BET : An Inductive Logic Programming Workbench

Srihari Kalgi, Chirag Gosar, Prasad Gawde
Ganesh Ramakrishnan, Kekin Gada,
Chander Iyer, Kiran T.V.S, and Ashwin Srinivasan

Department of Computer Science and Engineering
IIT Bombay, India
`{cgosar,ganesh}@cse.iitb.ac.in`
`{chanderjayaraman,kekin.gada,prasg41,srihari.kalgi,t.kiran05}@gmail.com`
`ashwin.srinivasan@in.ibm.com`
`http://www.cse.iitb.ac.in/~bet`

Abstract. Existing ILP (Inductive Logic Programming) systems are implemented in different languages namely C, Progol, etc. Also, each system has its customized format for the input data. This makes it very tedious and time consuming on the part of a user to utilize such a system for experimental purposes as it demands a thorough understanding of that system and its input specification. In the spirit of Weka [1], we present a relational learning workbench called BET(**B**ackground + **E**xamples = **T**heories), implemented in Java. The objective of BET is to shorten the learning curve of users (including novices) and to facilitate speedy development of new relational learning systems as well as quick integration of existing ILP systems. The standardized input format makes it easier to experiment with different relational learning algorithms on a common dataset.

Key words: BET, ILP systems, Golem, Progol, FOIL, PRISM, TILDE

1 Introduction

There have been several Inductive Logic Programming (ILP) system implementations. Each system has its own specifications for input data. Different systems do not necessarily agree on their inputs. This often makes comparisons across different implementations tricky, owing to either a difference in names or semantics of parameters and/or a difference in the choice of the programming language. Very often, the primitives and core components employed, such as theorem provers, SAT solvers, inference engines, etc., are also different across different implementations, rendering the computation and accuracy comparisons less reliable. This paper discusses a Workbench for ILP called BET. BET is developed in Java and it standardizes the specification of input using XML (eXtensible Markup Language). It provides a common framework for “building” as well as “integrating” different ILP systems. The provision for implementing algorithms in a common language (Java) improves the feasibility of comparing algorithms on their computational speeds. Whereas, the input standardization enables sound comparison of accuracies (or related measures) and also allows for experimenting with multiple ILP systems on the same dataset without any input conversion required to the system specific format.

BET includes several evaluation functions and operator primitives such as Least General Generalization (LGG) [9], Upward cover, Downward cover, etc. These features facilitate the rapid development of new relational learning systems and also ease out the integration of existing relational systems into BET. BET also allows a user to choose from multiple theorem provers as plug and play components. Presently, YAP (Yet Another Prolog) [12] and SWI-Prolog [13] are included in the basic package. The initial version of BET has three relational systems implemented namely FOIL [4], Golem [3] and TILDE [2][10] and four relational systems integrated namely Progol [11], FOIL, Golem and PRISM [5][6] (Though PRISM is not a standard ILP system, it has been integrated with the specific intension of learning probabilities in order to associate uncertainity with theories learnt using ILP systems).

We proceed with an overview of BET and the motivation to develop such a system in Section 2. Section 3 focuses on the design of the BET system. Section 4 explains how new relational learning systems can be integrated/implemented in BET. Section 5 compares BET with existing workbenches and highlights the advantages of BET. We summarize BET in Section 6.

2 Overview

A principal motivation for developing a system like BET is the reduction of the learning curve for both expert-users and novices as well as programmers in the area of relational learning, particularly ILP. For example, with BET, the end-user will need to understand only the standardized input parameters for BET. This reduces the time overheads involved in comparing different algorithmic implementations as well as in experimenting with individual implementations. More specifically, a user need not convert datasets from one format to another while switching between ILP systems. Further, the standardized APIs in BET make development of algorithms within BET much easier. Figure 1 shows the block diagram of BET.

Any ILP system in BET takes four files as its input namely positive examples, negative examples, background knowledge and language restriction files, and it outputs in the form of *theories*. There are primarily two important components of BET namely the *BET GUI* and the *BET Engine*. The BET Engine is the back-end engine at the core of BET and the BET GUI communicates with the BET engine through the medium of a *properties* file. BET users can experiment with the system using the GUI whereas programmers can use the APIs provided and extend BET with new ILP or relational learning systems. Section 5 explains how BET could be extended.

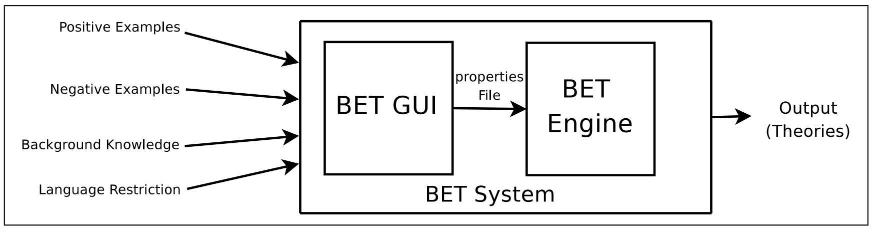

Fig. 1. Block diagram of BET

3 Design of BET

Almost all ILP algorithms as well as systems require following five inputs: positive examples, negative examples, background knowledge, mode declarations and hyper-parameters for tuning the system. The clauses learnt may be customized by means of the mode declarations, though some algorithms/systems may not allow for the specification of mode declarations (example is Confidence-based Concept Discovery system [7]). In this Section, we discuss the need for having a standard for the input, and eventually specify the BET standard input file formats.

3.1 Need for Standard Format

There is little or no consensus between the input specifications of ILP systems. For example, FOIL has its standard input specification which is completely different from that of Golem or Progol. FOIL accepts the positive examples, negative examples and background knowledge in a single file. On the other hand, Golem takes three files as input viz. positive example file ($.f$), negative example file ($.n$) and the background knowledge file ($.b$) which includes the mode declarations. Another ILP system Progol has an input format that is slightly similar to Golem, but has many more setting parameters.

Clearly any end user, desiring to experiment with a new ILP system, is required to understand its input format thoroughly, convert the data in hand to that format and only then can he/she use it. BET assists the end user with a standardized input specification that is generic enough and can be converted to the specification of any existing ILP system. Section 3.2 will discuss about BET standard formats.

3.2 BET Standard Format

As already stated, BET's inputs are comprised of four XML files namely positive examples, negative examples, background knowledge and language restriction file. Sample positive example, negative example, background knowledge and language restriction files for learning the *ancestor* relation are shown in *Table 1*, *Table 2*, and *Table 3* respectively in the document available at [15]. The hyper-parameters can be specified in Language Restriction file. The format for each file is as follows:

Positive Example File: The file consists of clauses which represents positive examples for the relation (predicate) for which the theory needs to be learned.

Negative Example File: This file consists of clauses which represents negative examples of the predicate (relation) for which the theory needs to be learned. There may be some ILP systems which do not consider any negative examples during construction of theory, but may utilize negative examples for pruning the theories that have been learned.

Background knowledge File: The background knowledge (BGK) file contains all the *domain specific information* which is required by the ILP system in order to learn the target predicate. All this information should be in the form of clauses. It can also contain rules instead of just facts.

Language Restriction File: The structured search space (lattice or simply a partial order) of clauses which needs to be searched in order to arrive at a reasonably good hypothesis is almost infinite. *Language Restrictions* endow an ILP system with the flexibility of reducing the search space. There are three generic parts to the Language Restriction viz. Mode declaration, Type and Determination. Also we can specify the system-specific hyper-parameters like stopping threshold, minimum accuracy, etc. in this file only. Each of them, is explained below.

Type: Type defines the type of constants. For example, the constant `john` is of type `person`, constant `1` is of type `integer`. Type specification can be done in the language restriction file.

Mode Declarations: Mode declarations declare the mode of the predicates that can be present in any of clausal theories. The mode declaration takes the form `mode(RecallNumber,PredicateMode)`, where
RecallNumber: bounds the non-determinacy of a form of predicate call.
PredicateMode: has the following syntax

```
predicate(ModeType,ModeType,...)
predicate(<+/-/#>Type,<+/-/#>Type,...)
```

+Type : the term must be an input variable of type Type.
-Type : the term must be an output variable of type Type.
#Type : the term must be a ground constant of type Type.

Determination: This component of mode declarations provides information regarding which predicates need to be learned in terms of which other predicates.

Hyper-parameters: Hyper-parameters are very much system specific and we can find the different parameters for a system by doing *system-name –help* or through the graphical interface for that particular system.

4 Class Diagram and Components of BET

The class diagram of BET is shown in Figure 1 in the document available at [15]. The major components of BET are explained in subsequent sections.

ClauseSet: ClauseSet refers to a set of clauses, which is often referred to in the description of ILP algorithms. The ClauseSet interface defines the signature of the methods used to access the ClauseSet object.

LanguageRestriction: LanguageRestriction is an interface for mode declarations, determinations and types.

InputEncoder: InputEncoder is mainly used for integrating legacy systems (i.e. ILP systems written in native code) into BET. Any wrapper around an integrated legacy system will require conversion of the BET standard data format to the format accepted by the legacy system. Any legacy system which requires to be integrated into BET has to implement `InputEncoder` interface and provide a method of converting the standard BET file format to the format accepted by the legacy system.

TheoryLearner: TheoryLearner is an important component of BET. It is responsible for learning the theory (hypothesis) from the input. `TheoryLearner` has objects of type `InputEncoder` and `EvalMetric` (explained next) as its data members. In case the ILP system is completely implemented in Java (i.e., if it is not a legacy system), it can use a dummy implementation of the `InputEncoder` interface and pass the object to the `TheoryLearner` class. The dummy implementation of `InputEncoder` interface is mainly used to access the BET input files.

EvalMetric (Evaluation Metric): Any ILP algorithm will try to explore the structured space (subsumption lattice or partial order) of clauses, by traversing the graph using refinement operators. At every point in the space, the algorithm has to decide whether a clause in the space is good enough to be present in the hypothesis. For this purpose, the covers relation is employed.

If the covers relation is defined by `Entailment` ($\models$), then the ILP system is said to learn from entailment [9]. Some other ILP systems learn from proof traces [9], while few others learn from interpretations [9], etc.

Any implementation of EvalMetric will specify the *covers* relation, *i.e.*, definition of an example being covered by background knowledge (with candidate hypothesis).

TheoremProver: The theorem prover is the backbone of most ILP systems. BET has two different theorem provers namely YAP and SWI-Prolog. YAP is perhaps the fastest theorem prover as of today. YAP is built in C and doesn't have any JNI (Java Native Interface), so the YAP process is run by spawning a process through a BET program. SWI-Prolog is also built in C, but it does support JNI (Java Native Interface). SWI-Prolog has a standard interface called JPL which comes bundled with SWI-Prolog itself.

5 Support for Building ILP Systems in BET

It is very easy to integrate[1] an existing system into BET. Also the implementation[2] of a new system within BET is faster than implementing it from scratch. The following section briefs on how to integrate and implement a system in BET.

5.1 Integrating a Legacy System in BET

Legacy systems can be integrated into BET as follows:

– Implement the interface `InputEncoder`, which should provide a functionality to convert files from BET standard format to the format accepted by the legacy system.
– Extend the `TheoryLearner` class and override the method `learnTheory`.

5.2 Implementing a New ILP System in BET

– A dummy implementation of the interface `InputEncoder` is required for this purpose, since the `TheoryLearner` class expects an object of `InputEncoder` while creating its object.
– Extend the `TheoryLearner` class and override the method `learnTheory`.
– Different types of Evaluation Metric, Theorem provers (YAP, SWI-Prolog) can be used in implementing the ILP system.

6 Related Work

Aleph and GILPS (Generic Inductive Logic Programming System) are two earlier efforts put in to develop a workbench like BET for Inductive Logic Programming. Aleph [8] is developed by Ashwin Srinivasan, whereas the GILPS [14] workbench is developed by Jose Santos.

[1] The authors of this paper took two weeks to integrate PRISM in BET and one week individually for integrating Golem and FOIL

[2] The authors of this paper implemented TILDE, FOIL and Golem in BET within a week for each algorithm

6.1 Aleph

Aleph is written in Prolog principally for use with Prolog compilers like Yap and SWI-Prolog compiler. Aleph offers a number of parameters to experiment with its basic algorithm. These parameters can be used to choose from different search strategies, various evaluation functions, user-defined constraints and cost specifications. Aleph also allows user-defined refinement operators.

6.2 GILPS

GILPS implements three relational systems namely TopLog, FuncLog and ProGolem. GILPS requires at least YAP 6.0 for its execution. Many of its user predicates and settings are shared with Aleph.

6.3 Advantages of BET over Existing Systems

BET offers following advantages over existing systems (Aleph, GILPS ,etc.):

- BET is more likely to find use in the larger applied machine learning and Mining communities who are less familiar with prolog and where it is important to interface learning tools in Java (such as BET) with applications largely written in procedural languages (very often Java). This can also make diagnosis and debugging easy. As against this, GILPS and Aleph assume familiarity with Prolog and cannot be easily invoked from applications written in Java.
- In the spirit of Weka, BET provides a nice framework for the integration of legacy systems as well as for the implementation of new systems. In fact, while the basic BET framework was provided by the first two co-authors, the remaining co-authors used that framework to implement several ILP algorithms. This is something that Aleph and GILPS do not cater to as well.
- BET allows saving the learned models as serialized files which can later be used for finding models for new problems.
- BET provides a choice of theorem prover where as GILPS can work only with YAP.
- BET comes with a YAP installer and all the "integrated" systems.
- BET has classes to convert the background knowledge, positive examples and negative examples files into XML format. Language restriction file needs little human intervention for this purpose.
- BET provides a standard input/output format which enables end-user to experiment with multiple relational systems with the same datasets. Aleph uses three different files namely (.f), (.n) and (.b) whereas GILPS takes input as prolog files (.pl).

7 Summing up BET

The standardized input format of BET makes it more convenient for user to have only one input format for all relational systems. The ability of BET to write learned classification/regression models as serialized files allows for saving the model and reusing the model at a later stage for classification or regression on new problems. The BET graphical user interface makes experimentation with implemented algorithms and integrated systems much more easier. Use of JAVA as an implementation language makes BET extensible and platform independent. The only visible disadvantage of BET is that the JAVA implementation of relational systems are slower as compared to their C/C++/Progol counterparts, but we chose Java over C++ owing to ease of extending and platform independence of the former over the latter.

References

1. E. Frank and M. A. Hall and G. Holmes and R. Kirkby and B. Pfahringer and I. H. Witten: "Weka- A machine learning workbench for data mining." In : Data Mining and Knowledge Discovery Handbook- A Complete Guide for Practitioners and Researchers, pp. 1305–1314. Springer, Berlin (2005)
2. Blockeel, Hendrik and De Raedt, Luc: "Top-down induction of first-order logical decision trees." In: Artificial Intelligence Journal (1998), pp. 285–297. Elsevier Science Publishers Ltd.
3. S.H. Muggleton and C. Feng: "Efficient induction of logic programs.": Proceedings of the First Conference on Algorithmic Learning Theory, pp. 368–381. Ohmsha, Tokyo (1990)
4. J. Ross Quinlan and R. Mike Cameron-Jones: "FOIL: A Midterm Report." : Machine Learning: ECML-93, European Conference on Machine Learning, Proceedings, vol. 667 pp. 3–20. Springer-Verlag (1993)
5. Taisuke Sato, Yoshitaka Kameya, Neng-Fa Zhou: "Generative Modeling with Failure in PRISM.": IJCAI, pp. 847–852 (2005)
6. Taisuke Sato and Yoshitaka Kameya: "PRISM: A Language for Symbolic-Statistical Modeling." : IJCAI, pp. 1330–1339 (1997)
7. Y. Kavurucu, P. Senkul and I.H. Toroslu: "ILP-based concept discovery in multi-relational data mining." : Expert Systems with Applications, vol. 36 pp. 11418 - 11428 (2009)
8. "Aleph Manual", http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/aleph.html
9. "Statistical Relational Learning Notes", http://www.cse.iitb.ac.in/~cs717/notes/
10. "TILDE: Top-down Induction of Logical Decision Trees", http://www-ai.ijs.si/~ilpnet2/systems/tilde.html
11. "Progol", http://www.doc.ic.ac.uk/~shm/progol.html
12. "YAP Manual", http://www.dcc.fc.up.pt/~vsc/Yap/
13. "SWI-Prolog", http://www.swi-prolog.org/
14. "General Inductive Logic Programming System", http://www.doc.ic.ac.uk/~jcs06/GILPS/
15. "ILP Systems Integrated and Implemented in BET and Sample Examples",http://www.cse.iitb.ac.in/~bet/appendix.pdf