City Research Online

City St George's, University of London

Citation: França, M. V. M., Zaverucha, G. & Garcez, A. (2014). Fast relational learning using bottom clause propositionalization with artificial neural networks. Machine Learning, 94(1), pp. 81-104. doi: 10.1007/s10994-013-5392-1

This is the unspecified version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/3057/

Link to published version: https://doi.org/10.1007/s10994-013-5392-1

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

Fast Relational Learning using Bottom Clause Propositionalization with Artificial Neural Networks

Manoel V. M. França
Gerson Zaverucha
Artur S. d'Avila Garcez

July 15, 2013

Abstract

Relational learning can be described as the task of learning first-order logic rules from examples. It has enabled a number of new machine learning applications, e.g. graph mining and link analysis. Inductive Logic Programming (ILP) performs relational learning either directly by manipulating first-order rules or through propositionalization, which translates the relational task into an attribute-value learning task by representing subsets of relations as features. In this paper, we introduce a fast method and system for relational learning based on a novel propositionalization called Bottom Clause Propositionalization (BCP). Bottom clauses are boundaries in the hypothesis search space used by ILP systems Progol and Aleph. Bottom clauses carry semantic meaning and can be mapped directly onto numerical vectors, simplifying the feature extraction process. We have integrated BCP with a well-known neural-symbolic system, C-IL^{2}P, to perform learning from numerical vectors. C-IL^{2}P uses background knowledge in the form of propositional logic programs to build a neural network. The integrated system, which we call CILP++, handles first-order logic knowledge and is available for download from Sourceforge. We have evaluated CILP++ on seven ILP datasets, comparing results with Aleph and a well-known propositionalization method, RSD. The results show that CILP++ can achieve accuracy comparable to Aleph, while being generally faster, BCP achieved statistically significant improvement in accuracy in comparison with RSD when running with a neural network, but BCP and RSD perform similarly when running with C4.5. We have also extended CILP++ to include a statistical feature selection method, mRMR, with preliminary results indicating that a reduction of more than 90% of features can be achieved with a small loss of accuracy.

Keywords: Relational Learning, Inductive Logic Programming, Propositionalization, Neural-Symbolic Integration, Artificial Neural Networks

1 Introduction

Relational learning can be described as the task of learning a first-order logic theory from examples (Džeroski and Lavrač, 2001; De Raedt, 2008). Differently from propositional learning, relational learning does not use a set of attributes and values. Instead, it is based on objects and relations among objects, which are represented by constants and predicates, respectively. This enables a range of applications of machine learning, for example in Bioinformatics, graph mining and link analysis, serious games, etc. (Bain and Muggleton, 1994; Srinivasan and Muggleton, 1994; Džeroski and Lavrač, 2001; King and Srinivasan, 1995; King et al., 2004;

Muggleton et al., 2010). Inductive Logic Programming (ILP) (Muggleton and Raedt, 1994; Nienhuys-Cheng and de Wolf, 1997) performs relational learning either directly by manipulating first-order clauses or through a method called propositionalization (Lavrač and Džeroski, 1994; Železný and Lavrač, 2006), which brings the relational task down to the propositional level by representing subsets of relations as features that can then be used as attributes. In comparison with full ILP, propositionalization normally exchanges accuracy for efficiency (Krogel et al., 2003), as it enables the use of fast attribute-value learners such as decision trees or even neural networks (Quinlan, 1993; Rumelhart et al., 1994), but could lose information in the translation of first-order clauses into features.

In this paper, we introduce a fast system for relational learning based on a new form of propositionalization, which we call Bottom Clause Propositionalization (BCP). Bottom clauses are boundaries on the hypothesis search space, first introduced by Muggleton (1995) as part of the Progol system, and are built from one random positive example, background knowledge (a set of clauses that describe what is known) and language bias (a set of clauses that define how clauses can be built). A bottom clause is the most specific clause (with most literals) that can be considered as a candidate hypothesis. BCP uses bottom clauses for propositionalization because they carry semantic meaning, and because bottom clause literals can be used directly as features in a truth-table, simplifying the feature extraction process (Muggleton and Tamaddoni-Nezhad, 2008; DiMaio and Shavlik, 2004; Pitangui and Zaverucha, 2012).

The idea of using BCP for learning came from our attempts to represent and learn first-order logic in neural networks (Garcez and Zaverucha, 2012). Neural networks (Rumelhart et al., 1994) are attribute-value learners based on gradient-descent. Learning in neural networks is achieved by performing small changes to a set of weights, in contrast with ILP, which performs learning at the concept level. Neural networks' distributed architecture is generally accredited as a reason for robustness; neural networks seem to perform well in continuous domains and when learning from noisy data (Rumelhart et al., 1994). Systems that combine symbolic computation with neural networks are called neural-symbolic systems (Garcez et al., 2002). In neural-symbolic integration, the representation of first-order logic by neural networks is of interest in its own right, since first-order logic learning and reasoning using connectionist systems remains an open research question (Garcez et al., 2008). As a result, we investigate whether neural-symbolic learning is a good match for BCP. The experiments reported below indicate that this is indeed the case, in comparison with standard ILP and a well-known propositionalization method.

The neural-symbolic system C-IL^{2}P has been shown effective at learning and reasoning from propositional data in a number of domains (Garcez and Zaverucha, 1999). C-IL^{2}P uses background knowledge in the form of propositional logic programs to build a neural network, which is in turn trained by examples using backpropagation (Rumelhart et al., 1994). We intend to achieve the same in relational domains. Hence, we have extended C-IL^{2}P to handle first-order logic by using BCP to train a *first-order* neural network. The extended system, which we call CILP++, has been implemented in C++ and is available to download from *Sourceforge* at `https://sourceforge.net/projects/cilppp/` (the experiments reported in this paper can be reproduced by downloading the datasets and list of parameters from `http://soi.city.ac.uk/~abdz937/bcexperiments.zip`). CILP++ incorporates BCP as a novel propositionalization method and differently from C-IL^{2}P, CILP++ networks are first-order in that each neuron denotes a first-order atom. Yet, CILP++ learning uses the same neural model as C-IL^{2}P, by transforming each first order example into a bottom clause. Experimental evaluations reported in this paper show that such a combination can lead to efficient learning of relational concepts. Given our experimental results, which are

summarised in the next paragraph, our long-term goal is to apply and evaluate BCP on other general settings, including discrete and continuous data, noisy environments with missing values, and problems containing errors in the background knowledge.

We have compared CILP++ with Aleph (Srinivasan, 2007) – a state-of-the-art ILP system based on Progol – and compared BCP with a well-known propositionalization method, RSD (Železný and Lavrač, 2006), using neural networks and the C4.5 decision tree learner (Quinlan, 1993), on a number of benchmarks: four *Alzheimer's* datasets (King and Srinivasan, 1995) and the *Mutagenesis* (Srinivasan and Muggleton, 1994), *KRK* (Bain and Muggleton, 1994) and *UW-CSE* (Richardson and Domingos, 2006) datasets. Several aspects were empirically evaluated: standard classification accuracy using cross-validation and runtime measurements, how BCP performs in comparison with RSD, and how CILP++ performs in different settings using feature selection (Guyon and Elisseeff, 2003). The CILP++ implementation has not been optimised for performance. We evaluated six different configurations of CILP++ in order to explore some of the capabilities of the approach: three versions of CILP++ trained with standard backpropagation, each one using three sizes of background knowledge, and three versions of CILP++ trained with *early stopping* (Prechelt, 1997), with the same three background knowledge sizes used for standard backpropagation. In the first set of experiments – accuracy vs. runtime – CILP++ achieved results comparable to Aleph and performed faster on most datasets.

Regarding the performance of BCP against RSD, BCP achieved a statistically significant improvement in accuracy in comparison with RSD when running with a neural network, but BCP and RSD have shown similar performance when running with C4.5. Nevertheless, BCP was faster than RSD in all cases. Since bottom clauses may have a large number of literals (Muggleton, 1995), BCP might generate a large number of features. Hence, we evaluated accuracy also using feature selection, as follows. CILP++ was extended to include a statistical feature selection method called mRMR which is widely used for visual recognition and audio analysis (Ding and Peng, 2005). We applied three-fold cross validation on training data to choose two models and used those on two *Alzheimer* datasets. The results indicate the existence of an optimal variable-depth parameter for generating bottom clauses and that "more is not merrier". In one CILP++ model, mRMR managed to reduce over 90% of features while having a loss of less than 2% on accuracy on two *Alzheimer* testbeds, although an increase in runtime was observed. Further experiments in different application domains and comparison with other propositionalization methods, e.g. Kuželka and Železný (2011), are under way.

Related Work: Approaches related to CILP++ can be grouped into three categories: approaches that also use bottom clauses, other propositionalization methods, and other relational learning methods. In the first category, DiMaio and Shavlik (2004) use bottom clauses with neural networks to build an efficient hypothesis evaluator for ILP. Instead, CILP++ uses bottom clauses to classify first-order examples. The QG/GA system (Muggleton and Tamaddoni-Nezhad, 2008) introduces a new hypothesis search algorithm for ILP, called Quick Generalization (QG), which performs random single reductions in bottom clauses to generate candidate clauses for hypothesis. Additionally, QG/GA proposes the use of Genetic Algorithms (GA) on those candidate clauses to further explore the search space, converting the clauses into numerical patterns. CILP++ does the same, but for use with neural networks instead of discrete GAs.

In the second category – other propositionalizations – LINUS (Lavrač and Džeroski, 1994) was the first system to introduce propositionalization. It worked with acyclic and function-free Horn Clauses, like Progol and BCP, but differently from BCP, it constrained the first-order language further to only accept clauses where all body variables also appear in the head. Its

successor, DINUS (Kramer et al., 2001), allows a larger subset of clauses to be accepted (determinate clauses), allowing clauses with body variables that do not appear in the head literal, but still allowing only one possible instantiation of those variables. Finally, SINUS (Kramer et al., 2001) improved on DINUS by allowing unconstrained clauses, making use of language bias in the feature selection and verifying if it is possible to unify newly found literals with existing ones, while keeping consistency between pre-existing variable namings, thus reducing the final number of features. LINUS and DINUS treat body literals as features, which is similar to BCP. However, BCP can deal with the same language as Progol, thus having none of the language restrictions of LINUS/DINUS. SINUS, on the other hand, propositionalizes similarly to another method, RSD, which is compared to this work and is explained separately in Section 2.3. RSD also has a recent successor, called RelF (Kuželka and Železný, 2011), which takes a more classification-driven approach than RSD by only considering features that are interesting for distinguishing between classes (it also discards features that θ-*subsume* any previously-generated feature). Comparisons with RelF are under way.

Finally, in the third category, we place the body of work on statistical relational learning (Getoor and Taskar, 2007; De Raedt et al., 2008), that albeit relevant for comparison, is less directly related to this work, e.g. Markov Logic Networks (MLN) (Richardson and Domingos, 2006) and other systems combining relational and probabilistic graphical models (Koller and Friedman, 2009; Paes et al., 2005), neural-symbolic systems for learning from first-order data in neural networks such as Basilio et al. (2001), Kijsirikul and Lerdlamnaochai (2005) and Guillame-Bert et al. (2010), and systems that propose to integrate neural networks and first-order logic through relational databases, e.g. Uwents et al. (2011). Those systems differ from CILP++ mainly in that they seek to embed relational data directly into the networks' structures, which is a difficult task. In contrast, CILP++ seeks to benefit from using a simple network structure as an attribute-value learner, following a propositionalization approach, as discussed earlier.

Summarizing, the contribution of this paper is two-fold. The paper introduces: (i) *a novel propositionalization method, BCP*, which converts first-order examples into propositional patterns by generating their bottom clauses, treating each body literal as a propositional feature, and (ii) *the successor of C-IL^{2}P, the CILP++ system*, which reduces C-IL^{2}P's learning times and system complexity, uses a new weight normalization, maintaining the integrity of first-order background knowledge, and is easily configurable to be used with any ILP dataset. CILP++ takes advantage of mode declarations and determinations to generate consistent bottom clauses which share variable namings, thus being applicable to any dataset that first-order systems Aleph or Progol are applicable. CILP++ may use C-IL^{2}P's knowledge extraction algorithm (Garcez et al., 2001) so that interpretable first-order rules can be obtained from the trained network. Currently, first-order rules can be obtained when BCP is used together with C4.5 (since each node represents a first-order literal). This is further discussed in the body of the paper.

The remainder of the paper is organized as follows. In Section 2, we introduce the ILP concepts used throughout the paper: propositionalization, neural networks and neural-symbolic systems. In Section 3, we show how CILP++ builds a neural network from bottom clauses and how the network can be trained using bottom clauses as examples with backpropagation. We also analyze two stopping criteria: standard training error minimization and early stopping, and discuss how knowledge extraction can be carried out. In Section 4, we report and discuss all experimental results, and in Section 5, we conclude and discuss directions for future work.

2 Background

In this section, both machine learning subfields that are directly related to this work (Inductive Logic Programming and Artificial Neural Networks) are reviewed. This section also introduces notation used throughout the paper. An introduction to C-IL^{2}P is also presented, followed by a review of propositionalization and feature selection.

2.1 ILP and Bottom Clause

Inductive Logic Programming (Muggleton and Raedt, 1994) is an area of machine learning that makes use of logical languages to induce theory-structured hypotheses. Given a set of labeled examples E and background knowledge B, an ILP system seeks to find a hypothesis H that minimizes a specified loss function. More precisely, an ILP task is defined as <E, B, L>, where $E = E^+ \cup E^-$ is a set of positive (E^+) and negative (E^-) clauses, called *examples*, B is a logic program called *background knowledge*, which is composed by *facts* and *rules*, and L is a set of logic theories called *language bias*.

The set of all possible hypotheses for a given task, which we call S_H, can be infinite (Muggleton, 1995). One of the features that constrains S_H in ILP is the language bias, L. It is usually composed by specification predicates, which define how the search is done and how far it can go. The most common specification language is called *mode declarations*, composed of: *modeh* predicates, that define what can appear as head of a clause; *modeb* predicates, that define what predicates can appear in the body of a clause; and *determination* predicates, which relate body and head literals. The *modeb* and *modeh* declarations also specify what is considered to be an *input variable*, an *output variable*, a *constant*, and an upper bound on how many times the predicate it specifies can appear in the same clause, called *recall*. The language bias L, through *mode declarations* and *determination* predicates, can restrict S_H during hypothesis search to only allow a smaller set of candidate hypotheses H_c to be searched. Formally, H_c is a candidate hypothesis for a given ILP task <E, B, L> iff $H_c \in L$, $B \cup H_c \models E^+ \cup N$ and $B \cup H_c \not\models E^- - N$, where $N \subseteq E^-$ is an allowed *noise* in H_c in order to ameliorate overfitting issues.

Something else that is used to restrict hypothesis search space in algorithms based on inverse entailment, such as Progol, is the *most specific (saturated) clause*, $\perp_e$. Given an example e, Progol firstly generates a clause that represents e in the most specific way as possible, by searching in L for *modeh* declarations that can unify with e and if it finds one, an initial $\perp_e$ is created. Then it passes through the *determination* predicates to verify which of the bodies specified among the *modeb* clauses can be added to $\perp_e$, repeatedly, until a number of cycles (known as *variable depth*) through the *modeb* declarations has been reached.

2.2 Artificial Neural Networks and C-IL^{2}P

An artificial neural network (ANN) is a directed graph with the following structure: a unit (or neuron) in the graph is characterized, at time t, by its *input vector* $I_i(t)$, its *input potential* $U_i(t)$, its *activation state* $A_i(t)$, and its *output* $O_i(t)$. The units of the network are interconnected via a set of directed and weighted connections such that if there is a connection from unit i to unit j then $W_{ji} \in \mathbb{R}$ denotes the *weight* of this connection. The input potential of neuron i at time t ($U_i(t)$) is obtained by computing a weighted sum for neuron i such that $U_i(t) = \sum_j W_{ij} I_i(t)$. The activation state $A_i(t)$ of neuron i at time t is then given by the neuron's *activation function* h_i such that $A_i(t) = h_i(U_i(t))$. In addition, b_i (an extra weight with input

always fixed at 1) is known as the *bias* of neuron i. We say that neuron i is *active* at time t if $A_i(t) > -b_i$. Finally, the neuron's output value is given by its activation state $A_i(t)$.

For learning, *backpropagation* (Rumelhart et al., 1994) is the most widely used algorithm, based on gradient descent. It aims to minimize an error function **E** regarding the difference between the network's answer and the example's actual classification. *Standard training* and *early stopping* (Haykin, 2009) are two commonly used stopping criteria for backpropagation training. In *standard training*, the full training dataset is used to minimize **E**, while *early stopping* uses a validation set to measure data overfitting: training stops when the validation set error starts to increase. When this happens, the best validation configuration obtained thus far is used as the learned model.

The Connectionist Inductive Learning and Logic Programming system, C-IL^{2}P (Garcez and Zaverucha, 1999), is a neural-symbolic system that builds a recursive ANN using background knowledge composed of propositional clauses (building phase). C-IL^{2}P also learns from examples using backpropagation (training phase), performs inference on unknown data by querying the post-training ANN, and extracts a revised knowledge from the trained network (Garcez et al., 2001) to obtain a new propositional theory (extraction phase). Fig. 1 illustrates the building phase and shows how to build a recursive ANN **N** from background knowledge **BK**[1].

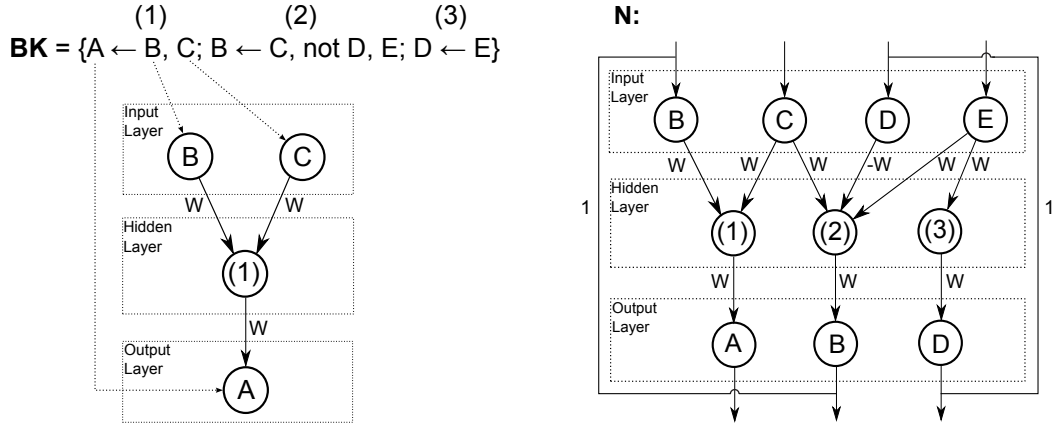

Figure 1: C-IL^{2}P building phase example. Starting from background knowledge **BK**, C-IL^{2}P creates an ANN by creating a hidden layer neuron for each clause in **BK**. Thus, the C-IL^{2}P network for **BK** has three hidden neurons. Then, each body literal in **BK** is associated with an input neuron, and each head literal in **BK** is associated with an output neuron. For example, for the first clause (A ← B, C), since it has two body literals B and C, two input neurons are created and are connected to a hidden neuron (1), corresponding to the clause, with positive weight W. If a literal is negated (for example, literal *not D* in the second clause of **BK**), its corresponding neuron is connected to the hidden neuron using weight $-W$. Hidden neuron (1) is then connected to the output neuron corresponding to A using connection weight W. Finally, input and output neurons that share the same label are recursively connected from the output to the input of the network with weight 1, so that output values can be propagated back to the input in the next calculation of rule chaining. For example, the head of the second clause in **BK** (B) is also one of the body literals of the first clause. Therefore, a recursive connection between the output neuron representing B and the input neuron representing B is created. The resulting network **N** encodes and can compute **BK** in parallel, as well as be trained from examples having **BK** as background knowledge.

C-IL^{2}P calculates weight and bias values for all neurons. The value of W is constrained by Eq. 2, the values of the biases of the input layer neurons are set as 0, and the biases of the hidden layer neurons n_h (b_{n_h}) and of the output layer neurons n_o (b_{n_o}) are given by Eqs. 3 and 4 below, respectively. Both W and the biases are functions of A_{min}: this parameter controls the activation of each neuron n in C-IL^{2}P by only allowing activation if the condition shown in Eq. 1 is satisfied, where w_i is a network weight that ends in n, x_i is an input and h_n is the activation function of neuron n, which is *linear* if it is an input neuron and *semi-linear bipolar*[2] if it is not. W and the biases values are set so that the network implements an AND-OR structure with hidden neurons implementing a logical-AND of input neurons, and output neurons implementing a logical-OR of hidden neurons, so that the network can be used to run the logic program. To exemplify the network computation, given **BK**, if E and C are set to true, and D is set to false in the network (i.e. neurons E and C are activated while neuron D is not), a feedforward propagation activates output neuron B (because of the second clause in **BK**). Then, a recursive connection carries this activation to input neuron B, and a second feedforward propagation would activate A. This process continues until a stable state is reached, when no change in activation is seen after a feedforward propagation.

$$h_n(\sum_{\forall i} w_i \cdot x_i + b) \geq A_{min} \tag{1}$$

$$W \geq \frac{2}{\beta} \cdot \frac{\ln(1+A_{min}) - \ln(1-A_{min})}{max(k_n, \mu_n) \cdot (A_{min}-1) + A_{min} + 1} \tag{2}$$

$$b_{n_h} = \frac{(1+A_{min})(k_{n_h}-1)}{2} \cdot W \tag{3}$$

$$b_{n_o} = \frac{(1+A_{min})(1-\mu_{n_o})}{2} \cdot W \tag{4}$$

In Eqs. 2, 3 and 4: k_{n_h} is the number of body literals in the clause corresponding to the hidden neuron n_h (i.e., the number of connections coming from the input layer to n_h); μ_{n_o} is the number of clauses in the background knowledge with the same head as the head literal mapped by the output neuron n_o (i.e., the number of connections coming from the hidden layer to n_o); $max(k_n, \mu_n)$ is the maximum value among all k and all μ, for all neurons n; and β is the *semi-linear bipolar* activation function slope.

After the building phase, training can take place. Optionally, more hidden neurons can be added (if this is needed in order to better approximate the training data), and the network is fully connected with near-zero weighted connections. The training algorithm used by C-IL^{2}P is standard backpropagation (Rumelhart et al., 1994). C-IL^{2}P also does not train recursive connections: they are fixed and only used for inference.

Then, inference and knowledge extraction can be done. Garcez et al. (2001) proposes a knowledge extraction algorithm for C-IL^{2}P by splitting of the trained network into "regular" ones, which do not have connections coming from the same neuron with different signs (positive and negative). It is shown that the extraction for those networks is sound and complete, and in the general case, soundness can be achieved, but not completeness.

[1] We follow logic programming notation where a clause of the form (A ← B, not C, D) denotes that A is true if B is true, C is false and D is true; clauses are separated by ";" (semi-colon) in a logic program.

[2] Semi-linear bipolar function: $f(x) = \frac{2}{1+e^{-\beta x}} - 1$, where β is a slope parameter with default value 1.

2.3 Propositionalization

Propositionalization is the conversion of a relational database into an attribute-value table, amenable to conventional propositional learners (Krogel et al., 2003). Propositionalization algorithms use background knowledge and examples to find distinctive features, which can differentiate subsets of examples. There are two kinds of propositionalization: *logic-oriented* and *database-oriented*. The former aims to build a set of relevant first-order features by distinguishing between first-order objects. The latter aims to exploit database relations and functions to generate features for propositionalization. The main representatives of *logic-oriented* approaches include: LINUS (and its successors), RSD and RelF; and the main representative of *database-oriented* approaches is RELAGGS (Krogel and Wrobel, 2003). BCP is a new *logic-oriented* propositionalization technique, which consists of generating bottom-clauses for each first-order example and using the set of all body literals that occur in them as possible features (in other words, as columns for an attribute-value table).

In order to evaluate how BCP performs, it will be compared with RSD (Železný and Lavrač, 2006), a well-known propositionalization algorithm for which an implementation is available at (`http://labe.felk.cvut.cz/~zelezny/rsd`). RSD is a system which tackles the *Relational Subgroup Discovery* problem: given a population of individuals and a property of interest, RSD seeks to find population subgroups that are as large as possible and have the most unusual distribution characteristics. RSD's input is an Aleph-formatted dataset, with background knowledge, example set and language bias and its output is a list of clauses that describe interesting subgroups of the examples dataset. RSD is composed of two steps: first-order feature construction and rule induction. The first is a propositionalization method that creates higher-level features that are used to replace groups of first-order literals, and the second is an extension of the propositional CN2 rule learner (Clark and Niblett, 1989), for use as a solver of the relational subgroup discovery problem. We are interested in the propositionalization component of RSD, which can be further divided into three steps: all expressions that by definition form a first-order feature and comply with the mode declarations are identified; the user can instantiate variables (through *instantiate/1* predicates) in the background knowledge and afterwards, irrelevant features are filtered out; and a propositionalization of each example using the generated features is created. From now on, when we refer to RSD, we are referring to the RSD propositionalization method, not the relational subgroup discovery system.

2.4 Feature Selection

As stated in Section 2.1, bottom clauses are extensive representations of an example, possibly having an infinite size. In order to tackle this problem, at least two approaches have been proposed: reducing the size of the clauses during generation or using a statistical approach afterwards. The first can be done as part of the bottom clause generation algorithm (Muggleton, 1995), by reducing the variable depth value. Variable depth specifies an upper bound on the number of times that the algorithm can pass through *mode declarations* and by reducing its value, it is possible to cut a considerable chunk of literals, although causing some information loss. Alternatively, statistical methods such as Pearson's correlation and Principal Component Analysis can be used (a survey of those methods can be found in May et al. (2011)), taking advantage of the use of numerical feature vectors as training patterns. A recent method, which has low computational cost, while surpassing most common methods in terms of information loss, is the mRMR algorithm (Ding and Peng, 2005), which focuses on balancing minimum redundancy and maximum relevance of features, selecting them by using mutual information

I between variables x and y, defined as:

$$I(x,y) = \sum_{i,j} p(x_i, y_j) \log \frac{p(x_i, y_j)}{p(x_i)p(y_j)}, \tag{5}$$

where $p(x,y)$ is the joint probability distribution, and $p(x)$ and $p(y)$ are the respective marginal probabilities. Given a subset S of the feature set Ω to be ranked by mRMR, the minimum redundancy condition and the maximum relevance condition, respectively, are:

$$min\ \{W_I\},\ W_I = \frac{1}{|S|^2} \sum_{i,j \in S} I(i,j) \text{ and} \tag{6}$$

$$max\ \{V_I\},\ V_I = \frac{1}{|S|} \sum_{i \in S} I(h,i), \tag{7}$$

where $h = \{h_1, h_2, \ldots, h_K\}$ is the classification variable of a dataset with K possible classes. Let $\Omega_S = \Omega - S$ be the set of unselected features from Ω. There are two ways of combining the two conditions above to select features from Ω_S: Mutual Information Difference (MID), defined as $\max(V_I - W_I)$, and Mutual Information Quotient (MIQ), defined as $\max(V_I / W_I)$. Results reported in Ding and Peng (2005) indicate that MIQ usually chooses better features. Thus, MIQ is the function we choose to select features in this work and for the sake of simplicity, whenever this work refers to mRMR, it is referring to mRMR with MIQ.

3 Learning with BCP using CILP++

Let us start with a motivating example: consider the well-known family relationship example (Muggleton and Raedt, 1994), with background knowledge B = {*mother(mom1, daughter1)*, *wife(daughter1, husband1)*, *wife(daughter2, husband2)*}, with positive example *motherInLaw(mom1, husband1)*, and negative example *motherInLaw(daughter1, husband2)*. It can be noticed that the relation between *mom1* and *husband1*, which the positive example establishes, can be alternatively described by the sequence of facts *mother(mom1, daughter1)* and *wife(daughter1, husband1)* in the background knowledge. This states semantically that *mom1* is a mother-in-law because *mom1* has a married daughter, namely, *daughter1*. Applied to this example, the bottom clause generation algorithm of Progol would create a clause $\perp$ = *motherInLaw*$(A,B) \leftarrow$ *mother(A, C)*, *wife(C, B)*. Comparing $\perp$ with the sequence of facts above, we notice that $\perp$ describes one possible meaning of mother-in-law: "*A* is a mother-in-law of *B* if *A* is a mother of *C* and *C* is wife of *B*", i.e. the mother of a married daughter is a mother-in-law. This is why, in this paper, we investigate learning from bottom clauses. However, for each learned clause, Progol uses a single random positive example to generate a bottom clause, for limiting the search space. To learn from bottom clauses, BCP generates one bottom clause for each (positive or negative) example e, which we denote as $\perp_e$.

In this section, we introduce the CILP++ system, which extends the C-IL^{2}P system to learn from first-order logic using BCP. Each step of this relational learning task is explained in detail in what follows.

3.1 Bottom Clause Propositionalization

The first step of relational learning with CILP++ is to apply BCP. Each target literal is converted into a numerical vector that an ANN can use as input. In order to achieve this, each

example is transformed into a bottom clause and mapped onto features on an attribute-value table, and numerical vectors are generated for each example. Thus, BCP has two steps: bottom clause generation and attribute-value mapping.

In the first step, each example is given to Progol's bottom clause generation algorithm (Tamaddoni-Nezhad and Muggleton, 2009) to create a corresponding bottom clause representation. To do so, a slight modification is needed to allow the same *hash* function to be shared among all examples, in order to keep consistency between variable associations, and to allow negative examples to have bottom clauses as well; the original algorithm deals with positive examples only. This modified version is shown in Algorithm 1, which has a single parameter, $depth$, which is the *variable depth* of the bottom clause generation algorithm.

Algorithm 1 Adapted Bottom Clause Generation

1: $E_\perp = \emptyset$
2: **for** each example e of E **do**
3: Add e to background knowledge and remove any previously inserted examples
4: $inTerms = \emptyset$, $\perp_e = \emptyset$, $currentDepth = 0$
5: Find the first *mode declaration* with head h which θ-subsumes e
6: **for all** $v/t \in \theta$ **do**
7: If v is of type #, replace v in h to t
8: If v is of one of $\{+, -\}$, replace v in h to v_k, where $k = hash(t)$
9: If v is of type $+$, add t to $inTerms$
10: **end for**
11: Add h to $\perp_e$
12: **for** each body mode declaration b with recall value *recall* **do**
13: **for all** substitutions θ of arguments $+$ of b to elements of $inTerms$ **do**
14: **repeat**
15: **if** querying $b\theta$ against the background knowledge succeeds **then**
16: **for** each v/t in θ **do**
17: If v is of type #, replace v in b to t
18: Else, replace v in b to v_k, where $k = hash(t)$
19: If v is of type $-$, add t to $inTerms$
20: **end for**
21: Add $b\theta$ to $\perp_e$, if it has not been added already
22: **end if**
23: **until** *recall* number of iterations has been reached
24: **end for**
25: **end for**
26: Increment $currentDepth$; if it is less than $depth$, go back to line 12
27: If e is a negative example, add an explicit negation symbol "$\sim$" to the head of $\perp_e$
28: Add $\perp_e$ to $E_\perp$
29: **end for**
30: **return** $E_\perp$

For example, if Algorithm 1 is executed with $depth = 1$ on the positive and negative examples of our motivating (family relationship) example above, *motherInLaw(mom1, husband1)* and *motherInLaw(daughter1, husband2)*, respectively, it generates the following training set:

$$E_\perp = \{motherInLaw(A,B) : -mother(A,C), wife(C,B);$$

$\sim motherInLaw(A,B) : -wife(A,C)\}$.

After the creation of the $E_\perp$ set, the second step of BCP is as follows: each element of $E_\perp$ (each bottom clause) is converted into an input vector v_i, $0 \leq i \leq n$, that a propositional learner can process. The algorithm for that, implemented by CILP++, is as follows:

1. Let $|L|$ be the number of distinct body literals in $E_\perp$;
2. Let E_v be the set of input vectors, converted from $E_\perp$, initially empty;
3. For each bottom clause $\perp_e$ of $E_\perp$ do
   (a) Create a numerical vector v_i of size $|L|$ and with 0 in all positions;
   (b) For each position corresponding to a body literal of $\perp_e$, change its value to 1;
   (c) Add v_i to E_v;
   (d) Associate a label 1 to v_i if e is a positive example, and -1 otherwise;
4. Return E_v.

As an example, for the same (family relationship) bottom clause set $E_\perp$ above, $|L|$ is equal to 3, since the literals are $mother(A,C)$, $wife(C,B)$ and $wife(A,C)$. For the positive bottom clause, a vector v_1 of size 3 is created with its first position corresponding to $mother(A,C)$, and second position corresponding to $wife(C,B)$ receiving value 1, resulting in a vector $v_1 = (1,1,0)$. For the negative example, only $wife(A,C)$ is in $E_\perp$ and its vector is $v_2 = (0,0,1)$.

3.2 CILP++ Building Phase

Having created numerical vectors from bottom clauses, CILP++ then creates an initial network for training. Background knowledge (BK) only passes through BCP's first step (resulting in a bottom clause set $E_\perp$), i.e. their bottom clauses are generated, but they are not converted into input vectors. CILP++ then maps each body literal onto an input neuron and each head literal onto an output neuron. Following the C-IL^{2}P building step, only bottom clauses generated from positive examples can be used as background knowledge. Let $E_\perp^+$ denote the subset of $E_\perp$ containing bottom clauses generated from positive examples only. Thus, any subset $E_\perp^{BK} \subseteq E_\perp^+$ can be used as background knowledge (or none at all) for the purpose of evaluating CILP++[3].

The CILP++ algorithm for the building phase is presented below. Following C-IL^{2}P, it uses positive weights W to encode positive literals, and negative weights $-W$ to encode negative literals. The value of W for CILP++ is also constrained by Eq. 2,which guarantees the correctness of the translation, i.e. it can be shown that the network computes an intended meaning of the background knowledge (Garcez and Zaverucha, 1999). As in C-IL^{2}P, CILP++ builds so-called AND-OR networks, setting network biases w.r.t. W so that the hidden neurons implement a logical-AND, and the output neurons implement a logical-OR, as discussed in the Background section, as follows:

For each bottom clause $\perp_e$ of $E_\perp^{BK}$, do:

[3]In the next section, we evaluate CILP++ using no BK and different percentages of $E_\perp^{BK}$ as BK. The BK is responsible for setting up initial weight configurations in the network. With no BK, the weights are chosen randomly. BK can also be reinforced during training when it is assumed to be correct. In our experiments in the next section, BK is reinforced by being also presented as examples for training with backpropagation (since BK and examples have the same format in CILP++). Initial weight configurations tend to fade away over time during learning, and BK reinforcement can help improve learning performance.

1. Add a neuron h to the hidden layer of a network $\mathbf{N}$ and label it $\bot_e$;
2. Add input neurons to $\mathbf{N}$ with labels corresponding to each literal in the body of $\bot_e$;
3. Connect the input neurons to h with weight W if the corresponding literals are positive, and $-W$ otherwise;
4. Add an output neuron o to $\mathbf{N}$ and label it with the head literal of $\bot_e$;
5. Connect h to o with weight W;
6. Set the biases in the following way: input neurons with bias 0, bias of h with Eq. 3, and bias of o with Eq. 4.

Continuing our example, suppose that the positive example of $\mathrm{E}_\bot$:

$$motherInLaw(A,B) \ :- \ mother(A,C), wife(C,B) \tag{8}$$

is to be used as background knowledge to build an initial ANN. In step 1 of the CILP++ building algorithm, a hidden neuron is created having Eq. (8) as associated label. In step 2, two input neurons are created, representing the body literals $mother(A,C)$ and $wife(C,B)$. In step 3, two connections are created from each input neuron to the hidden neuron, both having weight W. In step 4, an output neuron representing the head literal $motherInLaw(A,B)$ is created. In step 5, the hidden layer neuron is connected to the output neuron with weight W, and the network biases are set in step 6[4].

In order to evaluate network building, in the next section, we run experiments using different sizes of $E_\bot^{BK}$, including a network configuration with no BK, i.e. where only the input and output layers are built and associated with bottom clause literals, but no specific initial number of hidden neurons is prescribed, as detailed in what follows.

3.3 CILP++ Training Phase

After BCP is applied and a network is built, CILP++ training is next. As an extension of C-IL^{2}P, CILP++ uses backpropagation. Differently from C-IL^{2}P, CILP++ also has a built-in cross-validation method and an *early stopping* option (Prechelt, 1997). Validation is used to measure generalization error during each training epoch. With early stopping, when an error measure starts to increase, training is stopped. A more permissive version of early stopping, which we use, does not halt training immediately after the validation error increases, but when the criterion in Eq. 9 is satisfied, where α is the stopping criterion parameter, t is the current epoch number, $Err_{va}(t)$ is the average validation error on epoch t and $Err_{opt}(t)$ is the least validation error obtained from epochs 1 up to t. The reason we apply Eq. 9 is that, without feature selection, BCP can generate large networks; early stopping has been shown effective at avoiding overfitting in large networks (Caruana et al., 2000).

$$GL(t) > \alpha, \ GL(t) = 0.1 \cdot \left(\frac{Err_{va}(t)}{Err_{opt}(t)} - 1 \right) \tag{9}$$

Given a bottom clause set $E_\bot^{train}$, the steps below are followed for training network $\mathbf{N}$:

[4]Notice that CILP++ is able to build a recursive network in the same way as C-IL^{2}P, but no recursive connections are created by the building algorithm in this paper because a recursive network is not required when target concepts (head literals) cannot appear as body literals in a background knowledge rule or inside *modeb* declarations. This is the case of the experiments/datasets used in this paper.

1. For each bottom clause $\bot_e \in E_\bot^{train}$, $\bot_e = h$:- $l_1, l_2, ..., l_n$, do:
   (a) Add all $l_i, 1 \leq i \leq n$, that are not represented yet in the input layer of **N**, as new neurons;
   (b) If h does not exist yet in the network, create an output neuron corresponding to it;
2. Add new hidden neurons, if required for convergence;
3. Make the network fully-connected, by adding weights with zero values;
4. Normalize all weights and biases (as explained below);
5. Alter weights and biases slightly, to avoid the symmetry problem[5];
6. Apply backpropagation using each $\bot_e \in E_\bot^{train}$ as training example.

The normalization process of step 4 above is done to solve a problem found while experimenting with C-IL^{2}P: the initial weight values for the connections, depending on the background knowledge that is being mapped, could be excessively large, which makes the derivative of the semi-linear activation function tend to zero, thus not allowing proper training. We used a standard normalization procedure for ANNs, described in Haykin (2009): let w_l be a weight in layer l and similarly, let b_l be a bias. For each l, the normalized weights and biases (respectively, w_l^{norm} and b_l^{norm}) are defined as:

$$w_l^{norm} = w_l \cdot \frac{1}{(|l-1|^{\frac{1}{2}}) \cdot max_w} \text{ and}$$

$$b_l^{norm} = b_l \cdot \frac{1}{(|l-1|^{\frac{1}{2}}) \cdot max_w},$$

where $|l|$ is the number of neurons in layer l and max_w is the maximum absolute connection weight value among all weight connections in the network.

To illustrate the training phase, assume that the bottom clause set $E_\bot$ is our training data and no background knowledge has been used. In step 1(a), all body literals from both examples ($mother(A,C)$, $wife(C,B)$ and $wife(A,C)$) cause the generation of three new input neurons in the network, with labels identical to the corresponding literals. In step 1(b), an output neuron labeled $motherInLaw(A,B)$ is added. In step 2, let us assume that two hidden neurons are added. In step 3, zero-weighted connections are added from all three input neurons to both hidden neurons, and from those to the output neuron. Step 4 is only needed when background knowledge is used. In step 5, we add a random non-zero value in $[-0.01, 0.01]$ to each weight. Finally, in step 6, backpropagation is applied (see Fig. 2), firing the input neurons $mother(A,C)$ and $wife(C,B)$ when the positive example is being learned (example 1 in the figure), with target output 1, and firing the input neuron $wife(A,C)$ when the negative example is being learned (example 2 in the figure), with target output -1.

Additionally, notice that BCP does not combine first-order literals to generate features like RSD or SINUS: it treats each literal of a bottom clause as a feature. The hidden layer of the ANN can be seen as a (weighted) combination of the features provided by the input layer. Thus, ANNs can combine features when processing the data, which allows CILP++ to group features similarly to RSD or SINUS, but doing so dynamically (during learning), due to the small changes of real-valued weights in the network.

After training, CILP++ can be evaluated. Firstly, each test example e^{test} from a test set E^{test} is propositionalized with BCP, resulting in a propositional data set $E_\bot^{test}$, where each $e^{test} \in E^{test}$

[5]The symmetry problem states that if all weights start out with equal values and if the solution requires that unequal weights be developed, the system can never learn (Rumelhart et al., 1986).

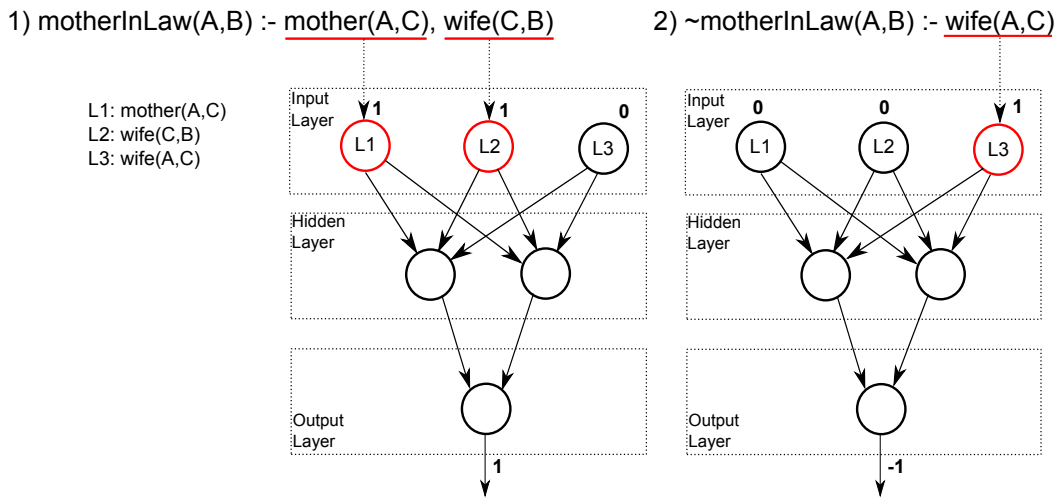

Figure 2: Illustration of CILP++'s training step. L1, L2 and L3 are labels corresponding to each distinct body literal found in $E_{\perp}$. The shown output values are the labels which are used for backpropagation training. In the figure, network **N** appears repeated for each example, for clarity.

has a corresponding $\perp_e^{test} \in E_{\perp}^{test}$. Then, each $\perp_e^{test}$ is tested in CILP++'s ANN: each input neuron corresponding to a body literal of $\perp_e^{test}$ receives input 1 and all other input neurons (input neurons which labels are not present in $\perp_e^{test}$) receive input 0. Lastly, a feedforward pass through the network is performed, and the output will be CILP++'s answer to $\perp_e^{test}$ and consequently, to e^{test}.

4 Experimental Results

In this section, we present the experimental methodology and results for CILP++ as a first-order neural-symbolic system and for BCP as a standalone propositionalization method. We also compare results with ILP system Aleph and propositionalization method RSD. Before we experiment on ILP problems, though, we have tested CILP++ against its predecessor, C-IL^{2}P, to evaluate whether CILP++ is as good as C-IL^{2}P on propositional problems. We used the Gene Sequences/Promoter Recognition dataset used in Garcez and Zaverucha (1999) with leave-one-out cross validation. CILP++ obtained 92.41% accuracy, against 92.48% obtained by C-IL^{2}P; CILP++ took 5:21 minutes to run the entire experiment, while C-IL^{2}P took 5:23 minutes. This suggests that CILP++ and C-IL^{2}P perform similarly on propositional problems.

As mentioned, we have compared results with Aleph and RSD. Aleph is an ILP system, which has several other algorithms built-in, such as Progol (by default). RSD is a well-known propositionalization method capable of obtaining results comparable to full ILP systems. We have used four benchmarks: the *Mutagenesis* dataset (Srinivasan and Muggleton, 1994), the *KRK* dataset (Bain and Muggleton, 1994), the *UW-CSE* dataset (Richardson and Domingos, 2006), and the *Alzheimers* benchmark (King and Srinivasan, 1995), which consists of four datasets: *Amine*, *Acetyl*, *Memory* and *Toxic*. Table 1 reports some general characteristics and the number of BCP features obtained for each dataset.

We have run CILP++ on the above datasets (all folds are available from `http://soi.city.ac.uk/~abdz937/bcexperiments.zip`, including our version of the *UW-CSE*

Table 1: Datasets statistics

Dataset	# Positive Examples	# Negative Examples	# Predicates	# BCP Features
Mutagenesis	125	63	34	1115
KRK	341	655	9	60
UW-CSE	113	226	37	430
Alz-amine	343	343	30	1090
Alz-acetyl	618	618	30	1363
Alz-memory	321	321	30	1052
Alz-toxic	443	443	30	1319

dataset, as explained below), reporting results on six CILP++ configurations. We report: accuracy vs. runtime on all datasets in comparison with Aleph, and a comparison between BCP and RSD on the *Mutagenesis* and *KRK* datasets[6]. We also evaluate feature selection in CILP++ by constraining the clause length when building bottom clauses with BCP and applying mRMR.

Since a varied number of accuracy results have been reported in the literature on the use of Aleph with the *Alzheimers* and *Mutagenesis* datasets (King and Srinivasan, 1995; Landwehr et al., 2007; Paes et al., 2007), we have decided to run both Aleph and CILP++ for our comparisons. We built 10 folds from each dataset (in the case of *UW-CSE*, we followed Davis et al. (2005) and used 5 folds) and both systems used the exact same training folds. BCP and RSD could not, however, share the exact same training folds, as a result of the way in which the RSD tool was implemented (the RSD tool generates features before the folds are created, while CILP++ creates the folds in the first place). As mentioned earlier, the CILP++ system, the different configurations/parametrizations, and all the data folds are available for download so that the results reported in this paper should be reproducible. The six CILP++ configurations include:

- *st*: uses standard backpropagation stopping criteria;
- *es*: uses early stopping;
- *n%bk*: the network is created using $n\%$ of the examples in $E_{\perp}^{train}$ as BK[7];
- $2h$: uses no building step and starts with 2 hidden neurons only.

The choice of the $2h$ configuration is explained in detail in Haykin (2009); ANNs having two neurons in the hidden layer can generalize binary problems approximately as well as any network. Furthermore, if a network has many features to evaluate, as in the case of BCP, i.e. the input layer has many neurons, it should have sufficient *degrees of freedom*; further increasing it by adding hidden neurons might increase the chances of overfitting. Since bottom clauses are "rough" representations of examples and we would like to model the general characteristics of the examples, a simpler model such as *2h* should be preferred (Caruana et al., 2000).

For all experiments with Aleph, the same configurations as in Landwehr et al. (2007) were used for the *Alzheimers* datasets (any parameter not specified below used Aleph's default values): *variable depth* = 3, *least positive coverage of a valid clause* = 2, *least accuracy of an acceptable clause* = 0.7, *minimum score of a valid clause* = 0.6, *maximum number of literals in a valid clause* = 5 and *maximum number of negative examples covered by a valid clause (noise)*

[6]Since the available implementation of RSD handles target concepts with arity 1, we were unable to apply the *Alzheimers* datasets to RSD (whose targets have arity 2). An alternative would be to modify the domain knowledge in the datasets, but this is not a straightforward task.

[7]$n = 2.5$ and 5 were used.

= 300. Regarding *Mutagenesis*, the parameters were based on Paes et al. (2007) (again, if a parameter is not listed below, the Aleph default value has been used): *least positive coverage of a valid clause* = 4. For *KRK*, the configuration provided by Aleph in its documentation was used. For *UW-CSE*, the same configuration as in Davis et al. (2005) was used: *variable depth* = 3, *least positive coverage of a valid clause* = 10, *least accuracy of an acceptable clause* = 0.1, *maximum number of literals in a valid clause* = 10, *maximum number of negative examples covered by a valid clause (noise)* = 1000 and *evaluation function* = *m-estimate*.

With regards to the *UW-CSE* dataset, we have used an ILP version of the dataset following Davis et al. (2005). The original *UW-CSE* dataset contains positive examples only for use with Markov Logic Networks (Richardson and Domingos, 2006). Davis et. al. generated negative examples for this dataset using Closed World Assumption (Davis et al., 2005). This has produced an unbalanced dataset containing 113 positive examples and 1772 negative examples. Thus, we have re-balanced the dataset by performing random undersampling, until we had obtained twice as many negative examples as positive examples. Our goal was to cover the distribution of negative examples as best as possible, while not allowing too much unbalancing, and to provide a fair comparison with Aleph. Alternative undersamplings and oversamplings have been investigated also with results reported below.

As for the CILP++ parameters, we used the same variable depth values as Aleph for BCP (except for *UW-CSE*, where we used *variable depth* = 1, as discussed below) and the following parameters for backpropagation[8]: on *st* configurations, *learning rate* = 0.1, *decay factor* = 0.995 and *momentum* = 0.1; and on *es* configurations: *learning rate* = 0.05, *decay factor* = 0.999, *momentum* = 0 and *alpha* (early stopping criterion) = 0.01.

Finally, extra hidden neurons were not added to the network configurations above, i.e. step 2 of CILP++'s training algorithm, Section 3.3, was not applied. The networks labeled as *2h* have only 2 hidden neurons, and those labeled *n%bk* have as many hidden neurons as the size of the BK, i.e. *n%* the size of the set $E_{\perp}^{train}$.

4.1 Accuracy Results

In this experiment, CILP++ is evaluated on accuracy vs. runtime against Aleph. Two tables are presented with accuracy averages, standard deviations and complete runtimes over 10-fold cross-validation for *Mutagenesis*, four *Alzheimers* datasets and *KRK*, and 5-fold cross-validation for *UW-CSE*, on the *st* (Table 2) and *es* (Table 3) CILP++ configurations. By "complete runtime" we mean the total building, training and testing times for each system. In both tables, accuracy results in bold are the highest ones and the difference between them and the ones marked with asterisk (*) are statistically significant by two-tailed, paired t-test. All experiments were run on a 3.2 Ghz Intel Core i3-2100 with 4 GB RAM.

Notice how CILP++ can achieve runtimes that are considerably faster than Aleph. We believe the speed-ups are caused by the following main factors: ILP covering-based search algorithms have well-known efficiency bottlenecks (Paes et al., 2007, 2008; DiMaio and Shavlik, 2004), while bottom clause generation is fast, and standard backpropagation learning is efficient (Rumelhart et al., 1986). Further, propositionalized examples are generally easier to handle computationally than first-order examples (Krogel et al., 2003). Tables 2 and 3 for the *st* and *es* configurations, respectively, seem to confirm an expected trade-off between speed

[8]CILP++ networks have four parameters: *learning rate*, which is the proportion of the gradient that is applied to each weight; *decay factor*, which indicates the proportion of the learning rate that is used in the next backpropagation iteration; *momentum*, indicates the proportion of the last weight update that is used in the current update; and *alpha*, which is exclusive for *early stopping*, is a threshold on the generalization loss. For more details, please refer to Haykin (2009); Caruana et al. (2000).

Table 2: Test set accuracy (standard deviation) results and runtimes (in % for accuracy and in *hh:mm:ss* format for runtimes) for *st* configurations. It can be seen that Aleph and CILP++ present comparable accuracy results for the standard CILP++ configurations, with the *st,2.5%bk* model winning on three datasets. CILP++ performs faster in most cases, confirming our expectation that relational learning through propositionalization should trade accuracy for efficiency, in comparison with full first-order ILP learners.

Dataset	***Aleph***	***CILP++***$_{st,2.5\%bk}$	***CILP++***$_{st,5\%bk}$	***CILP++***$_{st,2h}$
mutagenesis	$80.85 * (\pm 10.5)$ **0:08:15**	**91.70**($\pm$5.84) 0:10:34	90.65($\pm$8.97) 0:11:15	89.20($\pm$8.92) 0:10:16
krk	**99.6**($\pm$0.51) 0:11:03	$98.31 * (\pm 1.23)$ 0:04:38	$98.32 * (\pm 1.25)$ **0:04:34**	98.42($\pm$1.26) 0:04:40
uw-cse	**84.91**($\pm$7.32) 0:45:47	$66.24 * (\pm 7.01)$ **0:08:47**	$66.08 * (\pm 2.48)$ 0:10:19	$70.01 * (\pm 2.2)$ 0:08:54
alz-amine	78.71($\pm$5.25) 1:31:05	**78.99**($\pm$4.46) 1:23:42	$76.02 * (\pm 3.79)$ 2:07:04	77.08($\pm$5.17) **1:14:21**
alz-acetyl	**69.46**($\pm$3.6) 8:06:06	$63.64 * (\pm 4.01)$ 4:20:28	$63.49 * (\pm 4.16)$ 5:49:51	$63.30 * (\pm 5.09)$ **2:47:52**
alz-memory	**68.57**($\pm$5.7) 3:47:55	$60.44 * (\pm 4.11)$ 1:41:36	$59.19 * (\pm 5.91)$ 2:12:14	$59.82 * (\pm 6.76)$ **1:19:27**
alz-toxic	80.5($\pm$3.98) 6:02:05	79.92($\pm$3.09) 3:04:53	80.49($\pm$3.65) 3:33:17	**81.73**($\pm$4.68) **2:12:17**

and accuracy between propositionalization and methods dealing directly with first-order logic. We can also see that *st* configurations seem to emphasize accuracy, while *es* emphasizes speed.

Regarding the CILP++ results on *UW-CSE*, as mentioned earlier, we have used *variable depth* = 1 for BCP. The reason is that *UW-CSE* examples when propositionalized by BCP for *variable depths* higher than 1 become considerably large. At the same time, *variable depth* 1 causes serious information loss in the propositionalization procedure. To ameliorate this, we have tried an oversampling method called SMOTE, based on kNN (Chawla et al., 2002). SMOTE suggests a combination with random undersampling for better results, whereby we increased the positive examples five times (from 113 to 565 examples) using SMOTE, and undersampled the class of negative examples until we had the same number (565) of negative examples. The problem with this approach, in what concerns a comparison with Aleph, is that, to the best of our knowledge, no oversampling method exists for Aleph; SMOTE is applicable to numerical or propositional data only, thus we could not compare those results with Aleph. Hence, we do not report those results in the accuracy tables above. Nevertheless, with SMOTE, CILP++ obtained 93.34%, 90.11% and 93.58% accuracy for the *es,2h*, *es,2.5%bk* and *es,5%bk* configurations, respectively. None of the networks took longer than 6 minutes to run (train and test) on all 5 *UW-CSE* folds, including the SMOTE and undersampling pre-processing. In *st* configurations, CILP++ obtained 73.44% for *st,2.5%bk*, 77.35% for *st,5%bk* and 74.2% for *st,2h*, with no configuration taking longer than 9 minutes to run completely. Those results indicate that an adequate ANN data pre-processing, enabled by the BCP method, can improve results considerably.

Table 3: Test set accuracy (standard deviation) results and runtimes for the CILP++ configurations with early stopping (in % for accuracy and in *hh:mm:ss* format for runtimes). Using *es* models, CILP++ was much faster than Aleph but with a considerable decrease in accuracy. Aleph won in accuracy in all but the Mutagenesis dataset. This indicates that early stopping is not recommended in general for use with BCP, unless speed is paramount.

Dataset	***Aleph***	***CILP++***$_{es,2.5\%bk}$	***CILP++***$_{es,5\%bk}$	***CILP++***$_{es,2h}$
mutagenesis	80.85($\pm$10.51) 0:08:15	83.48($\pm$7.68) **0:01:25**	83.01($\pm$10.71) 0:01:43	**84.76**($\pm$8.34) 0:01:50
krk	**99.6**($\pm$0.51) 0:11:03	98.16 $*$ ($\pm$0.83) **0:04:08**	96.33 $*$ ($\pm$4.95) 0:04:28	98.31($\pm$1.23) 0:04:18
uw-cse	**84.91**($\pm$7.32) 0:45:47	68.16 $*$ ($\pm$4.77) **0:04:08**	65.69 $*$ ($\pm$1.81) 0:04:16	67.86 $*$ ($\pm$1.79) **0:04:08**
alz-amine	**78.71**($\pm$3.51) 1:31:05	65.33 $*$ ($\pm$9.32) 0:35:27	65.44 $*$ ($\pm$5.58) **0:08:30**	70.26 $*$ ($\pm$7.1) 0:10:14
alz-acetyl	**69.46**($\pm$3.6) 8:06:06	64.97 $*$ ($\pm$5.81) 3:04:47	64.88 $*$ ($\pm$4.64) 2:42:31	65.47 $*$ ($\pm$2.43) **0:25:43**
alz-memory	**68.57**($\pm$5.7) 3:47:55	53.43 $*$ ($\pm$5.64) 1:40:51	54.84 $*$ ($\pm$6.01) 3:57:39	51.57 $*$ ($\pm$5.36) **1:33:35**
alz-toxic	**80.5**($\pm$4.83) 6:02:05	67.55 $*$ ($\pm$6.36) **0:12:33**	67.26 $*$ ($\pm$7.5) 0:14:04	74.48 $*$ ($\pm$5.62) 0:28:39

So far, we have explored a number of CILP++ configurations. The use of other configurations and their combination through tuning sets is possible. However, the ILP literature on Aleph generally reports a single optimal configuration per dataset (and not per fold) (Paes et al., 2007; Landwehr et al., 2007). We believe, therefore, that applying tuning sets to CILP++ would lead to an unfair advantage to the network model, for the sake of comparison with Aleph. Nevertheless, an optimal CILP++ configuration would use tuning sets, and we report those results below on Table 4. A three-fold internal cross validation was applied on the training set of each one of the 10 folds used in Tables 2 and 3. The fold accuracy of the best model, chosen with tuning sets, was then chosen for that fold. Thus, the dataset accuracy of CILP++ using tuning sets is the average of the test set accuracy obtained for each fold with the model that obtained the best tuning set accuracy. We also report the runtimes obtained with this approach and the "best" model for each dataset, which is the one that is chosen the most times, for all the folds. The *best model* results shown in the table were used to guide our choice of model in the experiments on feature selection and BCP to follow.

In comparison with the results reported in Tables 2 and 3, the results using tuning sets were slightly lower than the results of the best individual models, but better than most of them. Additionally, we applied tuning sets to the version of *UW-CSE* to which we applied SMOTE and undersampling, and we obtained 81.12% test set accuracy, with CILP++ taking less than 8 minutes to finish, which is considerably better than the results obtained with *UW-CSE* without SMOTE. In the following experiments (feature selection analysis and BCP results), we choose the best models obtained from tuning sets for further analysis.

Table 4: Results using tuning sets for CILP++. We report three results in this table, from left to right: CILP++ test set accuracy using tuning sets averaged over the six CILP++ configurations, CILP++ runtime using tuning sets, and best model, i.e. the configuration with most wins on the 10 train/test folds (5 train/test folds, in the case of *UW-CSE*). Overall, the best *st* model is the *st,2.5%bk* configuration, the best *es* model is the *es,2h* model, and the best model overall is the *st,2.5%bk* model.

Dataset	**Test Set Accuracy**	**Runtime**	**Best model**
mutagenesis	88.84(±10.48)	0:07:54	*st,5%bk* (3/10)
krk	96.75(±4.9)	0:04:19	*st,2h* (8/10)
uw-cse	66.84(±7.32)	0:06:11	*st,2.5%bk* (2/5)
alz-amine	76.45(±3.45)	1:31:11	*st,2.5%bk* (6/10)
alz-acetyl	64.07(±6.2)	0:30:35	*es,2h* (7/10)
alz-memory	59.67(±5.7)	1:51:02	*st,2.5%bk* (4/10)
alz-toxic	81.73(±4.68)	2:12:17	*st,2h* (10/10)

4.2 Comparative Results with Propositionalization

In this section, comparative results against RSD are carried out, using the datasets *Mutagenesis* (named *muta* in the table below) and *KRK* (the reason for this choice of datasets is explained in the previous section). In Table 5, accuracy and runtimes are shown. We compare both BCP and RSD propositionalization when generating training patterns for CILP++ (labeled *ANN* in the table) and for the C4.5 decision tree learner. Aleph results are shown as well as a baseline. We use the CILP++ configuration that obtained the best results in the tuning sets for each dataset. Values in bold are the highest obtained, and the difference between those and the ones marked with (*) are statistically significant by two-tailed, unpaired t-test (we use unpaired t-test because of the RSD tool implementation issue, mentioned earlier). All experiments were also run on a 3.2 Ghz Intel Core i3-2100 with 4 GB RAM.

Table 5: Accuracy and runtime results for *Mutagenesis* and *KRK* datasets (in % for accuracy and in *hh:mm:ss* format for runtimes). The results show that BCP is faster than RSD, while showing highly competitive results w.r.t. Aleph, but RSD performed as well as BCP when using C4.5 as learner. BCP outperformed RSD in all models: BCP was faster in all cases, but in the *KRK* dataset, RSD with C4.5 showed higher accuracy, although the difference was not statistically significant. The results also show that BCP performs well with both learners (ANN and C4.5), but excels with ANNs. On the other hand, RSD did not perform well with ANNs.

Dataset	***Aleph***	***BCP+ANN***	***RSD+ANN***	***BCP+C4.5***	***RSD+C4.5***
muta	80.85 ∗ (±10.51) 0:08:15	**89.20**(±8.92) 0:10:16	67.63 ∗ (±16.44) 0:11:11	85.43 ∗ (±11.85) **0:02:01**	87.77(±1.02) 0:02:29
krk	**99.6**(±0.51) 0:11:03	98.42 ∗ (±1.26) 0:04:40	72.38 ∗ (±12.94) 0:06:21	98.84 ∗ (±0.77) **0:01:59**	96.1 ∗ (±0.11) 0:05:54

In summary, our hypothesis was that BCP, as a standalone propositionalization method, can be fast and is capable of generating accurate features for learning. The results indicate that BCP is a good match for ANN, indicating the promise of the CILP++ system. BCP performs on a par with RSD when integrated with C4.5. BCP is also faster than RSD in all cases,

empirically confirming our hypothesis.

4.3 Results with Feature Selection

In Section 2.4, it was discussed that, due to the extensive size of bottom clauses, feature selection techniques may obtain improved results when applied after BCP. Two ways of performing feature selection were discussed: changing the variable depth (see Algorithm 1) and using a statistical method, mRMR. We have chosen two datasets on which to run these experiments with feature selection: *Alz-amine* and *Alz-toxic*. We opted for those because CILP++ performed well on them, not outstandingly well (as in *Mutagenesis*), neither poorly (as in *Alz-acetyl*). Additionally, we have chosen the best *st* configuration ($st, 2.5bk$, chosen by tuning sets) and the best *es* configuration ($es, 2h$). Even though the results using tuning set showed $st, 2h$ as the best model for the *Alz-toxic* dataset, we wanted to analyze feature selection on *es* configurations as well, and so we have chosen the best *es* configuration.

First, we changed the variable depth in *Alz-amine* and *Alz-toxic*, which was 3, to 2 and 5, to analyze how changes in this parameter would affect performance. The results are shown in Fig. 3. Alternatively, we applied mRMR with three levels of selection: 50%, 25% and 10% of the best-ranked features. These results are shown in Fig.4.

In summary, statistical feature selection seems to be useful with BCP. Changes in variable depth did not seem to offer gains, but mRMR offered more than 90% feature reduction with a loss of less than 2% in accuracy. The goal of selecting features with mRMR should not be to improve efficiency, although in one case (*Alz-amine es-2h*), CILP++ with mRMR was faster at 90% feature reduction than CILP++, despite a loss of more than 10% of accuracy.

5 Conclusion and Future Work

This paper has introduced a fast method and algorithm for ILP learning with ANNs, by extending a neural-symbolic system called C-IL^{2}P. The paper's two contributions are: a novel propositionalization method, BCP, and the CILP++ system, an open-source, freely distributed neural-symbolic system for relational learning. CILP++ obtained accuracy comparable to Aleph on most standard configurations and stood behind Aleph, but was faster, on early stopping configurations. In comparison with RSD, CILP++ has been shown superior, but BCP and RSD present similar results when using C4.5 as learner. Nevertheless, BCP obtained better runtime results overall. Lastly, when using feature selection, results have shown that mRMR is applicable with CILP++ and it can reduce drastically the number of features with a small loss of accuracy, despite an increase in runtime in some cases. Feature selection with mRMR can be useful to reduce the size of the network and improve readability, especially if knowledge extraction is needed. Propositionalization methods usually show a trade-off between accuracy and efficiency. Our results show that CILP++ can improve on this trade-off by offering considerable speed-up in exchange for small accuracy loss in some datasets, even achieving better accuracy in some cases.

ILP covering-based hypothesis induction is an efficiency bottleneck in traditional ILP learners such as Aleph (Paes et al., 2007, 2008; DiMaio and Shavlik, 2004). On the other hand, bottom clause generation by itself is fast. Thus, we claim that propositionalizing first-order example with BCP and using an efficient learning algorithm such as backpropagation should offer a faster and reasonably accurate way of dealing with first-order data. Our empirical results seem to confirm this claim.

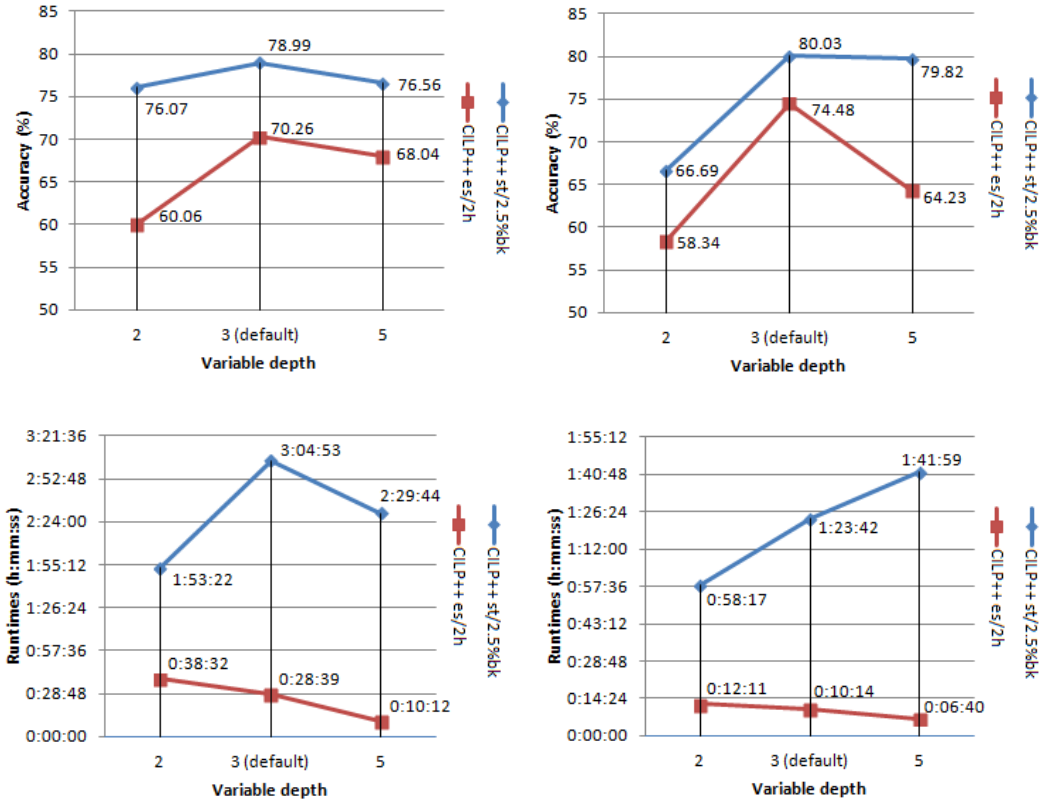

Figure 3: Accuracy (above) with varying variable depth on *Alz-amine* (left) and *Alz-toxic* (right), with runtimes (below) in *hh:mm:ss* format. The results indicate that the default variable depth is satisfactory: neither increasing it nor decreasing it has helped increase performance. As stated in Section 2.1, variable depth controls how far the bottom clause generation algorithm goes when generating concept chaining and it is a way of controlling how much information loss the propositionalization method will have. From this and the obtained results, it should be intuitive that higher variable depths should mean a better performance, but together with useful features, it seems to bring redundancy as well.

As future work, there are a number of avenues for research. First, background knowledge translation into ANNs can be explored further in CILP++. A first attempt could be to use the language bias and the definite clauses from background knowledge to build the network. The study of how the last step of C-IL^{2}P's learning cycle, knowledge extraction, can be done in CILP++, is another area for future work. One option (Craven and Shavlik, 1995) would be to create one clause for each class c and add antecedents to it which correspond to body literals of each bottom clause that belongs to c. Alternatively, the same knowledge extraction procedure which C-IL^{2}P uses can be applied to CILP++, although it is considerably costly and further analysis on the fidelity of the extracted theory is required. This work has shown that learning first-order data with CILP++ is fast, but without considering knowledge extraction. If extraction is to be taken into consideration, faster learning algorithms for ANNs (Jacobs, 1988; Møller, 1993) can be used to try and keep up with Aleph in terms of runtime. Lastly, regarding other analyses that can be done with CILP++, the work of DiMaio and Shavlik (2004), which uses bottom clauses as training patterns to build a hypothesis scoring function, used several meta-parameters such as *size* of the bottom clause and *number* of distinct predicates. The same

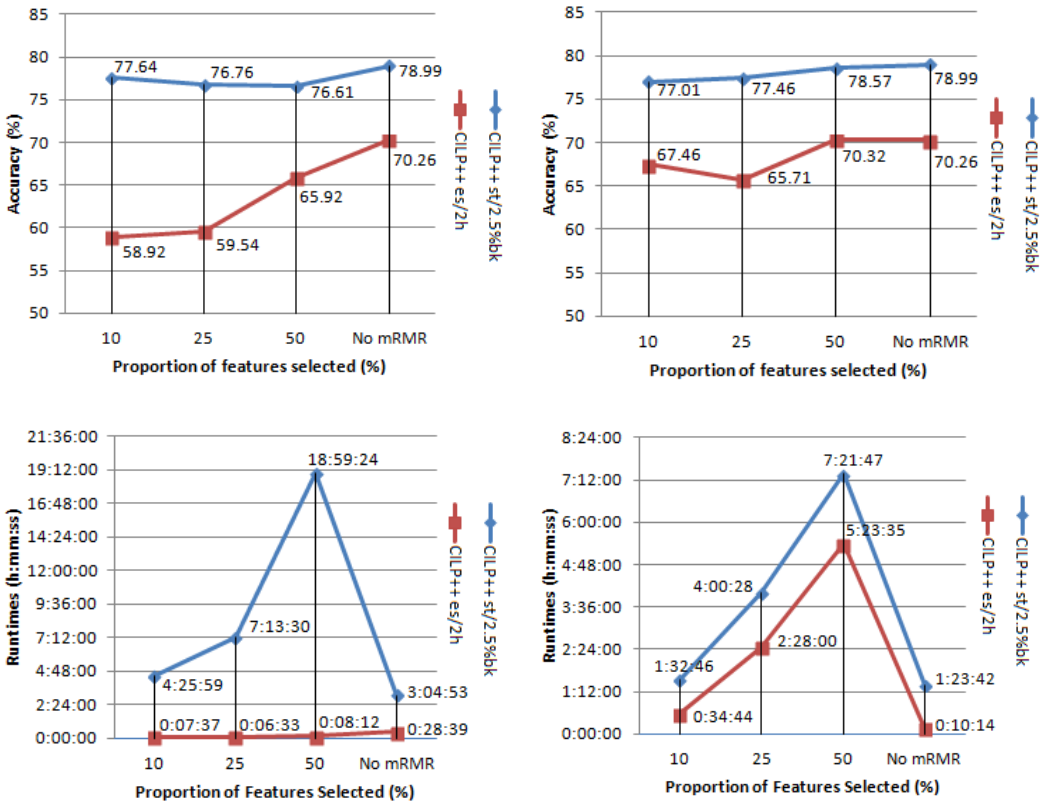

Figure 4: Accuracy (above) when using mRMR on *Alz-amine* (left) and *Alz-toxic* (right), with runtimes (below) in *hh:mm:ss* format. The results show that in both *Alz-amine* and *Alz-toxic* datasets, a reduction of 90% in the number of features caused a loss of less than 2% in accuracy, albeit with an increase in runtime. The reduction in features caused CILP++ to take more training epochs to converge and mRMR itself also contributed to the increase in runtime. However, at 90% filtered features, the runtimes approached in general the ones obtained without mRMR filtering. Even with an increase in runtime, feature selection with mRMR seems useful to reduce the size of the network and improve readability, especially if knowledge extraction is to be carried out.

meta-parameters can be useful to CILP++. Furthermore, experiments on datasets with continuous data could be done: it should be interesting to see how CILP++ behaves on this kind of data and to analyze if this approach inherits the additive noise robustness from traditional backpropagation ANNs (Copelli et al., 1997). Also, due to the results for feature selection with mRMR, it is worth evaluating how our approach deals with very large relational datasets, e.g. CORA or Proteins, which are considered to be challenging for ILP learners (Perlich and Merugu, 2005).

Acknowledgements: Manoel França thanks the Brazilian funding agency CAPES and Gerson Zaverucha thanks the Brazilian funding agencies CNPq, FAPERJ and FACEPE. We also thank Filip Železný for helpful discussions and comments, Alan Perotti for his feedback on the CILP++ system and the anonymous reviewers for their useful comments.

References

Bain, M., & Muggleton, S. (1994). Learning optimal chess strategies. *Machine Intelligence*, 13, 291–309.

Basilio, R., Zaverucha, G., and Barbosa, V. (2001). Learning logic programs with neural networks. In *Proc. ILP*, *LNAI* 2157: 402–408. Springer.

Caruana, R., Lawrence, S., & Giles, C. L. (2000). Overfitting in neural nets: backpropagation, conjugate gradient, and early stopping. In *Proc. NIPS*, 13: 402–408. MIT Press.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16 (1): 321–357.

Clark, P., & Niblett, T. 1989. The CN2 induction algorithm. *Machine Learning*, 3: 261–283.

Copelli, M., Eichhorn, R., Kinouchi, O., Biehl, M., Simonetti, R., Riegler, P., & Caticha, N. (1997). Noise robustness in multilayer neural networks. *EPL (Europhysics Letters)*, 37 (6): 427–432.

Craven, M., & Shavlik, J. W. (1995). Extracting tree-structured representations of trained networks. In *Proc. NIPS*, 9: 24–30. Cambridge, MA, USA: The MIT Press.

Davis, J., Burnside, E. S., Dutra, I. C., Page, D., & Costa, V. S. (2005). An integrated approach to learning Bayesian networks of rules. In *Proc. ECML*, *LNAI* 3720: 84–95. Berlin-Heidelberg, Germany: Springer.

De Raedt, L. (2008). *Logical and relational learning*. Berlin-Heidelberg, Germany: Springer.

De Raedt, L., Frasconi, P., Kersting, K., & Muggleton, S. (2008). *Probabilistic inductive logic programming*. LNAI 4911. Berlin-Heidelberg, Germany: Springer.

DiMaio, F., & Shavlik, J. W. (2004). Learning an approximation to inductive logic programming clause evaluation. In *Proc. ILP*, *LNAI* 3194: 80–97. Springer.

Ding, C., & Peng, H. (2005). Minimum redundancy feature selection from microarray gene expression data. *Journal of Bioinformatics and Computational Biology*, 3 (2): 185–205.

Džeroski, S., & Lavrač, N. (2001). *Relational data mining*. Berlin-Heidelberg, Germany: Springer.

Garcez, A. S. D., & Zaverucha, G. (2012). Multi-instance learning using recurrent neural networks. In *Proc. IJCNN*. 1–6. IEEE.

Garcez, A. S. D., & Zaverucha, G. (1999). The connectionist inductive learning and logic programming system. *Appllied Intelligence*, 11: 59–77.

Garcez, A. S. D., Broda, K., & Gabbay, D. M. (2001). Symbolic knowledge extraction from trained neural networks: a sound approach. *Artificial Intelligence*, 125 (1-2): 155–207.

Garcez, A. S. D., Lamb, L. C., & Gabbay, D. M. (2008). *Neural-symbolic cognitive reasoning*, Berlin-Heidelberg, Germany: Springer.

Garcez, A. S. D., Broda, K. B., & Gabbay, D. M. (2002). *Neural-symbolic learning systems*. Berlin-Heidelberg, Germany: Springer.

Getoor, L., & Taskar, B. (2007). *Introduction to statistical relational learning*. Cambridge, MA, USA: The MIT Press.

Guillame-Bert, M., Broda, K., & Garcez, A. S. D. (2010). First-order logic learning in artificial neural networks. In *Proc. IJCNN*. 1–8. IEEE.

Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3: 1157–1182.

Haykin, S. S. (2009). *Neural networks and learning machines*. Upper Saddle River, NJ, USA: Prentice Hall.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1 (4): 295–307.

Kijsirikul, B., & Lerdlamnaochai, B. K. (2005). First-order logical neural networks. *International Journal of Hybrid Intelligent Systems*, 2 (4): 253–267.

King, R. D., & Srinivasan, A. (1995). Relating chemical activity to structure: An examination of ILP successes. *New Generation Computing*, 13 (3-4): 411–434.

King, R. D., Whelan, K. E., Jones, F. M., Reiser, F. G. K., Bryant, C. H., Muggleton, S. H., Kell, D. B., & Oliver, S. G. (2004). Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427 (6971): 247–252.

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. Cambridge, MA, USA: The MIT Press.

Kramer, S., Lavrač, N., & Flach, P. (2001). Propositionalization approaches to relational data mining. In S. Džeroski (Ed.), Relational Data Mining (pp. 262–291). New York, NY, USA: Springer.

Krogel, M. A., Rawles, S., Železný, F., Flach, P., Lavrač, N., & Wrobel, S. (2003). Comparative evaluation of approaches to propositionalization. In *Proc. ILP*, *LNAI* 2835: 197–214. Springer.

Krogel, M. A., & Wrobel, S. (2003). Facets of aggregation approaches to propositionalization, In *Proc. ILP*, *LNAI* 2835: 30–39. Springer.

Kuželka, O., & Železný, F. (2011). Block-wise construction of tree-like relational features with monotone reducibility and redundancy. *Machine Learning*, 83: 163–192.

Landwehr, N., Kersting, K., & De Raedt, L. D. (2007). Integrating naive Bayes and FOIL. *Journal of Machine Learning Research*, 8: 481–507.

Lavrač, N., & Džeroski, S. (1994). *Inductive logic programming: techniques and applications*. Chichester, UK: E. Horwood.

May, R., Dandy, G., & Maier, H. (2011). Review of input variable selection methods for artificial neural networks. In K. Suzuki (Ed.), *Artificial Neural Networks – Methodological Advances and Biomedical Applications* (pp. 19–44). InTech, doi:10.5772/16004.

Møller, M. F. (1993). A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6 (4): 525–533.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing*, 13 (3-4): 245–286.

Muggleton, S., & De Raedt, L. D. (1994). Inductive logic programming: theory and methods. *Journal of Logic Programming*, 19/20: 629–679.

Muggleton, S., Paes, A., Costa, V. S., & Zaverucha, G. (2010). Chess revision: Acquiring the rules of chess variants through FOL theory revision from examples. In *Proc. ILP*, *LNAI* 5989: 123–130. Springer.

Muggleton, S., & Tamaddoni-Nezhad, A. (2008). QG/GA: a stochastic search for Progol. *Machine Learning*, 70: 121–133.

Nienhuys-Cheng, S. H., & de Wolf, R. (1997). *Foundations of inductive logic programming*. LNAI 1228. Berlin-Heidelberg, Germany: Springer.

Paes, A., Revoredo, K., Zaverucha, G., & Costa, V. S. (2005). Probabilistic first-order theory revision from examples. In *Proc. ILP*, *LNAI* 3625: 295–311. Springer.

Paes, A., Zaverucha, G., & Costa, V. S. (2008). Revising first-order logic theories from examples through stochastic local search. In *Proc. ILP*, *LNAI* 4894: 200–210. Springer.

Paes, A., Železný, F., Zaverucha, G., Page, D., & Srinivasan, A. (2007). ILP through propositionalization and stochastic k-term DNF learning. In *Proc. ILP*, *LNAI* 4455: 379–393. Springer.

Perlich, C., & Merugu, S. (2005). Gene classification: issues and challenges for relational learning. In *Proc. 4th International Workshop on Multi-Relational Mining*, ACM. 61–67.

Pitangui, C. G., & Zaverucha, G. (2012). Learning theories using estimation distribution algorithms and (reduced) bottom clauses. In *Proc. ILP*, *LNAI* 7207: 286–301. Springer.

Prechelt, L. (1997). Early stopping - but when? In *Neural networks: tricks of the trade*, *LNAI* 1524 (2): 55–69. Springer.

Quinlan, J. R. (1993). *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine Learning*, 62: 107–136.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart, & J. L. McClelland (Eds.), *Parallel distributed processing: explorations in the microstructure of cognition* (pp. 318–362). Cambridge, MA, USA: MIT Press.

Rumelhart, D. E., Widrow, B., & Lehr, M. A. (1994). The basic ideas in neural networks. *Communications of the ACM*, 37 (3): 87–92.

Srinivasan, A. (2007). The Aleph System, version 5. http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html. Accessed 27 March 2013.

Srinivasan, A., & Muggleton, S. H. (1994). Mutagenesis: ILP experiments in a non-determinate biological domain. In *Proc. ILP*, *LNAI* 237: 217–232.

Tamaddoni-Nezhad, A., & Muggleton, S. (2009). The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. *Machine Learning*, 76 (1): 37–72.

Uwents, W., Monfardini, G., Blockeel, H., Gori, M., & Scarselli, F. (2011). Neural networks for relational learning: an experimental comparison. *Machine Learning*, 82 (3): 315–349.

Železný, F., & Lavrač, N. (2006). Propositionalization-based relational subgroup discovery with RSD. *Machine Learning*, 62: 33–63.