '∼a')

- Binary (2 arg) op:

  '==>' or '>>', representing forward implication

  '<==' or '<<', representing backward implication

  '<=>' or '%', representing logical equality

  '=/=' or '∧', representing logical disequality (XOR)

- N-ary (0 or more args) op: '&', '|', representing conjunction and disjunction

Internally PPL converts your sentence to the conjunctive normal form (CNF) using the package to_CNF.

5.3 The function p

This is the function that allows the insertion of sentence and probabilities in order to create a knowledge base. For doing this follow the structure:

`PPL.p(`*`sentence, lowprob, upperprob`*`)`

Where *`sentence`* is the sentence you want to insert. The arguments *`lowprob, upperprob`* are, respectively, the lower and upper probabilities of the sentence.

Note that the last two arguments can be omitted, inserting, then, sentences without a specific probability interval. If only one was omitted, the function will interpret as if the lower and upper probabilities are equals.

5.4 Inserting sentences and probabilities

Using the sintax specified in the section 5.2, there two possible ways of inserting a sentence. You can set a variable with the desired sentence and then use this variable in the function `PPL.p`

```
>>> s1 = '∼(B|C)'
>>> PPL.p(s1, 0.3, 0.8)
```

Or you can enter the sentence using directly the function `PPL.p`:

```
>>> PPL.p('∼(B|C)', 0.3, 0.8)
```

5.5 Inference

PPL can check coherency and perform extensions (inferences) in a knowledge base. To do so, first you have to insert some assessments (section 5.4) and then, if they are coherent, PPL will computate the upper and lower probabilities for any adicional event.

After inserting the assessments, you have two choices: just check their coherency and/or extend them. If you choose the first option, the function to be used is the `PPL.checkCoherence`:

```
>>> PPL.checkCoherence()
```

The result of this interaction should be 'coherent!' or 'incoherent!'. In case you want to perform inference:

```
>>> s2 = 'A∧C'
>>> PPL.extension(s2)
```

Where `s2` is the adicional assessment that you want to calculate the upper and lower probabilities. If the assessments are coherent, the result of this interaction should be something like this:

```
Lower:  0.2
Upper:  0.6
```

When the function `extension` is used, PPL calls the function `checkCoherence` and if it returns 'coherent!', then the probabilities interval is calculated.

6 Examples

Here is an example of use of PPL:

```
>>> import PPL
>>> s1 = 'a <=> (b|c)'
>>> s1
'a <=> (b|c)'
>>> s2 = PPL.toCNF(s1)
>>> s2
'((~b | a) & (~c | a) & (b | c | ~a))'
>>> PPL.p(s1, 0.5)
>>> s3 = 'd | (e & f) | g'
>>> s3
'd | (e & f) | g'
>>> s4 = PPL.toCNF(s3)
>>> s4
'((e | d | g) & (f | d | g))'
>>> PPL.p(s3, 0.3, 0.8)
>>> PPL.checkCoherence()
Coherent!
```

In Figure 1 a similar example is executed with IDLE:

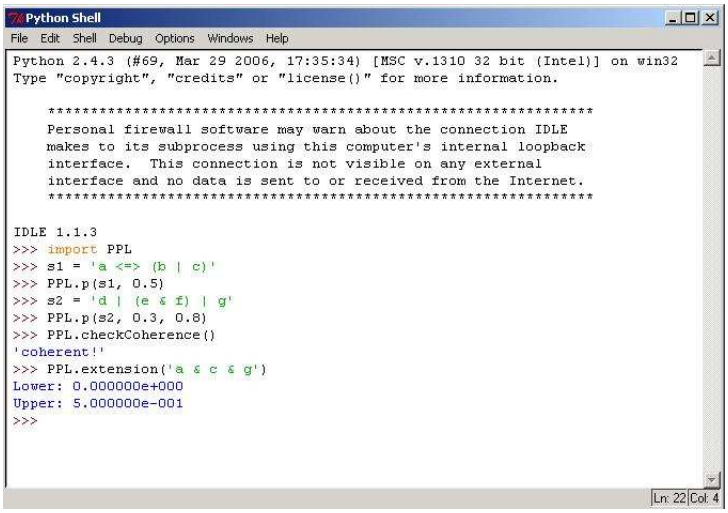

Figure 1: Example of using PPL in IDLE

7 The Consistency Checker

7.1 The algorithm

This algorithm was implemented by Paulo Sérgio de Souza Andrade in the Geracol package [8].

7.2 Comunication with PPL

The following sections present the formats of the files used to send informantion (inserted sentences, probabilities, etc) from PPL to Geracol and to receive the results from Geracol to PPL. It is important to notice that these files are internal to the system; as an user you do not need to use them nor edit them.

Figure 2 presents all components of the system:

Figure 2: Schematical diagram of PPL System

Format Manual: PPL → Geracol

This file is called `ppl_file.txt`.

Comments can appear in any part of the file but they must preceded by the symbol # at the beginning of the line.

The first line that is not a comment must contain the following constants, in order of appeareance:

`numvar numcla numcnf`

- `numvar`: number of distinct variables in the problem.
- `numcla`: number of clauses that appear in the following `numcla` lines, not including commentaries lines.
- `numcnf`: number of sentences, in the conjuctive normal form, with their probabilities interval. The last sentence is the one to be inferred.

The number 0 at the end of a line identifies the end of a clause.
A simple example:

```
# 5 variables, 3 clauses e 3 sentences
5 3 3
```

```
# Clauses group
1 3 4 0
-1 3 -5 0
-3 4 2 -1 0

# Senteces and probabilities gruops
# Probabilities interval for the conjuction
#of first and third clauses
0.1 0.3 1 3 0

# Exact probabilitie for the second clause
0.5 0.5 2 0

# Sentence to be inferred
0.0 0.0 1 2 0
```

Format: Geracol → PPL

The file is called `saida.txt`; itts structure is presented in the following lines with some explanation.

INICIO
HINI start of processing date and hour.
ARQ name of the file.
RSL S for a consistent problems, N for inconsistents and F for fail.
MIN lower probability value for the inferred sentence.
MAX upper probability value for the inferred sentence.
TMPC approximated total time of processing of the Restricted Master Problem(RMP).
TMPI approximated total time of processing of Subproblem (SP).
ITEC number of iterations to solve the RMP.
ITEI number of iterations to solve the SP.
ITGC number of iterations of the columm generator method.
ITEX total number of times that SP was solved with the standard parameters definitions.
HFIM end of processing date and hour.
FIM

A simple example:

```
INICIO
HINI Sab Aug 11 16:12:12 2007
ARQ ppl_file.txt
RSL S
```

```
MIN 2.000000e-001
MAX 5.000000e-001
TMPC 3.200000e-002
TMPI 1.560000e-001
ITGC 18
ITEX 2
HFIM Fri Feb 23 18:55:12 2007
FIM
```

8 Acknowledgements

PPL was created at Escola Politecnica da Universidade de São Paulo (`http://www.poli.usp.br`) from 2006 to 2007, with substantial support from CNPq (`http://www.cnpq.br`) during the first year of development and FAPESP (`http://www.fapesp.br`) during the second one.

We hope PPL works well and provides useful assistance and guidance.

Good luck!

André da Costa Teves
Danillo Paulo Couto

References

[1] G. Boole. The Laws of Thought. Dover edition, 1958.

[2] T. Hailperin. Sentential Probability Logic, Lehigh University Press, Bethlehem, 1996.

[3] N. J. Nilsson. Probabilistic logic. Artificial Intelligence, 28:71-87, 1986.

[4] T. Berners-Lee, J. Hendlers, O. Lassila. The Semantic Web, Scientific American, p. 34-43, 2001.

[5] Stuart J. Russel, P. Norvig. Artificial Intelligence: A Modern Aproach, p.281-284, 1995

[6] R. E. Neapolitan. Probabilistic Reasoning in Expert Systems. Prentice Hall, 1990

[7] G. Georgakapoulos, D. Kavvadias, C. Papadimitriou. Probabilistic satisfiability. Journal of Complexity, 4:111, 1988.

[8] P. S. de S. Andrade. Método de Geracão de Colunas aplicado ao problema de Satisfatibilidade Probabilistica, 2006.

[9] P. Hansen, B. Jaumard. Probabilistic Satisfiability, Technical Report, 1997