APRENDIZADO RELACIONAL ATRAVÉS DO USO DE CLÁUSULAS MAIS ESPECÍFICAS NO SISTEMA C-IL^{2}P

Manoel Vitor Macedo França

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Gerson Zaverucha

Rio de Janeiro
Dezembro de 2012

APRENDIZADO RELACIONAL ATRAVÉS DO USO DE CLÁUSULAS MAIS ESPECÍFICAS NO SISTEMA C-IL^{2}P

Manoel Vitor Macedo França

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Gerson Zaverucha, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Teresa Bernarda Ludermir, Ph.D.

Prof. Marley Maria Bernades Rebuzzi Vellasco, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 2012

França, Manoel Vitor Macedo

Aprendizado Relacional Através do Uso de Cláusulas Mais Específicas no Sistema C-IL^{2}P/Manoel Vitor Macedo França. – Rio de Janeiro: UFRJ/COPPE, 2012.

X, 94 p. 29, 7cm.

Orientador: Gerson Zaverucha

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2012.

Referências Bibliográficas: p. 34 – 39.

1. Aprendizado. 2. Relacional. 3. Neuro-simbólico. 4. Lógica. 5. Primeira-ordem. I. Zaverucha, Gerson. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Às mulheres da minha vida e à Deus, sempre acima de tudo. "Os sonhos são nossos, mas o veredito da realização é Dele".

Agradecimentos

Meus primeiros agradecimentos vão para os professores Gerson Zaverucha pela orientação e pela paciência que teve ao longo desse mestrado e Artur Garcez pelo auxílio inestimável ao desenvolvimento do meu trabalho.

Agradeço também à minha família pelo apoio, incentivo e puxões de orelha em alguns momentos cruciais, em especial à minha mãe e à minha irmã. E claro, agradeço à minha princesa por ter ficado todo esse tempo ao meu lado.

Por fim, menciono algumas pessoas que tiveram participação direta ou indireta na conclusão desse trajeto: as colegas de laboratório Ana Duboc e Aline Paes e os professores Mário Benevides e Rodrigo Salvador. Finalizo agradecendo à UFRJ pelas instalações físicas e à CAPES pelo suporte financeiro, e deixando um abraço para todos os colegas de graduação pelos momentos de descontração e palavras de incentivo.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

APRENDIZADO RELACIONAL ATRAVÉS DO USO DE CLÁUSULAS MAIS ESPECÍFICAS NO SISTEMA C-IL^{2}P

Manoel Vitor Macedo França

Dezembro/2012

Orientador: Gerson Zaverucha

Programa: Engenharia de Sistemas e Computação

Neste trabalho, é apresentado um algoritmo para aprendizado relacional usando um novo método de proposicionalização e aprendendo com redes neurais artificiais, através da extensão do sistema neuro-simbólico C-IL^{2}P de Garcez e Zaverucha, nomeado CILP++, que usa conhecimento prévio codificado como um programa de lógica proposicional para construir uma rede neural artificial recorrente e aprende com exemplos através do algoritmo de retro-propagação. CILP++ incorpora uma série de melhorias, além da capacidade de lidar com regras de primeira ordem através do novo método de proposicionalização desenvolvido, Proposicionalização de Cláusulas Mais Específicas (BCP), que transforma cada exemplo de primeira ordem em cláusulas mais específicas e as usa como exemplos proposicionais. Cláusulas mais específicas são as fronteiras do espaço de busca de hipóteses de primeira-ordem que delimitam o numero máximo de literais que uma hipótese candidata pode possuir. Para comparações experimentais, três outros sistemas serão utilizados: Aleph, um sistema padrão de ILP; RSD, um método bem conhecido de proposicionalização; e o sistema construtor de árvores de decisão C4.5. Vários aspectos serão avaliados empiricamente: acurácia da classificação com medidas de tempo de execução; análise de seleção de atributos; como BCP se sai em comparação com o RSD; e análise de robustez a ruído multiplicativo. Os resultados mostram que: CILP++ obteve acurácia de classificação competitiva com Aleph, mas é geralmente mais rápido; com seleção de atributos, mais de 90% dos atributos podem ser eliminados com pouca perda em acurácia; os exemplos proposicionalizados com BCP obtiveram maior acurácia do que RSD quando aplicados na rede neural do CILP++ e acurácia equivalente quando aplicados no C4.5, mas em ambos os casos, BCP foi mais rápido; e que CILP++ não é robusto a ruído multiplicativo se comparado ao Aleph, porém Aleph mostrou um decaimento maior de acurácia maior do que CILP++ com o aumento da intensidade do ruído.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

RELATIONAL LEARNING BY USING BOTTOM CLAUSES INTO C-IL^{2}P

Manoel Vitor Macedo França

December/2012

Advisor: Gerson Zaverucha

Department: Systems Engineering and Computer Science

In this work is introduced a fast algorithm for relational learning by using a novel propositionalization method and learning with artificial neural networks by extending Garcez and Zaverucha neural-symbolic system C-IL^{2}P, named CILP++, which uses a background knowledge encoded as a propositional logic program to build a recurrent artificial neural network and learn from examples with back-propagation. CILP++ incorporates a number of improvements, as well as the capability of handling first order rules through a novel propositionalization method, Bottom Clause Propositionalization (BCP), which transforms each first-order example into bottom clauses and use them as propositional patterns. Bottom clauses are most-specific first-order hypothesis search boundaries which delimit the maximum literal number that a candidate hypothesis can have. For experimental comparisons, three other systems will be used: Aleph, as a standard ILP system; RSD, a well-known propositionalization method; and the C4.5 decision tree learner. Several aspects will be empirically evaluated: standard classification accuracy and runtime measurements; feature selection analysis; how BCP performs as a propositionalization method, in comparison with RSD; and robustness to multiplicative noise. The results show that: CILP++ has competitive classification accuracy if compared with Aleph, but it is generally faster; feature selection can successfully prune over 90% of features with only little accuracy loss; the examples which are propositionalized with BCP obtained better accuracy results than RSD when applied in CILP++'s neural network and both performed equally well when using the C4.5 learner, but on both cases, BCP was faster; and that CILP++ is not robust to multiplicative noise if compared to Aleph, even that Aleph showed higher accuracy loss than CILP++ when noise level started to go up.

Sumário

Capítulo 1

Introdução

Conexionismo e simbolismo são dois dos principais paradigmas da Inteligência Artificial [14]: o primeiro engloba sistemas que aprendem através da simulação de algumas características do processamento cerebral e o segundo contém sistemas que aprendem em um nível maior de abstração, através de uma linguagem formal e de algoritmos de construção de hipóteses. O que tem sido observado sobre suas vantagens e desvantagens [33] é que eles são, de certa forma, complementares: o que é excelente em um, não é no outro, e vice-versa. Sistemas neuro-simbólicos [15, 19, 61] visam explorar esta complementaridade: combinando algoritmos e técnicas de ambos os paradigmas de uma forma vantajosa, tenta-se desenvolver um sistema híbrido capaz de combinar suas qualidades e suprimir suas falhas.

Neste trabalho, apresentamos um algoritmo para Programação em Lógica Indutiva (ILP) [34] utilizando Redes Neurais Artificiais (ANN's) [52], através da extensão de um sistema neuro-simbólico chamado C-IL^{2}P [16], que é capaz de usar conhecimento prévio, codificado como um programa lógico proposicional, para construir uma ANN recorrente e aprender a partir de exemplos usando retro-propagação. A implementação desta extensão recebeu o nome CILP++ e é um sistema de código aberto, hospedado e disponível no *sourceforge* (`https://sourceforge.net/projects/cilppp/`), assim também como o C-IL^{2}P original, disponível em (`https://sourceforge.net/projects/cil2p/`), ambos sob licença Apache 2.0.

Uma ANN é um modelo conexionista de aprendizado de máquina que busca simular a atividade sináptica entre neurônios para aprender e inferir: quando um neurônio recebe um conjunto de sinais de entrada de outros neurônios, as intensidades dos sinais recebidos são somadas, a sua função interna de ativação é aplicada a esse total e verifica-se se o resultado é suficiente para provocar um estado excitatório no neurônio em questão, comparando esse total com seu valor limite de ativação (*threshold*). Se o resultado for o suficiente para provocar um estado excitatório, o neurônio emite uma sinal em suas terminações de saída, que depende do tipo de função de ativação utilizada, para os neurônios vizinhos que estão conectados a elas. Caso não seja suficiente, esse neurônio se man-

tém em um estado inibitório e nenhum sinal é emitido. Todo o conhecimento de uma ANN é armazenado em dois locais: os pesos de suas conexões e seu *threshold*. Porém, o *threshold* pode ser visto como uma ligação com entrada fixa 1, reduzindo os locais de armazenagem de conhecimento de uma ANN a apenas seus pesos.

A forma como uma ANN aprende é através de pequenas mudanças em seus pesos. O método mais comumente usado para o aprendizado de uma ANN é o *método do gradiente*, onde uma fração do gradiente de uma função de erro que reflete quão próxima a resposta da rede está, em comparação com a resposta desejada, é usada para atualizar os pesos da rede iterativamente. O aprendizado ILP tradicional, por sua vez, é feito em nível conceitual, representando exemplos e hipóteses através de linguagens lógicas. A forma de aprendizagem da ANN é uma forma natural de lidar com uma forte crítica contra ILP, que é a respeito de sua "fragilidade" (*brittleness*): sistemas ILP têm dificuldades de aprendizagem quando enfrentam uma perturbação inesperada que levem-os, mesmo que apenas um pouco, fora do que é definido por seus conhecimentos prévios [1]. Quanto a perturbações nos exemplos (ruído), sabe-se que a ANN possui robustez contra ruído aditivo (ruído na entrada), o que significa que, se o valor original de uma entrada é um pouco diferente do que deveria ser, ainda assim, a ANN será capaz de aprender adequadamente. No entanto, esta afirmação não é necessariamente verdade quando se considera ruído multiplicativo (ruído na classificação): é preciso saber o nível de ruído nos rótulos e a correlação verdadeira entre entradas e conceitos alvo antecipadamente, para se configurar um modelo adequado [7].

Aprendizado relacional com ANN ainda é um problema em aberto [63] e há também uma grande diversidade de sistemas híbridos que lidam com dados relacionais através de modelos numéricos, como Redes Lógicas de Markov [51] (Campos Aleatórios de Markov + ILP), ILP Bayesiana [35] (Redes Bayesianas + ILP) e SVILP [40] (Máquinas de Vetores Suporte + ILP). Nossa escolha em usar uma ANN para aprender dados relacionais é principalmente devido à sua simplicidade e ao trabalho anterior do C-IL^{2}P, que conseguiu de forma eficiente e precisa aprender programas lógicos proposicionais usando uma ANN recursiva. Este trabalho visa alcançar este sucesso também com dados relacionais em primeira-ordem, através do sistema CILP++.

CILP++ implementa um método novo de proposicionalização [23], chamado Proposicionalização de Cláusula mais Específica (BCP), para transformar um problema relacional em um proposicional e, assim, tornar possível o aprendizado com o mesmo modelo neural do sistema C-IL^{2}P. Cláusulas mais específicas são as fronteiras do espaço de busca de hipóteses de primeira-ordem que delimitam o numero máximo de literais que uma hipótese candidata pode possuir, introduzidas pelo sistema Progol [37], que usa um exemplo aleatório, o conhecimento prévio (um conjunto de cláusulas que descrevem o que é conhecido a priori com respeito a um determinado problema) e viés de linguagem (um conjunto de cláusulas que delimitam como os candidatos a hipótese poderão ser formados) para

construí-los. Por isso, uma forma alternativa de se definir uma cláusula mais específica é dizer que elas são a representação mais extensa possível de um exemplo ILP, levando em consideração o conhecimento prévio e o viés de linguagem. Por isso, deve ser possível usá-las como padrões de treino, em vez de lidar diretamente com exemplos de primeira ordem. Mas por que usá-las dessa forma? A primeira resposta é que os literais da cláusula mais específica podem ser usados diretamente como atributos em uma tabela-verdade de proposicionalização, como mostrado em [42], o que significa que elas tornam possível o aprendizado relacional em modelos proposicionais. A segunda resposta é que cláusulas mais específicas possuem um valor semântico interno a elas, que será ilustrado e explicado na seção seguinte, juntamente com um exemplo.

1.1 Motivação

Considere a descrição da relação familiar "sogra" (*mother-in-law*) abaixo:

Conhecimento prévio:	**Exemplos Positivos:**
parent(mom1, daughter11)	*motherInLaw(mom1, husband1)*
wife(daughter11, husband1)	**Exemplos Negativos:**
wife(daughter12, husband2)	*motherInLaw(daughter11, husband2)*

O conceito-alvo, *motherInlaw*, pode ser descrito pelas relações *parent* e *wife*: pode-se notar que a relação entre *mom1* e *husband1* que o exemplo positivo *motherInLaw(mom1, husband1)* estabelece pode ser explicada pela sequência de fatos *parent(mom1, daughter11)* e *wife(daughter11, husband1)*. Semanticamente, ela afirma que *mom1* pode ser considerada como sogra porque ela tem uma filha casada, *daughter11*.

Com relação a sistemas relacionais, esta idéia geral pode ser aprendida de muitas maneiras diferentes, dependendo da abordagem de aprendizagem escolhida, tais como vinculação inversa (*inverse entailment*) [37] e resolução inversa (*inverse resolution*) [36]. Considerando *Progol*, um sistema conhecido na literatura que é baseado em vinculação inversa, o primeiro passo que ele iria tomar a fim de tentar chegar à mesma conclusão sobre a relação *mother-in-law* seria limitar o conjunto de possíveis explicações que poderiam ser usadas para explicá-la (ou seja, limitar o espaço de busca de hipóteses), através da definição de fronteiras de busca.

Fronteiras de busca de hipótese em Progol são definidas por duas cláusulas especiais: a cláusula mais geral e a cláusula mais extensa (saturada) que pode ser construída com o conhecimento prévio, chamada cláusula mais específica ($\perp_e$). O pequeno e na notação é para enfatizar que a cláusula mais específica é gerada através da saturação de um exemplo e portanto, diferentes exemplos podem gerar cláusulas mais específicas diferentes. Uma possível saturação para o exemplo positivo *motherInLaw(mom1, husband1)* poderia ser a

cláusula [$motherInLaw(A, B) \leftarrow$ *parent(A, C)*, *wife(C, B)*], por exemplo.

Verificando-se o exemplo de cláusula mais específica [$motherInLaw(A, B) \leftarrow$ *parent(mom1, daughter11)*, *wife(daughter11, husband1)*], pode-se perceber que ela descreve uma possível definição de sogra. Ela afirma que "A é sogra de B se A é mãe de C e C é esposa de B", o que resumidamente significa que a mãe de uma filha casada é uma sogra. Este exemplo mostra que as cláusulas mais específicas são capazes de possuir significado, motivando este trabalho a pesquisar a possibilidade de usá-las como exemplos de treinamento no aprendizado, em vez de apenas fronteiras de busca de hipótese.

1.2 Contribuições

As duas principais contribuições deste trabalho são:

- **O novo método de proposicionalização, BCP.** Foi desenvolvido um método novo de proposicionalização que é usado pelo CILP++ como uma etapa de pré-processamento extra, se comparado ao C-IL^{2}P, antes de construir um modelo inicial com conhecimento prévio, chamado BCP. Nessa nova etapa, cláusulas mais específicas são geradas para cada exemplo de primeira ordem, uma tabela-verdade de proposicionalização que contém todos os literais de corpo encontrados em cada cláusula gerada como sendo colunas é construída e todos os exemplos são convertidos em vectores binários.

- **o sucessor do sistema C-IL^{2}P, CILP++.** Vários gargalos de eficiência no C-IL^{2}P foram otimizados, como seus critérios de parada que exigiam verificações demoradas de ativação de cada neurônio e sua função de ativação, que antes era a mesma para todos os neurônios e agora é diferente para cada neurônio. É interessante haver diferentes critérios de ativação para neurônios de diferentes camadas, por exemplo, pois eles correspondem a diferentes partes do conhecimento prévio. Além dessas duas otimizações, uma normalização de pesos que mantém a integridade do conhecimento prévio embutido na rede também foi implementada. Esta é uma otimização importante em relação ao C-IL^{2}P porque ela permite um comportamento adequado da rede, independentemente da natureza do conjunto de dados utilizado, o que antes não era possível com o C-IL^{2}P, devido aos valores iniciais consideravelmente grandes que eram atribuídos aos pesos ao se calcular seus pesos iniciais e limiares de ativação [18].

Adicionalmente, ambas as implementações do C-IL^{2}P e CILP++, para lidar com problemas proposicionais e relacionais, respectivamente, estão disponibilizadas no *sourceforge*, com código aberto, e também são contribuições desse trabalho.

1.3 Objetivos

Este trabalho usa a fronteira mais específica de busca de hipótese, $\perp_e$, para treinar um sistema neuro-simbólico de primeira-ordem chamado CILP++, que é baseado na versão proposicional C-IL^{2}P [16]. Como $\perp_e$ é uma representação extensa de um exemplo, um sistema capaz de lidar adequadamente com os dados redundantes e representações de entrada extensas pode ser apto a aprender com ela. O sistema C-IL^{2}P, que utilizou uma rede neural recorrente para lidar com programas lógicos proposicionais, é um candidato nato para essa tarefa. Iremos gerar uma cláusula mais específica para cada exemplo, tratar cada literal como um átomo proposicional para o CILP++ e depois executar seu algoritmo de treinamento para aprender as características em comum que exemplos de mesma classe possuem.

Os resultados experimentais cobrirão algumas das perguntas que podem surgir ao apresentar a abordagem implementada nesse trabalho:

- O sistema CILP++ é competitivo com outros sistemas ILP em termos de acurácia e tempo de treinamento?
- A nova proposicionalização desenvolvida, BCP, é comparável com outras do estado-da-arte?
- Uma melhor seleção de atributos (filtragem) pode trazer benefícios aos resultados atuais?

Adicionalmente, como parte dos objetivos desse trabalho, todos os experimentos e resultados obtidos aqui estão disponibilizados e poderão ser reproduzidos em (`http://vega.soi.city.ac.uk/~abdz937/bcexperiments.zip`), junto com a listagem de todos os parâmetros e configurações usadas.

1.4 Organização

Este trabalho foi inicialmente escrito em inglês e cada um dos capítulos apresentados consistirão de uma versão sucinta em português que referenciará seu correspondente capítulo, em inglês, no apêndice.

Começando pelo Capítulo 2, as duas áreas principais envolvidas neste trabalho, ILP e ANN's, serão brevemente introduzidas. Duas técnicas geralmente utilizadas quando se trabalha com eles serão introduzidas também: proposicionalização e seleção de atributos, respectivamente. Esse capítulo será concluído com uma revisão de um sistema que lida com dados proposicionais, alternativo à ANN desenvolvida nessa dissertação, que será experimentado no capítulo 5: árvores de decisão.

No Capítulo 3, uma introdução a sistemas neuro-simbólicos será feita e o sistema C-IL^{2}P será apresentado. Todos os passos do ciclo de aprendizado do C-IL^{2}P serão des-

critos: a construção de um modelo inicial com um programa lógico proposicional que representa o conhecimento prévio; o refinamento deste modelo com a aprendizagem a partir de exemplos usando retro-propagação; a inferência de dados desconhecidos com a rede resultante; e a extração do que foi aprendido, para se obter um programa proposicional revisado e que representa a combinação do conhecimento prévio com os exemplos apresentados durante o treinamento. Ao final deste capítulo, algumas aplicações do C-IL^{2}P serão apresentadas, discutindo seus domínios e os resultados obtidos.

No Capítulo 4 apresentamos as duas contribuições deste trabalho: BCP e CILP++. Uma explicação completa do que exatamente consiste CILP++, o que faz e como, será dada. Esse capítulo começará descrevendo a etapa de pré-processamento do CILP++, a conversão BCP. Depois disso, seu algoritmo de construção será explicado, seguido por seu treinamento e como se infere dados desconhecidos. A fim de completar o ciclo de aprendizagem do C-IL^{2}P, a extração do conhecimento aprendido durante o treinamento é deixada como trabalho futuro. Para concluir o capítulo, alguns dos principais trabalhos relacionados à nossa abordagem são citados e brevemente apresentados.

No Capítulo 5, todos os experimentos realizados serão apresentados. Dois *benchmarks* de ILP foram utilizados: os conjuntos de dados *Alzheimer* [20] e o conjunto de dados *Mutagenesis* [57]. Os experimentos irão cobrir os seguintes aspectos: acurácia e tempo de execução, com validação cruzada; desempenho ao se fazer seleção de atributos; desempenho comparativo com um método estado-da-arte de proposicionalização, RSD [65], usando uma ANN e um sistema construtor de árvores de decisão; e desempenho na presença de ruído multiplicativo. Mais especificamente a respeito de seleção de atributos, uma vez que o BCP é baseado na geração de cláusulas mais específicas para cada exemplo e elas são representações extensivas, o exemplo proposicionalizado acaba possuindo dimensionalidade consideravelmente elevada e um método de seleção de atributos possivelmente melhorará o desempenho da nossa abordagem. O sistema ILP estado-da-arte escolhido para se realizar as comparações foi o Aleph [55] e o sistema construtor de árvores de decisão que será usado como uma alternativa para aprender dados proposicionais, para complementar os resultados comparativos com o RSD, é o sistema C4.5 [49].

Finalmente, o Capítulo 6 concluirá esse trabalho revisando o que foi apresentado em cada capítulo anterior, fazendo alguns comentários adicionais sobre resultados obtidos e discutirá alguns trabalhos futuros, incluindo experimentos com conjuntos de dados diferentes e a completude do ciclo de aprendizagem do C-IL^{2}P no CILP++, com a implementação de extração de conhecimento na rede pós-treinamento.

A versão completa da dissertação em inglês em formato contínuo (apêndices referentes a capítulos nos seus lugares apropriados) encontra-se em `http://soi.city.ac.uk/~abdz937/dissertation.pdf` e todos os experimentos realizados, com as bases de dados utilizadas e resultados experimentais, podem ser baixados em `http://soi.city.ac.uk/~abdz937/bcexperiments.zip`.

Capítulo 2

Conceitos Preliminares

Neste capítulo, ambas as áreas de Aprendizado de Máquina [32] que são utilizadas neste trabalho (Programação em Lógica Indutiva e Redes Neurais Artificiais) serão revisadas, com foco em tópicos que mais se relacionam com este trabalho. Em seguida, as técnicas de seleção de atributos e proposicionalização serão brevemente explicadas. No final desse capítulo, uma breve introdução a um dos modelos que serão usados nos experimentos, árvores de decisão, será feita. Este capítulo apresenta apenas os conceitos básicos de cada tópico e de cada técnica listada acima e uma descrição mais completa, com todas as definições necessárias, encontra-se no Apêndice A.

2.1 Programação em Lógica Indutiva

Programação em Lógica Indutiva [41] é uma sub-área de Aprendizado de Máquina que faz uso de linguagens lógicas para induzir hipóteses estruturadas em teorias e consultando-as, pode-se realizar inferência em dados desconhecidos. Dado um conjunto de exemplos rotulados E e um conhecimento preliminar B, um sistema de ILP vai tentar encontrar uma hipótese H, que minimiza uma função de perda especificada. Mais precisamente, um problema em ILP é definido como $<$ E, B, L $>$, onde E = $E^+ \cup E^-$ é um conjunto que contém cláusulas positivas (E^+) e negativas (E^-), conhecidas como *exemplos*; B é um programa lógico chamado *conhecimento prévio*, que é composto por *fatos* e *regras*; e L é um conjunto de teorias lógicas chamado *viés de linguagem*.

O conjunto de todas as hipóteses possíveis para um determinado problema, que vamos chamar S_H, pode ser infinito [37]. Um dos elementos que restringe S_H em ILP é o viés de linguagem, L. Ele geralmente é composto por predicados de especificação, que definem como a busca será feita e até onde ela pode ir. A linguagem de especificação mais comumente usada é composta por dois tipos de viés: *declaração de modo*, contendo predicados *modeh*, que define o que pode aparecer como cabeça de uma cláusula e predicados *modeb*, que definem o que pode aparecer no corpo de uma cláusula; e *determinantes*, que rela-

cionam literais de corpo e de cabeça. As declarações de modo *modeb* e *modeh* também especificam o que é considerado ser uma *variável de entrada*, uma *variável de saída* e uma *constante*. O viés de linguagem L, através dos predicados de declaração de modo e determinantes, podem restringir S_H durante a busca de hipótese para permitir que apenas um pequeno conjunto de hipóteses candidatas H_c sejam analisadas. Formalmente, H_c é uma hipótese candidata em um problema ILP $<$ E, B, L $>$ *sse* $H_c \in L$, $b \cup H_c \models E^+$ e $B \cup H_c \not\models E^-$.

Outro elemento que é usado para restringir a busca no espaço de hipótese em algoritmos com base na *vinculação inversa* (*inverse entailment*) [37], como Progol, é a *cláusula mais específica (saturada)*, $\perp_e$. Dado um exemplo aleatório e, Progol primeiramente gera uma cláusula que representa e da forma mais específica possível, buscando em L por cláusulas *modeh* que podem unificar com e e caso alguma seja encontrada, uma cláusula mais específica inicial $\perp_e$ é criada. Em seguida, uma varredura é feita pelos predicados *determinantes* para verificar qual dos predicados definidos nas cláusulas *modeb* é o mais adequado para ser adicionado a $\perp_e$. Esse processo é feito repetidamente, até que um número máximo de ciclos (conhecido como *profundidade de variável*) tenha sido atingido, percorrendo as declarações *modeb*.

O leitor familiar com ILP e Prolog pode desconsiderar o Apêndice A.1 que contém essa seção de forma completa.

2.2 Redes Neurais Artificiais

Uma rede neural artificial é um grafo direcionado com a seguinte estrutura: uma unidade (ou neurônio) no grafo é caracterizado, no tempo t, por seu *vetor de entrada* $I_i(t)$, seu *potencial de entrada* $U_i(t)$, seu *estado de ativação* $A_i(t)$ e sua *saída* $O_i(t)$. As unidades da rede estão interligadas através de um conjunto de conexões direcionadas e ponderadas de tal forma que, se houver uma conexão vinda da unidade i para a unidade j então $W_{ji}\ in\mathbb{R}$ denota o *peso* desta ligação. O potencial de entrada do neurônio i no tempo t ($U_i(t)$) é obtido calculando-se uma soma ponderada para o neurônio i tal que $U_i(t) = \sum_j W_{ij} I_i(t)$. O estado de ativação $A_i(t)$ do neurônio i no tempo t é dado pela *função de ativação* h_i tal que $A_i(t) = h_i(U_i(t))$. Além disso, b_i (um peso extra com entrada sempre fixada em 1) é conhecido como *viés* do neurônio i. Define-se que um neurônio i é *ativo* no momento t se $A_i(t) > -b_i$. Por fim, o valor do neurônio de saída é dado por seu estado de ativação $A_i(t)$.

Para o aprendizado, *retro-propagação* (*back-propagation*) [52] é o algoritmo mais utilizado, baseado no método do gradiente. Ele visa minimizar uma função de erro $\mathbf{E}$, que mede a diferença entre a resposta da rede e classificação do exemplo. Quanto aos critérios de parada, *treinamento padrão* e *parada antecipada* (*early stopping*) [18] são os dois mais utilizados. Em *treinamento padrão*, o conjunto completo de dados de treinamento é

usado para minimizar $\mathbf{E}$, enquanto que *parada antecipada* usa um conjunto de validação definido criado para medir *overfitting* de dados [5]: o treinamento é interrompido quando o erro do conjunto de validação começa a ficar mais elevado do que os passos anteriores. Quando isso acontece, a melhor configuração de validação obtida, até agora, é usada como o modelo aprendido.

A versão completa dessa introdução a redes neurais artificiais encontra-se no Apêndice A.2.

2.3 Seleção de Atributos

Tal como indicado na seção 2.1, cláusulas mais específicas são representações extensivas de um exemplo, possivelmente tendo tamanho infinito [37]. Por isso, a fim de reduzir seu tamanho, há duas abordagens que podem ser tomadas: reduzi-la durante sua geração ou usando uma abordagem estatística depois. A primeira opção pode ser feita dentro do algoritmo de geração da cláusula mais específica [37] (um pseudo-algoritmo para essa geração pode ser encontrado no Algoritmo A.2), através da redução da profundidade de variável. Ao limitar a quantidade de ciclos que o algoritmo pode passar através das *declarações de modo*, é possível eliminar uma porção considerável de literais, embora causando perda de informação. A segunda opção para se realizar seleção de atributos, por um outro lado, é através de métodos estatísticos, tais como a correlação de Pearson e Análise de Componentes Principais (uma revisão comparativa desses métodos pode ser encontrada em [30]). Um método do estado-da-arte interessante, que tem complexidade, custo e tamanho reduzidos e supera a maioria dos métodos comuns em termos de perda de informação é o algoritmo mRMR [10], que se propõe a encontrar um meio-termo entre mínima redundância e máxima relevância dos atributos, selecionando-os através da métrica de *informação mútua* (I). A informação mútua de duas variáveis x e y é definida como

$$I(x, y) = \sum_{i,j} p(x_i, y_j) \log \frac{p(x_i, y_j)}{p(x_i)p(y_j)}, \tag{2.1}$$

onde $p(x, y)$ é a distribuição conjunta delas e $p(x)$ e $p(y)$ são as suas respectivas probabilidades marginais. Dado um subconjunto S do conjunto de atributos Ω a ser ranqueado pelo mRMR, as condições de mínima redundância e máxima relevância, respectivamente, são definidas por

$$min\, W_I,\ W_I = \frac{1}{|S|^2} \sum_{i,j \in S} I(i, j) \text{ e} \tag{2.2}$$

$$max\, V_I,\ V_I = \frac{1}{|S|} \sum_{i \in S} I(h, i), \tag{2.3}$$

onde $h = \{h_1, h_2, \ldots, h_K\}$ é a variável de classificação de um conjunto de dados com K classes possíveis.

Sejam Ω, S e $\Omega_S = \Omega - S$ o conjunto total de atributos, o conjunto de atributos que já foram selecionados pelo mRMR e o conjunto de atributos ainda não selecionados, respectivamente. Há duas maneiras de se combinar as duas condições apresentadas acima para se selecionar mais atributos em Ω_S para fazer parte de S: Diferença de Informação Mútua (MID), definida como $\max(V_I - W_I)$; e Quociente de Informação Mútua (MIQ), definida como $\max(V_I/W_I)$. O que difere MID e MIQ é o quanto que elas são sensíveis a pequenas alterações entre a máxima relevância e mínima redundância: a diferença entre V_I e W_I mantém-se linear no MID, mas no segundo, cresce exponencialmente e por isso, diferenciar atributos candidatos a serem incluídos em S é mais fácil. Além disso, os resultados mostrados em [10] indicam que MIQ geralmente escolhe melhores atributos em uma quantidade considerável de conjuntos de dados. Assim, esta última será a função escolhida para selecionar atributos neste trabalho e por motivos de simplicidade, se esse trabalho referir-se a mRMR, assuma que MIQ é a abordagem usada.

2.4 Proposicionalização

Proposicionalização é a conversão de uma base de dados relacional em uma tabela valor-atributo que viabiliza o aprendizado usando sistemas que lidam com dados proposicionais [23]. Algoritmos de proposicionalização usam o conhecimento prévio e os exemplos para encontrar características distintas, que podem diferenciar subconjuntos de exemplos.

Existem dois tipos de proposicionalização: *orientada a lógica* e *orientada a banco de dados*. O primeiro visa construir um conjunto de atributos de primeira-ordem que são relevantes para distinguir entre objetos de primeira-ordem e o segundo tenta explorar relações e funções de bancos de dados relacionais para gerar atributos para proposicionalização. Este trabalho introduz uma nova técnica de proposicionalização *orientada à lógica*, chamada Proposicionalização de Cláusulas Mais Específicas (BCP), que consiste na geração de cláusulas mais específicas para cada exemplo de primeira-ordem e usando o conjunto de todos os literais de corpo que ocorrem nelas como atributos possíveis (em outras palavras, como colunas para uma tabela valor-atributo). A fim de avaliar como esta abordagem se comportará, BCP será comparado com o algoritmo de proposicionalização do sistema RSD [65], bem conhecido na literatura.

RSD é um sistema que lida com o problema de *Descoberta de Subgrupo Relacional*: dada uma população de indivíduos e uma propriedade de interesse deles, deve-se encontrar subgrupos populacionais que são tão grandes quanto possível e que possuam as características mais incomuns de distribuição. Sua entrada é um conjunto de dados em formato semelhante ao Aleph, com conhecimento preliminar, conjunto de exemplos e

viés de linguagem. A saída é uma lista de cláusulas que descreve subgrupos interessantes dentro do conjunto de dados de exemplo. Ele é composto por duas etapas: construção de atributos de primeira-ordem e indução de regras. O primeiro é um método que cria atributos de alto nível que são usados para substituir grupos de literais de primeira-ordem e o último é uma extensão do aprendedor de regras proposicionais CN2 [6], para torná-lo capaz de ser usado como um solucionador do problema que RSD está tentando resolver.

Apenas a parte de proposicionalização do sistema RSD é de interesse para este trabalho (a primeira parte das duas citada acima), que é composta por três etapas:

1. Todas as expressões que, por definição, formam uma função de primeira ordem e, ao mesmo tempo, estão de acordo com as declarações de modo são identificados;
2. Constantes são empregadas através da cópia de alguns atributos, definidas pelo usuário, para instanciar alguns predicados. Em seguida, atributos que não aparecem em nenhum exemplo, ou em todos, são eliminados do conjunto final de atributos;
3. Uma representação proposicionalizada de cada exemplo é feita.

Para rodar experimentos com RSD, iremos usar a ferramenta de proposicionalização com RSD disponibilizada pelo Filip Železný em (`http://labe.felk.cvut.cz/~Zelezny/rsd`). A partir de agora, quando RSD for mencionado, o método de proposicionalização RSD estará sendo referido, ao invés do sistema de descoberta de subgrupos relacionais RSD, como um todo.

2.5 Árvores de Decisão

Árvores de Decisão [53] são modelos que visam criar uma árvore que prevê o valor de uma variável de classe com base em um conjunto de variáveis de entrada. Cada nó interior corresponde a uma das variáveis de entrada e cada ramificação saindo desse nó corresponde a um possível valor dessa variável de entrada. Por fim, cada folha representa um valor da variável de classe dado os valores das variáveis de entrada representados pelo caminho da raiz a ele. A árvore pode ser "aprendida" através da divisão do conjunto de dados em subconjuntos, baseados em teste de atributo-valor. Este processo é repetido em cada subconjunto derivado, recursivamente, até todos os subconjuntos de dados de um nó possuirem o mesmo valor da variável de classe, ou quando a divisão já não acrescentar valor às previsões. Este processo *top-down* de indução de árvores de decisão [48] é um exemplo de um algoritmo guloso e é a estratégia mais comum para a aprendizagem de árvores de decisão a partir de dados, apesar da possibilidade também de se usar abordagens *bottom-up* [3].

As árvores de decisão podem ser divididas em dois tipos:

- Árvores de classificação, que analisam a quais classes um dado exemplo pertence;

- Árvores de regressão, cujas respostas são números reais e portanto, pode-se reduzir a dimensionalidade da entrada.

Mais especificamente sobre algoritmos de aprendizado de árvores de classificação, a idéia é construir uma árvore de decisão usando-se um conjunto de exemplos rotulados. Eles têm uma estrutura semelhante a um diagrama de fluxo, onde cada nó interno (não folha) indica um teste em um atributo, cada ramo representa um resultado do teste, e cada folha de nó (ou terminal) possui uma classe etiquetada. Dentre as árvores de decisão de classificação mais comuns na literatura, duas delas serão introduzidas a seguir: ID3 (Iterative Dichotomiser 3) [48] e seu sucessor, C4.5 [49].

ID3 é uma árvore de decisão de classificação *top-down*, que usa uma métrica chamada *ganho de informação* para decidir entre os atributos que serão escolhidos como pontos de divisão ao avaliar os dados. O algoritmo básico (*greedy*), resumidamente, é: a partir do exemplo com o maior ganho de informação (que será o rótulo raiz da árvore), o conjunto de exemplos irão ser divididos em subconjuntos, um para cada valor possível do atributo, gerando caminhos para cada um deles. Os próximos nós filhos de cada caminho são escolhidos novamente através do ganho de informação e isso continuará a ser feito de forma iterativa. Quando os exemplos cobertos por um determinado caminho tiverem classes idênticos, esse caminho não será mais expandido. O algoritmo pára quando todos os caminhos pararam de gerar novos nós. C4.5 constrói árvores de decisão a partir de um conjunto de dados de treino da mesma forma que ID3, utilizando ganho de informação, mas implementa várias melhorias:

- Ele pode lidar com atributos contínuos e discretos. Para processar atributos contínuos durante o treinamento, C4.5 cria um limiar e depois divide a lista de exemplo em aqueles que possuem valores superiores e aqueles que possuem valores inferiores a esse atributo;
- Ele também é capaz de lidar com dados de treinamento com valores de atributos ausentes. C4.5 permite que os valores dos atributos possam ser marcados como "?" (faltando). Atributos com valores faltando apenas não são utilizados no cálculo do ganho de informação;
- É possivel trabalhar com ponderações nos atributos (ele é capaz de associar um peso a alguns valores de atributos, dando-lhes mais importância se eles aparecerem em um padrão);
- Por fim, ele é capaz de podar árvores após a criação. Uma consulta inversa (das folhas à raiz) é feita na árvore pós-treinamento e galhos que não ajudam na inferência de dados desconhecidos são removidos, substituindo-os por nós folha. A decisão de se podar ou não um galho é feita através de um limite inferior sobre o ganho de informação.

C4.5 será escolhido para este trabalho como um algoritmo alternativo para o aprendizado de cláusulas mais específicas, juntamente com o ANN recursivo do C-IL^{2}P.

Capítulo 3

Sistemas Neuro-Simbólicos

Neste capítulo, uma rápida introdução a sistemas neuro-simbólicos e seus principais aspectos será feita, seguida por uma explicação geral sobre o sistema C-IL^{2}P [16], no qual esse trabalho se baseou. Essa revisão cobrirá as quatro partes de seu ciclo de aprendizado: construção de um modelo inicial, treinamento, inferência de dados desconhecidos e extração de conhecimento. A versão completa desse capítulo, em inglês, encontra-se no apêndice B.

3.1 Introdução

Quando um pai está ensinando sua filha um novo conceito como, por exemplo, como beber um copo de água sozinha, há duas formas dele fazer isso. A primeira é explicar com palavras, ilustrações, imagens, etc, como agarrar um copo vazio, como preenchê-lo com água pura e como usá-lo corretamente. A outra abordagem (e talvez mais fácil) é mostrar para a filha como se faz, bebendo um copo de água na frente dela. Ao se fazer isso repetidamente, modificando apenas alguns detalhes tais como o tipo de copo usado e a fonte da qual ele coletou água, ela será capaz de imitar a maioria dessas ações e aprender pequenas variações sozinha.

O que está descrito acima é as duas maneiras mais tradicionais em que os seres humanos aprendem: descrevendo mentalmente como uma situação de aprendizado se encaixa em uma categoria de situações conhecidas e analogicamente associar suas respostas corretas para reagir adequadamente a esta nova aplicação (por exemplo, mostrando simbolicamente como situações como “pegar um copo” ou “preencher copos com líquidos” pode se encaixar na tarefa de beber água), ou mostrando exemplos de respostas corretas para o problema dado (similar a ensinar uma criança como beber água, mostrando para ela diferentes formas de se fazer isso, de modo que ela pode replicar os movimentos sozinha quando necessário).

No cenário fictício descrito, havia uma abundância de informações disponíveis: o pai

tinha conhecimento prévio e exemplos suficientes de ações corretas de como se beber "um copo de água". Isso não é o que normalmente acontece em problemas reais: alguns problemas vão se encaixar melhor com a primeira abordagem mostrada acima, conhecida como *Aprendizado Baseado em Explicação* [54], caracterizada por carregar muitas descrições formais e fatos, mas sem um conjunto completo de exemplos que cubram todas as situações possíveis. Por outro lado, existem os tipos de problemas que são descritos apenas por exemplos empíricos de soluções corretas e incorretas, que geralmente são representados por números e não detém descrições simbólicas formais. Esses tipos de problemas são conhecidos como *empíricos*. Mais formalmente, essas duas classes de problemas e abordagens são campos de pesquisa em duas áreas:

- O primeiro (aprendizado baseado em explicação) é explorado principalmente por sistemas relacionais e programas lógicos proposicionais, que tentam gerar teorias que possam explicar uma determinada tarefa e responder apropriadamente se uma tarefa semelhante aparecer;
- O segundo (aprendizado empírico) é explorado principalmente por conexionistas e estatísticos, através de modelos que aproximam curvas geométricas e regiões que melhor se ajustam ao conjunto de exemplos dado, possibilitando classificação e agrupamento de dados sem qualquer conhecimento formal do modelo obtido.

Mas e se um *modelo baseado em explicação* necessitar de aprendizado paralelo (aprendendo múltiplos conceitos ao mesmo tempo)? E o que poderia ser feito se um sistema *empírico* precisasse de extração de conhecimento, para que se possa descobrir o que foi aprendido? Estas perguntas iniciaram uma pesquisa que culminou em uma nova área em Inteligência Artificial: *sistemas neuro-simbólicos*. O que se espera é que a integração de modelos baseados em *aprendizado baseado em explicação* com modelos *empíricos* habilite suprimir as falhas dos dois sistemas, quando usados individualmente, mantendo suas vantagens. Diferentes técnicas têm sido usadas como fontes para novos sistemas híbridos [14], incluindo dois comumente usados em *Aprendizado de Máquina*: *Redes Neurais Artificiais* e *Programação em Lógica*. O sistema neuro-simbólico C-ILP^2 [16] foi feito a partir deles e é um sistema híbrido que obteve resultados de classificação sólidos em comparação a ANN e programação em lógica, quando usados separadamente.

3.2 C-IL^{2}P

Tendo introduzido sistemas neuro-simbólicos, o sistema C-IL^{2}P [16] (*Connectionist and Inductive Learning and Logic Programming*) será agora apresentado. Ele é um sistema que integra um programa lógico proposicional representando conhecimento prévio e uma ANN recorrente da seguinte forma: primeiramente, ele constrói uma ANN com pesos fixos recorrentes, usando um conhecimento prévio composto por cláusulas proposicionais

(**fase de construção**); em seguida, aprende com exemplos usando retro-propagação simples, sem parada antecipada e ignorando as conexões recursivas (*fase de treinamento*); depois de treinado, está apto a fazer inferência sobre dados desconhecidos consultando a ANN pós-treinamento; e por último, o conhecimento aprendido pode ser extraído [13] dele para se obter a nova teoria que foi aprendida durante o treinamento (**fase de extração de conhecimento**). No Apêndice E, algoritmos para todas as fases do C-IL^{2}P que são relacionadas a este trabalho são apresentados.

A Figura 3.1 ilustra a primeira fase, a construção, e mostra como construir uma ANN recursiva **N** utilizando-se um conhecimento de prévio **B**.

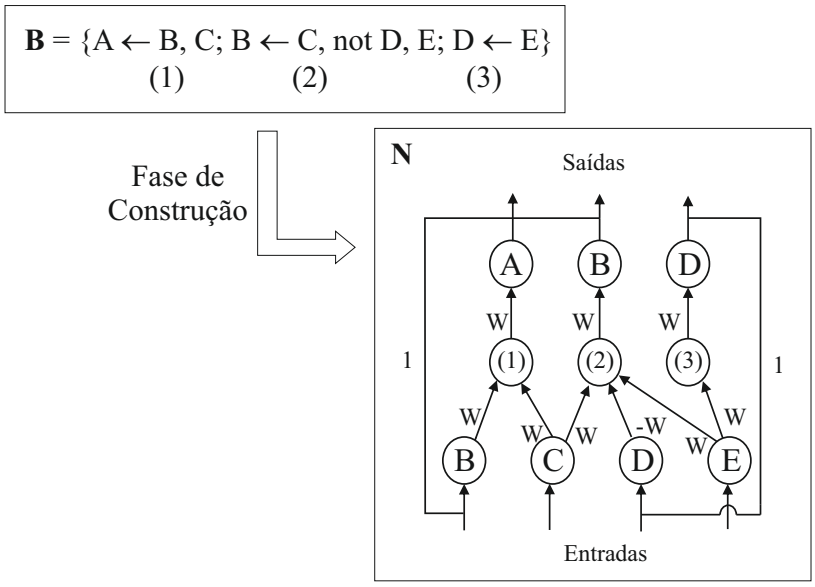

Figura 3.1: Exemplo da fase de construção do C-IL^{2}P. A partir de um conhecimento prévio **B** e uma rede vazia, C-IL^{2}P cria um neurônio na camada oculta para cada cláusula de **B** (cláusulas 1, 2 e 3). Assim, C-IL^{2}P terá três neurônios ocultos. Para as camadas restantes: uma vez que cada neurônio oculto está relacionado a uma cláusula, cada um dos literais do seu corpo vai estar relacionado a um neurônio de entrada e seu literal da cabeça vai se tornar um neurônio de saída. Para a cláusula 1, uma vez que ela tem dois literais de corpo, dois neurônios de entrada serão criados para representá-los e após isso eles vão ser conectados ao neurônio oculto correspondente à sua cláusula original. Ambos serão conectados a ele com um peso positivo calculado W. Se um literal de corpo é negativo, sua conexão com o neurônio oculto correspondente à cláusula a qual pertence será $-W$. Por fim, neurônios de entrada e saída que mapeiam literais idênticos possuirão uma conexão recursiva com peso fixo 1. Depois de repetir este processo para as outras duas cláusulas de **B**, a rede resultante **N** é como a mostrada na figura.

Além da estrutura e os valores de peso, C-IL^{2}P também calcula os valores de viés para todos os neurônios. Ambos W e os viés são funções de um terceiro parâmetro, A_{min}: esse parâmetro controla a ativação de todos os neurônios em C-IL^{2}P, permitindo apenas a ativação se a condição mostrada na equação 3.1 a seguir for satisfeita, onde W_i é um peso da rede, x_i é uma entrada, b é o viés do neurônio e h_n é a função de ativação de neurônio

n, que é *linear* se é um neurônio de entrada e *semi-linear bipolar*[1], caso contrário.

$$h_n(\sum_{\forall i} w_i \cdot x_i + b) \geq A_{min} \tag{3.1}$$

Após a fase de construção, o treinamento pode ser iniciado. Opcionalmente, antes de começá-lo, mais neurônios podem ser adicionados na camada oculta se necessário (para aproximar melhor os dados de treinamento), conectando-os com todos os neurônios das outras camadas. O algoritmo de treinamento utilizado é retro-propagação padrão, tendo taxa de aprendizado como único parâmetro. Ele também ignora as conexões recursivas quando está treinando: essas conexões só são usadas para inferência, para consultar fatos que são conseqüências de outros fatos.

Após o treinamento, inferência de dados desconhecidos pode ser feita e por último, a extração de conhecimento. O que o trabalho em [13] propõe como um algoritmo de extração de conhecimento para o C-IL^{2}P é uma divisão da rede treinada em sub-redes "regulares", que são caracterizadas por não possuirem conexões saindo de um mesmo neurônio e com sinais diferentes (positivo e negativo). O trabalho mostra que a extração em redes regulares é correta e completa, mas caso a divisão em sub-redes tenha sido necessária (no caso de a rede não ser regular), apenas corretude pode ser garantida.

[1]Função de ativação semi-linear bipolar: $f(x) = \frac{2}{1+e^{-\beta.x}} - 1$, onde β controla a curvatura.

Capítulo 4

Aprendizado Relacional Através da Adição de Cláusulas Mais Específicas ao C-IL^{2}P

Neste capítulo, serão apresentadas as contribuições deste trabalho, o novo método de proposicionalização, BCP, e a extensão do sistema C-IL^{2}P, CILP++. Adicionalmente, trabalhos relacionados com a nossa abordagem também serão discutidos. Novamente, a versão completa em inglês deste capítulo pode ser encontrada no apêndice C.

4.1 CILP++

CILP++ é a implementação do trabalho apresentado até então: um sistema integrado que usa BCP para trazer o problema relacional ao nível proposicional e depois, uma versão melhorada do C-IL^{2}P é usada para o aprendizado com exemplos. Assim, o primeiro passo para se aprender com CILP++ é a aplicação do BCP no conjunto de exemplos. Cada predicado alvo instanciado (cada exemplo de primeira-ordem), como por exemplo $target(a_1, \ldots, a_n)$, será convertido em algo que uma rede neural possa usar como padrão de entrada. Para conseguir isso, em primeiro lugar, cada exemplo é transformado em uma cláusula mais específica, a qual contém todos os conceitos possíveis que podem ser relacionados a ela, como um ponto de partida para uma representação adequada e, depois, é diretamente mapeada como atributos de uma tabela-verdade e vetores numéricos são então gerados para cada exemplo. Assim, o BCP tem duas etapas: *geração de cláusulas mais específicas* e *mapeamento atributo-valor*.

No primeiro passo, geração de cláusulas mais específicas, o conjunto de exemplos é usado como entrada para o algoritmo de geração de cláusulas mais específicas[58] do sistema ILP Progol (introduzido no Capítulo 2.1) para transformá-los em cláusulas mais específicas. Para isso, uma pequena modificação no algoritmo original foi necessária.

Esta versão modificada é mostrada no Algoritmo C.1. Ela foi necessária por duas razões: para permitir que a mesma função *função* que associa variáveis a constantes possa ser compartilhada entre todos os exemplos, a fim de manter a consistência entre as associações de variáveis; e permitir que os exemplos negativos possam ser aceitos pelo algoritmo. O algoritmo original pode lidar apenas com exemplos positivos.

Se o Algoritmo C.1 for executado com $depth$ (profundidade de variável) = 1 com os exemplos mostrados na Seção 1.1, ele gerará o conjunto de cláusulas mais específicas abaixo (a geração dessas cláusulas, passo-a-passo, encontra-se no Capítulo A.1):

$$S_{\perp} = \{motherInLaw(A, B) : -parent(A, C), wife(C, B);$$
$$\sim motherInLaw(A, B) : -wife(A, C)\}.$$

Após a criação do conjunto $S_{\perp}$, é a vez de se executar o segundo passo do BCP: cada elemento de $S_{\perp}$ será convertido em um vetor numérico v_i, $0 \leq i \leq n$, para permitir que qualquer aprendedor proposicional possa processá-las. O algoritmo que foi implementado (em pseudo-código) segue abaixo.

1. Seja $|L|$ o número de literais de corpo distintos em $S_{\perp}$;
2. Seja S_v o conjunto de vetores de entrada, que conterá os exemplos convertidos de $S_{\perp}$ e inicialmente vazio;
3. Para cada cláusula mais específica $\perp_e$ de $S_{\perp}$ faça
   (a) Crie um vetor numérico v_i de tamanho $|L|$ e com 0 em todas as posições;
   (b) Para cada posição correspondente a um literal de corpo de $\perp_e$, mude seu valor para 1;
   (c) Adicione v_i a S_v;
4. Retorne S_v;

Como exemplo, suponha que o conjunto de cláusulas mais específicas $S_{\perp}$ será pré-processado pela segunda etapa do BCP. $|L|$ seria igual a 3, já que os literais encontrados nele são $parent(A, C)$, $wife(C, B)$ e $wife(A, C)$. Para a cláusula mais específica positiva, um vetor v_1 com tamanho 3 iria ser inicializado com 0 e em seguida, ele receberia valoração 1 em todas as posições correspondentes a literais encontrados em sua cláusula mais específica. Assim, a sua primeira posição, que corresponde a $parent(A, C)$, e a segunda posição, que corresponde a $wife(C, B)$, receberiam valor 1, resultando em um vetor $v_1 = (1, 1, 0)$. Em relação ao exemplo negativo, ele tem apenas o terceiro dos três literais de corpo encontrados em $S_{\perp}$ e por isso, o seu vetor final seria $v_2 = (0, 0, 1)$.

Ao descrever C-IL^{2}P na Seção 3.2, três fases de aprendizado foram descritas: **fase de construção**, **fase de treinamento** e **fase de extração de conhecimento**. A versão

CILP++ das duas primeiras fases serão apresentadas a seguir, deixando a análise da última (extração de conhecimento) como trabalho futuro.

Após BCP ter sido aplicado nos exemplos, o algoritmo de construção do C-IL^{2}P poderá ser iniciado para se construir uma rede inicial, tratando cada literal de cada cláusula mais específica como um atributo, como explicado acima. Como o C-IL^{2}P usa conhecimento prévio para construir a rede e os únicos dados que temos são as cláusulas mais específicas, será necessário tratar uma porção do conjunto de exemplos como cláusulas de conhecimento preliminar (este subconjunto será chamado $S_{\perp}^{BK}$). É importante notar que, mesmo tendo usado exemplos para construir um modelo inicial, eles ainda precisam ser reforçados durante o treinamento ao invés de serem tratados apenas como conhecimento preliminar e ignorados durante o treinamento: quando o treinamento por retro-propagação é iniciado, as configurações iniciais de peso que a fase de construção define tendem a influenciar cada vez menos o resultado final e assim, retirar exemplos do conjunto de treino para usar exclusivamente como cláusulas de conhecimento prévio reduziria o tamanho do conjunto de treinamento e prejudicaria a capacidade de aproximar dados do modelo neural do CILP++. O pseudo-algoritmo do CILP++ para a construção da ANN recursiva inicial segue abaixo, que difere da versão do C-IL^{2}P na forma em como lidar com os literais das cláusulas, que no caso do CILP++ são predicados.

Para cada cláusula mais específica $\perp_e$ em $S_{\perp}^{BK}$, faça

1. Adicione um neurônio h na camada escondida e rotule ele como $\perp_e$;
2. Adicione neurônios de entrada com rótulos correspondendo aos átomos de cada literal encontrado no corpo de $\perp_e$;
3. Conecte eles a h com um peso calculado W se eles são positivos; $-W$ caso contrário;
4. Adicione um neurônio de saída o e rotule ele como o literal da cabeça de $\perp_e$;
5. Conecte h com o, com o mesmo peso calculado W;
6. Se o rótulo de o é idêntico a algum rótulo na camada de entrada, adicione uma conexão recursiva entre eles, com peso fixo 1;

Alternativamente, é interessante observar o quão importante é esta inicialização feita pela etapa de construção, em relação ao C-IL^{2}P: na versão proposicional (aplicando o C-IL^{2}P diretamente), essa inicialização é importante, pois o conhecimento prévio é transformado em um modelo completo de forma correta e completa. Este não é o caso neste trabalho: o subconjunto $S_{\perp}^{BK} \subseteq \perp_e$ é apenas uma amostra do conjunto de possíveis exemplos do domínio do problema, não é especificamente um conhecimento prévio. Assim, para avaliar se no caso do CILP++ essa etapa de construção é importante, foi testada uma segunda versão do pseudo-algoritmo, onde apenas as camadas de entrada e saída são construídas e um número específico de neurônios iniciais na camada escondida é inserida, mas apenas para fins de convergência: nenhum conhecimento prévio será representado nelas.

Depois de preparar as cláusulas mais específicas e construir o modelo neural inicial, o treinamento da rede vem a seguir. Ele é feito de forma bastante semelhante ao C-IL^{2}P [16], aprendendo com retro-propagação e esquecendo as conexões recursivas durante o treinamento. Porém, diferentemente do C-IL^{2}P, CILP++ pode usar retro-propagação com *conjunto de validação* e *parada antecipada*: usamos o conjunto de validação definido para medir erro de generalização durante cada época de treinamento. Assim que ele começa a subir, o treinamento é interrompido e o melhor modelo em termos de erro de validação é retornado como o ideal. Foi aplicada uma versão mais "permissiva" de parada antecipada [47]: em vez de se terminar o treinamento imediatamente assim que a validação começa a subir, o treino é encerrado quando o critério

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

é satisfeito, onde α é o parâmetro que controla quando o treino irá parar; t é o número da época de treinamento atual em execução, $E_{va}(t)$ é o erro médio de validação na época t (erro médio quadrático) e $E_{opt}(t)$ é o erro mínimo de validação obtido até então, desde a época 1 até t. A razão pela qual a parada antecipada foi escolhida para este sistema é devido à complexidade das cláusulas mais específicas: o critério de parada prematura é muito bom para se evitar *overfitting* [5] quando a complexidade do sistema (graus de liberdade) é muito maior do que o número de exemplos. Especificamente a respeito do critério de parada apresentado acima, ele foi escolhido porque o trabalho em [47] mostrou empiricamente que esse critério obteve melhores resultados aos outros comparados, em termos de acurácia e tempo de execução.

Dado um conjunto de cláusulas mais específicas $S_{\perp}^{train}$, os seguintes passos serão executados no treinamento:

1. Para cada cláusula mais específica $\perp_e \in S_{\perp}^{train}$, $\perp_e = h$:- $l_1, l_2, ..., l_n$, faça
   (a) Adicione todos os literais $l_i, 1 \leq i \leq n$ que ainda não estão representados na rede, na camada de entrada como novos neurônios;
   (b) Se h não existe ainda na rede, crie um neurônio de saída correspondendo a ele;
2. Adicione novos neurônios ocultos, se for necessário para possibilitar um treinamento adequado;
3. Torne a rede completamente conectada, adicionando conexões com peso 0 onde necessário;
4. Aplique uma pequena perturbação em todos os pesos e viés, para prevenir o problema de simetria[1];
5. Normalize todos os pesos e viés;
6. Aplique treinamento por retro-propragação usando cada $\perp_e \in S_{\perp}^{train}$, convertida para um vetor numérico;

Para ilustrar esse processo, suponha que o conjunto $S_\perp$ de cláusulas mais específicas, obtido anteriormente nesta seção, serão usados como conjunto de treino e nenhum conhecimento prévio foi usado. Na etapa 1.a, todos os literais de corpo de ambos os exemplos [$parent(A, C)$; $wife(C, B)$ e $wife(A, C)$] causariam a geração de três novos neurônios de entrada na rede, com o rótulos idênticos a seus literais correspondentes, uma vez que a rede inicial é vazia devido ao fato de que nenhum conhecimento prévio foi utilizado. A seguir, no passo 1.b, um neurónio de saída rotulado $motherInLaw(A, B)$ seria adicionado. Para o passo 2, suponha que dois neurônios escondidos serão adicionados. Em seguida, na etapa 3, serão adicionadas conexões com peso 0, saindo de todos os três neurônios de entrada para ambos os dois novos neurônios ocultos e deles para o novo neurônio de saída. A perturbação no passo 4 precisa ser grande o suficiente para evitar o problema de simetria, mas pequena o suficiente para não influenciar uma possível modelagem inicial e para isso, foi usado um valor aleatório, diferente de zero e entre $[-0,01, 0,01]$. No passo 5, a rede é normalizada, mas de um modo que se qualquer conhecimento prévio tiver sido utilizado para construir um modelo inicial na fase de construção, a sua informação ainda será mantida. Esta é uma melhoria importante do CILP++ em relação ao C-IL^{2}P que não normalizava pesos antes do treino e, dependendo do conjunto de dados que ele estava usando para treinamento, havia a possibilidade de acabar obtendo valores zerados de atualização por causa dos valores excessivamente altos que podiam ser atribuídos a W pelo algoritmo de construção. Por último, no passo 6, aprendizado por retro-propagação é executado para ambos os exemplos, disparando os neurônios de entrada $parent(A, C)$ e $wife(C, B)$ quando o exemplo positivo é inserido na rede e disparando $wife(A, C)$ quando o negativo é utilizado.

Após o treinamento, CILP++ está pronto para inferir dados desconhecidos. A inferência é feita exatamente como em uma ANN regular, fazendo-se um passo de alimentação direta (*feed-forward*) na rede com um padrão de exemplo como entrada e calcula-se a saída.

4.2 Trabalhos Relacionados

Em primeiro lugar, sobre as abordagens que utilizam cláusulas mais específicas, elas já foram usadas em ANNs como padrões de aprendizado antes, em [9], mas para se obter um eficiente avaliador de hipótese, ao invés de classificar exemplos de primeira ordem. Nesse trabalho, um subconjunto de exemplos é escolhido, as suas cláusulas mais específicas são geradas e são utilizadas como padrões de treino em conjunto com parâmetros auxiliares, tais como, o número de literais existentes e o número de variáveis diferentes encontradas na cláusula. Além disso, o trabalho em [42] introduz um novo algoritmo de

[1] O problema de simetria [18], no contexto de redes neurais, especifica que pesos idênticos, associados aos mesmos neurônios, sempre possuirão atualizações similares.

busca de hipótese em ILP, *bottom-up*, chamada Generalização Rápida (Quick Generalization – QG), que realiza reduções aleatórias individuais na cláusula mais específica, para gerar hipóteses candidatas. Além disso, ele sugere o uso de Algoritmos Genéticos (GA) nelas, para explorar ainda mais o espaço de busca. A forma como GA converte as cláusulas mais específicas é a mesma que a usada nesse trabalho: cada ocorrência de um literal é mapeado como "1" e não-ocorrências, como "0".

Com relação a proposicionalização, um sucessor recente para RSD, chamado Relf [26] foi recentemente desenvolvido. Ele aborda o problema de uma forma mais "orientada à classificação", por considerar apenas os atributos que são interessantes para se distinguir classes. Ele também descarta atributos que θ-*subsumem* qualquer uma que tenha sido gerada anteriormente. Em contraste, LINUS [28] é o primeiro sistema publicado que lida com proposicionalização. Ele apenas pode lidar com cláusulas de Horn acíclicas e livres de função, como Progol, BCP e RSD. Além disso, ele restringe as possíveis ocorrências de variáveis em uma cláusula aceitável, permitindo-a ter apenas variáveis de cabeça em seu corpo. Seu sucessor, DINUS [22], permite um subconjunto maior de cláusulas em seu método: cláusulas contendo literais de corpo com variáveis que não aparecem em um literal da cabeça (cláusulas definidas) são aceitas, mas apenas uma instanciação possível para essas variáveis é permitida. Por último, SINUS [22] melhora DINUS ao conseguir lidar com o mesmo tipo de cláusulas que Progol, fazendo uso de viés de linguagem para restringir a construção de atributos, e adicionar em seu método de proposicionalização um teste para saber se é possível unificar literais recém-encontrados com os existentes, mantendo a coerência entre a nomenclatura pré-existente de variáveis e, assim, simplifica o conjunto final de atributos. Todos os três métodos tratam literais de corpo como atributos diretamente, de forma semelhante ao que é feito em BCP. No entanto, é importante salientar que o BCP pode lidar com o mesmo tipo de cláusulas que Progol e, portanto, não possui nenhuma das limitações que a família LINUS possui. Por último, RELAGGS [24] adota uma abordagem completamente diferente para proposicionalização, conhecida como *orientada a banco de dados*: ele usa funções padrão de agregação SQL, tais como *soma* (*sum*), *contagem* (*count*) e *média* (*avg*) como atributos proposicionais.

Em relação a outros sistemas híbridos, que tentam aprender dados relacionais com outros modelos, a Rede Lógica de Markov (MLN) [51] é um sistema de Aprendizado Relacional Estatístico (SRL) [17] que combina modelos relacionais e Campos Aleatórios de Markov [21], ao invés de ANNs, como CILP++. Integração Neuro-simbólica também foi realizada em [4] para aprender com os dados de primeira ordem, mas usando modelos ARTMAP em vez de ANNs. Esses modelos são compostos por duas redes ART, onde uma é responsável pelo agrupamento de literais de corpo e outra é responsável por aglomerar literais de cabeça, e um módulo de mapeamento, que cria relações entre as duas redes para construir uma hipótese candidata. Por fim, uma abordagem diferente para a integração de ANN e lógica de primeira ordem é através de bancos de dados relacionais:

uma teoria pode ser representada estruturalmente por uma rede em forma de grafo, onde nós são predicados e implicações são conexões entre nós. Dois modelos são comparados em [63], RelNN e GNN. Eles diferem do CILP++ porque incorporam dados relacionais diretamente em sua estrutura, enquanto que a nossa abordagem proposicionaliza dados primeiro. Aqui, a mesma troca que ocorre ao se usar sistemas ILP e abordagens de proposicionalização pode ser vista: completude de informação e melhor desempenho versus eficiência.

Capítulo 5

Resultados Experimentais

Neste capítulo, a metodologia experimental adotada neste trabalho será apresentada, juntamente com os resultados obtidos com o sistema CILP++ e com o método de proposicionalização BCP, separadamente. Três sistemas serão comparados: Aleph, RSD e C4.5. O primeiro é um sistema padrão ILP, baseado em Progol, código-aberto e bem conhecido na literatura. O segundo é um método de proposicionalização também conhecido na literatura, capaz de obter resultados comparáveis com outros sistemas ILP usando um conjunto pequeno de atributos. Por fim, o terceiro é uma alternativa para lidar com BCP, que avaliaremos empiricamente.

5.1 Metodologia

Foram escolhidos seis conjuntos de dados para realizar os experimentos: o conjunto de dados *mutagenesis* [57]; os quatro conjuntos de dados *alzheimers* [20]: *amine*, *acetyl*, *memory* e *toxic*; e a base de dados KRK [2]. Na tabela D.1, algumas estatísticas a respeito desses conjuntos de dados podem ser vistas.

CILP++ foi experimentado com várias configurações diferentes, para fazer comparações entre diferentes modelos e avaliar melhor como a nossa abordagem para integração neuro-simbólica com lógica de primeira ordem se sai. Quatro análises serão feitas: acurácia e tempo de execução em comparação com o sistema Aleph, usando *mutagenesis* e os conjuntos de dados *alzheimers*; acurácia ao se realizar seleção de atributos, restringindo o comprimento da cláusula mais específica gerada através do parâmetro de profundidade de variável e aplicando mRMR; uma comparação entre BCP e RSD, em termos de acurácia e tempos de execução, obtidos ao gerar atributos para a rede neural do CILP++ e o sistema C4.5; e análise de robustez a ruído multiplicativo (na saída), em comparação ao Aleph. Aqui, usamos *mutagenesis* e *krk* ao invés de *mutagenesis* e *alzheimers* devido a uma limitação no RSD [1].

[1]Devido ao fato de que implementação disponível da proposicionalização RSD só pode lidar com conceitor-alvo (predicados que serão aprendidos) contendo aridade (número de termos internos ao pre-

Uma observação precisa ser feita: em relação aos conjuntos de dados *alzheimers* e *mutagenesis*, um número variado de resultados de acurácia usando Aleph têm sido relatados na literatura, como [20, 27, 43]. Como resultado, decidimos executar nossas próprias comparações entre CILP++ e Aleph. Nós construímos 10 divisões (*folds*) para todos os experimentos e os dois sistemas o compartilhou. Em relação ao RSD, uma vez que foi usada a sua ferramenta, disponibilizada pelo seu autor, esse compartilhamento não foi possível. Por causa disso, RSD separou os dados de treinamento em 10 *folds*. No que diz respeito às configurações experimentais do CILP++, usaremos quatro tipos de modelos:

- st: usa retro-propagação padrão para treinar;
- es: usa parada repentina;
- $n\%bk$: sua rede é construída com um subconjunto de $n\%$ de exemplos;
- $2h$: não passa pela etapa de construção e começa com dois neurônios ocultos somente.

A configuração $2h$ é explicada com detalhes em [18]: redes neurais artificiais, com apenas dois neurônios na camada escondida, generalizam problemas binários aproximadamente tão bem quanto outros modelos mais complexos. Além disso, se uma rede neural tem também muitos atributos para avaliar (se a camada de entrada tem muitos neurônios), ele já um número elevado de *graus de liberdade* (*degrees of freedom*), fazendo com que acréscimos desnecessários de neurônios ocultos causem o aumento da probabilidade de acontecer o problema de *overfitting* [5].

Para todos os experimentos com o Aleph, as mesmas configurações que [27] serão utilizadas para os conjuntos de dados *alzheimers* (qualquer outro parâmetro que não for especificado abaixo foi utilizado com os valores padrão fornecidos pelo sistema): **profundidade de variável** = 3, **cobertura positiva mínima de uma cláusula válida** = 2, **precisão mínima de uma cláusula aceitável** = 0.7, **pontuação mínima de de uma cláusula válida** = 0.6, **número máximo de literais em uma cláusula válida** = 5 e **número máximo de exemplos negativos cobertos por uma cláusula válida (ruído)** = 300. Com relação a *mutagenesis*, com base em [43], um parâmetro foi alterado: **cobertura positiva mímina de uma cláusula válida** = 4. Para o *krk*, vamos usar a configuração fornecida pelo próprio Aleph em sua documentação.

Para o CILP++, serão usados os mesmos valores de profundidade de variável que o Aleph usou para o BCP e os seguintes parâmetros serão utilizados para o treinamento com retro-propagação nas configurações *st*: **taxa de aprendizado** = 0.1, **fator de decaimento** = 0.995 e **momento** = 0. Para todos os experimentos com configurações *es*, esses são os parâmetros utilizados: **taxa de aprendizado** = 0.05, **fator de decaimento** =

dicado) 1, não foi possível aplicá-lo nos conjuntos de dados *alzheimers* (que possuem conceitos-alvo com aridade 2) sem modificar o seu conhecimento prévio. Foi considerado que as mudanças necessárias nos conjuntos de dados originais iria comprometer as comparações de desempenho com outros trabalhos que utilizaram os mesmos conjuntos de dados.

0.999, **momento** = 0.1 e ***alpha*** (critério de parada do treinamento com parada prematura) = 0.01. Os valores usados para a taxa de aprendizado e para o fator de decaimento são recomendados em [18] para bons resultados de treinamento de um modo geral, e os valores de momento foram escolhidos através de experimentos preliminares realizados durante o desenvolvimento do CILP++.

5.2 Resultados

A primeira análise feita, cujos resultados podem ser vistos nas Tabelas D.2 e D.3, procurou mostrar as configurações *st* e *es*, em termos de acurácia e tempo de execução quando comparadas ao Aleph.

Em relação aos resultados da configuração *st*, podemos ver que Aleph e CILP++ são comparáveis em relação à configuração *st,2.5%bk*, obtendo acurácia melhor em três dos cinco conjuntos de dados, sendo mais rápido em quatro deles. Por exemplo, a configuração *st,2h* no conjunto de dados *alz-toxic* obteve acurácia melhor do que o Aleph e foi três vezes mais rápido. Isso indica que a configuração *standard* (treinando da mesma forma que o C-IL^{2}P) é adequado para aprendizado relacional, em ambas acurácia e velocidade de execução.

Nos resultados da configuração *es*, por um outro lado, o CILP++ obteve tempos de execução ainda mais expressivos, mas decaiu consideravelmente com relação à acurácia devido ao número muito menor de épocas de treinamento executadas, sendo que o Aleph conseguiu superar o CILP++ em quase todos os conjuntos de dados, exceto mutagenesis. Isso pode indicar que o foco principal do critério de parada repentina, que é a prevenção de *overfitting*, possivelmente não é útil para treinar cláusulas mais específicas: ela não leva em consideração como o erro de treinamento está decaindo e apenas olha para o erro de validação. Devido a isso, um critério de parada que sacrifica uma porção do conjunto de treino para ser usada como validação para evitar algo que possa não estar acontecendo provavelmente não é o melhor pra esse tipo de problema. Pode-se concluir com essa análise que há uma troca entre tempo de execução e acurácia quando: se é interessante para um determinado problema de classificação aprender o mais rápido possível, mesmo se não aprender bem, a configuração *es* pode ser uma opção. Mas se a situação for oposta, a configuração *st* é mais adequada.

A segunda análise envolveu o estudo da influência da seleção de atributos no desempenho do CILP++, através de variações no parâmetro de profundidade de variável do algoritmo de geração de cláusulas mais específicas (Algoritmo A.2) e da aplicação de um método estatístico de escolha de atributos chamado mRMR, cujos resultados podem ser vistos nas Figuras D.1 e D.2, respectivamente. Ambos os resultados foram obtidos com os conjuntos de dados *alz-amine* e *alz-toxic*.

Os resultados da primeira figura, com diferentes valores de profundidade de variável,

indicam que o valor padrão que escolhemos para a profundidade de variável é satisfatório: não houve melhoria nos resultados de acurácia, nem aumentando, nem diminuindo seu valor. Como dito na seção 2.1, a profundidade de variável controla o quão fundo o algoritmo de geração de cláusula mais específica irá ao gerar encadeamento de conceitos e é uma forma de controlar a quantidade de perda de informação no BCP. Intuitivamente, isso indica que maior profundidade de variável deve significar um melhor desempenho, mas junto com atributos interessantes e discriminativos do conjunto de dados, atributos redundantes também estarão dentro de representações muito extensas: nem todos os encadeamentos gerados serão úteis para descrever um exemplo, uma vez que o algoritmo de geração de cláusulas mais específicas aplica todos os possíveis literais de corpo que o viés de linguagem permite a cada ciclo pelas declarações de modo.

Com relação aos resultados da segunda figura, avaliando-se a acurácia obtida com o uso do mRMR, pode-se ver que este método, ao contrário da profundidade de variável, pode realmente ser utilizado para simplificar o conjunto de atributos: para o conjunto de dados *alz-amine*, por exemplo, uma redução de 90% do número de atributos causou uma perda inferior a 3% na acurácia com a configuração *st, 2,5%bk*. A conclusão é que a seleção de atributos pode ser útil para o BCP, mas apenas após a geração da cláusula mais específica, porque mudanças na profundidade de variável mostrou-se ineficaz em reduzir o conjunto de atributos sem causar perdas consideráveis na acurácia do CILP++. Porém, usando-se mRMR, uma redução de mais de 90% no conjunto de atributos teve como consequência uma perda de acurácia de menos de 3%. Isso pode ser útil se um aprendedor não funciona bem com muitos atributos.

Os resultados da terceira análise, referente a comparações do BCP com RSD, mostradas na Tabela D.4, mostram que o sistema CILP++, como um todo (configuração *BCP + ANN*) é melhor do que o RSD e ao mesmo tempo mantendo resultados competitivos com Aleph, mas BCP e RSD obtiveram acurácia similar quando o aprendedor de árvores de decisão C4.5 foi usado. Quanto a tempos de execução, CILP++ mostrou-se mais rápido que RSD. Os resultados também mostram que o BCP proporciona aos aprendedores proposicionais um conjunto de dados preciso, tornando-os aptos a classificar dados com acurácia competitiva com Aleph e com RSD, mas com o modelo neural sugerido (a ANN do CILP++), BCP se sobressai. Por outro lado, RSD obteve performance bastante fraca usando a ANN do CILP++. Quanto a tempo de execução, quando comparado com RSD, BCP é mais rápido. O objetivo neste cenário foi mostrar que o BCP, quando usado como um método de proposicionalização separado do CILP++, é rápido e capaz de gerar atributos significativos para a aprendizagem. Os resultados com Aleph também foram mostrados para os conjuntos de dados apresentados, para que sejam usados como base.

Por fim, a quarta e última análise envolveu analisar como o CILP++ se sai quando lida com ruído multiplicativo. Como mencionado anteriormente, ruído multiplicativo não é algo que uma rede neural artificial consiga lidar naturalmente, como acontece com ruído

aditivo: é preciso um modelo apropriado para isso [7]. Os resultados obtidos nessa análise são mostrados na Figura D.3, com os conjuntos de dados *alz-amine* e *alz-toxic*. Nela, pode-se ver que Aleph é mais robusto a ruído multiplicativo. Como ele possui um parâmetro (*noise*) que controla diretamente quão robusto ele vai ser, isso explica a diferença de robustez observada. Também deve ser levado em consideração que não é esperado que métodos heurísticos de proposicionalização tenham performance tão bem quanto sistemas ILP, pois há perda de informação nesse tipo de proposicionalização. Mesmo assim, quando o nível de ruído começou a aumentar (com 20% e 30% de nível de ruído), a inclinação da curva do modelo *st,2.5%bk* do CILP++ começou a decair mais suavemente que Aleph. Esse comportamento pode ser visto na configuração *es,2h* também, mas com essa intensidade de ruído, essa configuração não é capaz de aprender praticamente nada, em ambos os conjuntos de dados usados (a acurácia obtida é muito próxima de 50%). A conclusão desta análise é que da forma como se encontra atualmente, CILP++ não é robusto a ruído multiplicativo. Como trabalho futuro, uma análise mais profunda desse comportamento, usando conjuntos de dados contendo predicados com atributos contínuos, deve ser feita.

Capítulo 6

Conclusão

Este trabalho introduziu um algoritmo para ILP aprendizagem com ANNs, através da extensão de um sistema neuro-simbólico chamado C-IL^{2}P. As duas contribuições desta dissertação são: um novo método de proposicionalização, BCP; e CILP++, um sistema neuro-simbólico, com código-fonte aberto, capaz de lidar com dados relacionais. CILP++ conseguiu obter acurácia comparável com o Aleph nas configurações *st* e foi pior nas configurações *es*. Levando tempo de execução em consideração, CILP++ obteve desempenho superior em ambos os conjuntos de configurações, porém no conjunto *es*, ele chegou a ser mais de 10 vezes mais rápido. Pode-se observar uma troca em termos de rapidez e acurácia nessas duas configurações. Levando seleção de atributos em consideração, foi mostrado que mRMR é aplicável nos dados proposicionalizados com BCP e que pode reduzir drasticamente o número de atributos gerados, em troca de uma perda mínima de acurácia (menor do que 3%). Sobre as comparações com RSD, BCP mostrou-se superior usando ANN e manteve-se no mesmo nível usando C4.5, porém sendo mais rápido em ambos. Por último, no que diz respeito à robustez a ruído multiplicativo, Aleph demonstrou ser superior ao CILP++, mas sua acurácia começa a decair mais rapidamente com o aumento do nível de ruído. É importante ressaltar que a ANN não necessariamente é robusta a este tipo de ruído e que ajustes pré-treinamento precisam ser feitos para que a ANN adquira tais propriedades [7]. Ainda assim, existem melhorias que precisam ser pesquisadas e estudadas, que serão mostradas mais tarde neste capítulo, quando trabalhos futuros forem discutidos.

6.1 Observações Finais

Em primeiro lugar, uma vez que qualquer problema de ILP que contém um conhecimento prévio bem definido e um viés de linguagem estruturado pode gerar cláusulas mais específicas a partir de exemplos, CILP++ pode ser aplicado nos mesmos problemas que sistemas ILP podem. Isto é importante de enfatizar porque RSD, por exemplo, não pode

ser aplicado em dados com predicados alvo com aridade maior que 1.

Além disso, é importante salientar a importância de se modelar um conhecimento prévio apropriado como um modelo inicial neural. C-IL^{2}P tem capacidade de dedução e modos corretos e completos de traduzir um conhecimento prévio na forma de um programa de lógica proposicional em uma ANN recursiva e vice-versa, bem como restringe os possíveis valores iniciais de pesos e limiares, para garantir a consistência do conhecimento prévio subjacente: se um conjunto de neurônios de entrada é definido como ativado, após a estabilização da rede, todos os neurônios de saída correspondentes a literais que são verdadeiros quando os literais correspondentes de entrada também são, estarão ativados. CILP++, por outro lado, teve que usar parte dos exemplos de treinamento como conhecimento preliminar, assumindo que eles são verdadeiros (a mesma premissa que sistemas ILP definem em relação a conhecimento preliminar, em geral). Isso é um pré-requisito forte, visto que é comum que exemplos de treino possuam ruído.

6.2 Trabalhos Futuros

Como mencionado anteriormente, ainda há muito a ser feito em relação ao aprendizado relacional com o CILP++, já que ela é nova e essa foi a nossa primeira tentativa de estender C-IL^{2}P.

6.2.1 Aprimoramentos no CILP++

Em primeiro lugar, tal como sugerido na última seção, uma tradução adequada do conhecimento preliminar em um modelo neural é um dos principais trabalhos futuros desta abordagem. Ao desenvolvê-la e comprovar sua adequabilidade através de teoremas que comprovem completude e corretude do método, CILP++ será capaz de inferir dados desconhecidos a partir de uma rede integrada com conhecimento preliminar e aprender com exemplos ainda mais rápido devido a uma melhor inicialização da ANN, de forma semelhante ao C-IL^{2}P. Esse aprimoramento está sendo investigado e a primeira tentativa que será realizada usará o viés de linguagem (*declarações de modo* e *determinantes*) e cláusulas do conhecimento preliminar (excluindo os fatos) para construir este modelo inicial.

Depois, será adicionada uma etapa intermediária entre as etapas de construção e treinamento do CILP++, onde o conjunto de exemplos será verificado e os conceitos intermediários que ocorrem no conhecimento preliminar e são desconhecidos em um ou mais exemplos serão completados através da própria rede, consultando-a e inserindo as respostas nos atributos desconhecidos. Conceitos intermediários são predicados que não aparecem como cabeça de qualquer cláusula e entre os literais do corpo de alguma outra cláusula. O trabalho original do C-IL^{2}P em [16] fez isso, mas fora de sua estrutura (os

atributos dos exemplos de treino que são desconhecidos foram completados através da consulta do conhecimento preliminar usando Prolog). O objetivo disso é analisar se pode haver algum benefício em se fazer isso numericamente, usando a ANN inicial que foi modelada na etapa de construção, em vez de uma consulta Prolog, tendo-se um exemplo que tem atributos desconhecidos como entrada. Qualquer resposta diferente de zero, para qualquer conceito intermediário, será usado no lugar de "desconhecido" (zero) naquele exemplo e após isso, ele estará apto a ser usado como padrão de treinamento.

6.2.2 Extração de Conhecimento

O estudo de como a última etapa do ciclo de aprendizado do C-IL^{2}P, extração de conhecimento, pode ser feito no CILP++ é um importante trabalho futuro. Uma primeira tentativa, inspirada em [13], é associar uma ordem para literais da cláusula mais específica, de acordo com a ordem (posição) de sua respectiva *declaração de modo* e extrair cláusulas usando o encadeamento de variáveis de literais fortemente conectados (com altos valores nos pesos após o treinamento) a um neurônio oculto em comum. Isto pode gerar cláusulas interessantes, mas um estudo aprofundado é necessário.

Uma forma imediata de se obter uma lista proposicional de regras com o CILP++ em seu estado atual é usando uma aprendedor de regras ou de árvores de decisão com o BCP. A árvore de decisão gerada a partir do sistema C4.5, por exemplo, pode ser facilmente convertida para regras do tipo se-então.

Alternativamente, o próprio procedimento de extração de conhecimento que o C-IL^{2}P usou pode ser aplicado no CILP++ [13], embora seja consideravelmente dispendioso e possa gerar cláusulas sem sentido, com muitas variáveis livres (por exemplo, se ele extrair os rótulos de neurônios que mapeiam literais do início de uma cláusula mais específica, juntamente com neurônios que mapeiam literais presentes no final dela).

6.2.3 Outras Análises

Em primeiro lugar, uma comparação mais ampla dos resultados obtidos com os trabalhos relacionados poderia ser interessante, para verificar o quão eficiente CILP++ é em relação a eles, especialmente contra diferentes métodos de proposicionalização, para enriquecer os resultados obtidos. Mais especificamente a respeito do trabalho em [9], que utiliza as cláusulas mais específicas como padrões de treino para construir um avaliador de hipóteses, foram adicionados vários meta-parâmetros, tais como o tamanho da cláusula mais específica e o número de predicados distintos, entre outros, como atributos adicionais. Esses atributos extras podem contribuir para uma melhor aprendizagem e inferência com o CILP++.

Em segundo lugar, experimentos em conjuntos de dados contínuos são necessários. Será interessante ver como o CILP++ se comporta quando usado nesse tipo de dados e

analisar se esta abordagem herda a robustez a ruído aditivo do algoritmo tradicional de retro-propagação da ANN. O conjunto de dados *carcinogenesis* [56], por exemplo, contém vários predicados contendo atributos contínuos, que por sua vez podem receber ruído aditivo. Um problema imediato com o CILP++ quando lidar com dados contínuos é que os atributos de predicados contendo valores reais não podem ser diretamente considerados como atributos proposicionais. Por exemplo, um atributo do tipo "P(0.454)" seria muito pouco provável de aparecer novamente em outro exemplo e portanto não teria utilidade no aprendizado neural que busca aprender as características que conceitos que possuem a mesma classe compartilham. Este problema poderia ser remediado, por exemplo, se um pré-processamento adicional nos dados pós-BCP fosse feito para converter tais valores em intervalos numéricos, de um modo semelhante como o sistema FOIL [53] faz.

Em terceiro lugar, existem algumas áreas recentes que precisam ser estudadas em relação a como elas podem se relacionar ou como podem ser aplicadas a este trabalho, como Transferência de Aprendizagem [44], que se concentra em armazenar conhecimento obtido no aprendizado de um problema e usá-lo em outro diferente, mas relacionado. Isto pode ser útil se ele permitir que o CILP++ reutilize modelos previamente construidos. Problemas que compartilham o mesmo domínio, como os conjuntos de dados *alzheimer*, podem vir a se beneficiar desse compartilhamento de modelos.

6.2.4 Outras Aplicações

CILP++ abre caminho a aplicações que poderiam se beneficiar com aprendizado relacional mas requerem eficiência, tais como web-semântica e agentes inteligentes. Até o presente momento, não há nenhuma aplicação ILP existente em nenhuma dessas áreas e talvez uma abordagem neuro-simbólica possa ser um caminho a seguir.

Por último, devido aos resultados mostrados para seleção de atributos com mRMR, vale a pena avaliar como a abordagem desse trabalho irá lidar com conjuntos de dados relacionais muito grandes, como por exemplo, *CORA* ou *proteins*, que são considerados desafiadores para indutores ILP [45]. Os resultados mostram que, mesmo com uma redução de mais de 90% no número de atributos, não há perda considerável de acurácia.

Referências Bibliográficas

[1] Anderson, M. L. and Perlis, D. R. (2005). Logic, self-awareness and self-improvement: The metacognitive loop and the problem of brittleness. *Journal of Logic and Computation*, 15(1):21–40.

[2] Bain, M. and Muggleton, S. (1994). Learning optimal chess strategies. In *Machine Intelligence 13*, page 291.

[3] Barros, R., Cerri, R., Jaskowiak, P., and de Carvalho, A. (2011). A bottom-up oblique decision tree induction algorithm. In *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*, pages 450–456.

[4] Basilio, R., Zaverucha, G., and Barbosa, V. (2001). Learning Logic Programs with Neural Networks. In *Inductive Logic Programming*, Lecture Notes in Computer Science, pages 15–26. Springer Berlin / Heidelberg.

[5] Caruana, R., Lawrence, S., and Giles, C. L. (2000). Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *in Proc. Neural Information Processing Systems Conference*, pages 402–408.

[6] Clark, P. and Niblett, T. (1989). The CN2 Induction Algorithm. *Machine Learning*, 3:261–283.

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

[8] Corapi, D. (2011). *Nonmonotonic Inductive Logic Programming as Abductive Search*. Ph.D. thesis, Imperial College, London, United Kingdom.

[9] DiMaio, F. and Shavlik, J. W. (2004). Learning an Approximation to Inductive Logic Programming Clause Evaluation. In *ILP*, pages 80–97.

[10] Ding, C. and Peng, H. (2005). Minimum redundancy feature selection from microarray gene expression data. *Journal of bioinformatics and computational biology*, 3(2):185–205.

[11] Duboc, A. L., Paes, A., and Zaverucha, G. (2009). Using the bottom clause and mode declarations in FOL theory revision from examples. *Mach. Learn.*, 76(1):73–107.

[12] Elman, J. L. (1990). Finding Structure in Time. *Cognitive Science*, 14(2):179–211.

[13] Garcez, A. S. D., Broda, K., and Gabbay, D. M. (2001). Symbolic Knowledge Extraction From Trained Neural Networks: A Sound Approach. *Artifcial Intelligence*, 125(1-2):155–207.

[14] Garcez, A. S. D., Broda, K. B., and Gabbay, D. M. (2002). *Neural-Symbolic Learning System: Foundations and Applications*. Perspectives in Neural Computing Series. Springer Verlag.

[15] Garcez, A. S. D., Lamb, L. C., and Gabbay, D. M. (2008). *Neural-Symbolic Cognitive Reasoning*. Springer Publishing Company, Incorporated, 1 edition.

[16] Garcez, A. S. D. and Zaverucha, G. (1999). The Connectionist Inductive Learning and Logic Programming System. *Applied Intelligence*, 11:59–77.

[17] Getoor, L. and Taskar, B. (2007). *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

[18] Haykin, S. (2009). *Neural Networks and Learning Machines*. Number v. 10 in Neural networks and learning machines. Prentice Hall.

[19] Kijsirikul, B. and Lerdlamnaochai, B. K. (2005). First-Order Logical Neural Networks. *Int. J. Hybrid Intell. Syst.*, 2(4):253–267.

[20] King, R. and Srinivasan, A. (1995). Relating chemical activity to structure: An examination of ILP successes. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):411–434.

[21] Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning Series. Mit Press.

[22] Kramer, S., Lavrač, N., and Flach, P. (2000). Relational Data Mining. chapter Propositionalization approaches to relational data mining, pages 262–286. Springer-Verlag New York, Inc., New York, NY, USA.

[23] Krogel, M.-A., Rawles, S., Železný, F., Flach, P., Lavrač, N., and Wrobel, S. (2003). Comparative Evaluation Of Approaches To Propositionalization. In *ILP'2003*, pages 194–217. Springer-Verlag.

[24] Krogel, M. A. and Wrobel, S. (2003). Facets of Aggregation Approaches to Propositionalization. pages 30–39. Department of Informatics, University of Szeged.

[25] Kuželka, O. and Železný, F. (2008). HiFi: Tractable Propositionalization through Hierarchical Feature Construction. In Železný, F. and Lavrač, N., editors, *Late Breaking Papers, the 18th International Conference on Inductive Logic Programming*.

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

[27] Landwehr, N., Kersting, K., and Raedt, L. D. (2007). Integrating Naive Bayes and FOIL. 8:481–507.

[28] Lavrač, N. and Džeroski, S. (1994). *Inductive logic programming: techniques and applications*. Ellis Horwood series in artificial intelligence. E. Horwood.

[29] Lavrač, N. and Flach, P. A. (2001). An extended transformation approach to inductive logic programming. *ACM Trans. Comput. Logic*, 2(4):458–494.

[30] May, R., Dandy, G., and Maier, H. (2011). *Review of Input Variable Selection Methods for Artificial Neural Networks*, pages 19–44. InTech.

[31] Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D. L., and Kolobov, A. (2005). BLOG: probabilistic models with unknown objects. In *Proceedings of the 19th international joint conference on Artificial intelligence*, IJCAI'05, pages 1352–1359, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[32] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York.

[33] Mooney, R. J., Shavlik, J., Towell, G., and Gove, A. (1989). An Experimental Comparison of Symbolic and Connectionist Learning Algorithms. In *IJCAI'89*, pages 775–780, Detroit, MI.

[34] Muggleton, S. (1992). Inductive Logic Programming. In *New Generation Computing*. Academic Press.

[35] Muggleton, S. (1994). Bayesian inductive logic programming. In *Proceedings of the seventh annual conference on Computational learning theory*, pages 3–11, New York, USA. ACM.

[36] Muggleton, S. (1995a). Inductive Logic Programming: Inverse Resolution and Beyond. In *IJCAI*, page 997.

[37] Muggleton, S. (1995b). Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286.

[38] Muggleton, S. and Feng, C. (1990). Efficient Induction Of Logic Programs. In *New Generation Computing*. Academic Press.

[39] Muggleton, S. and Feng, C. (1992). Efficient induction of logic programs. *Inductive logic programming*, pages 1–14.

[40] Muggleton, S., Lodhi, H., Amini, A., and Sternberg, M. J. E. (2005). Support vector inductive logic programming. In *In Proceedings of the Eighth International Conference on Discovery Science, volume 3735 of LNAI*, pages 163–175. Springer-Verlag.

[41] Muggleton, S. and Raedt, L. D. (1994). Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19(20):629–679.

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

[43] Paes, A., Železný, F., Zaverucha, G., Page, D., and Srinivasan, A. (2007). Inductive logic programming. chapter ILP Through Propositionalization and Stochastic k-Term DNF Learning, pages 379–393. Springer-Verlag, Berlin, Heidelberg.

[44] Pan, S. J. and Yang, Q. (2010). A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359.

[45] Perlich, C. and Merugu, S. (2005). Gene Classification: Issues And Challenges For Relational Learning. In *Proceedings of the 4th international workshop on Multi-relational mining*, pages 61–67, New York, NY, USA. ACM.

[46] Pollack, J. B. (1990). Recursive distributed representations. *Artif. Intell.*, 46(1-2):77–105.

[47] Prechelt, L. (1997). Early stopping - but when? In *Neural Networks: Tricks of the Trade, volume 1524 of LNCS, chapter 2*, pages 55–69. Springer-Verlag.

[48] Quinlan, J. R. (1986). Induction of Decision Trees. *Mach. Learn.*, 1(1):81–106.

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

[50] Richards, B. L. and Mooney, R. J. (1991). First-Order Theory Revision. In *Proceedings of the Eighth International Machine Learning Workshop*, pages pp. 447–451, Evanston, IL.

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

[52] Rumelhart, D. E., Widrow, B., and Lehr, M. A. (1994). The Basic Ideas In Neural Networks. *Commun. ACM*, 37(3):87–92.

[53] Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

[54] Sørmo, F., Cassens, J., and Aamodt, A. (2005). Explanation in Case-Based Reasoning-Perspectives and Goals. *Artif. Intell. Rev.*, 24(2):109–143.

[55] Srinivasan, A. (2007). The Aleph System, version 5. http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html. Last accessed on nov/2012.

[56] Srinivasan, A., King, R. D., Muggleton, S. H., and Sternberg, M. J. E. (1997). Carcinogenesis Predictions using ILP. *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, 1297(1297):273–287.

[57] Srinivasan, A. and Muggleton, S. H. (1994). Mutagenesis: ILP experiments in a non-determinate biological domain. In *Proceedings of the 4th International Workshop on Inductive Logic Programming, volume 237 of GMD-Studien*, pages 217–232.

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

[59] Tarski, A. and Helmer-Hirschberg, O. (1946). *Introduction to logic and to the methodology of deductive sciences*. Oxford University Press.

[60] Torre, F. and Rouveirol, C. (1997). *Private Properties and Natural Relations in Inductive Logic Programming*. Rapports de recherche. Université Paris-Sud, Centre d'Orsay, Laboratoire de recherche en Informatique.

[61] Towell, G. G. and Shavlik, J. W. (1994). Knowledge-Based Artificial Neural Networks. *Artif. Intell.*, 70(1-2):119–165.

[62] Towell, G. G., Shavlik, J. W., and Noordewier, M. O. (1990). Refinement of Approximate Domain Theories by Knowledge-Based Neural Networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 861–866.

[63] Uwents, W., Monfardini, G., Blockeel, H., Gori, M., and Scarselli, F. (2011). Neural networks for relational learning: an experimental comparison. *Mach. Learn.*, 82(3):315–349.

[64] Železný, F. (2004). Efficiency-conscious Propositionalization for Relational Learning. *Kybernetika*, 4(3):275–292.

[65] Železný, F. and Lavrač, N. (2006). Propositionalization-based Relational Subgroup Discovery With RSD. *Mach. Learn.*, 62:33–63.

[66] Zhang, Y., Ding, C., and Li, T. (2008). Gene selection algorithm by combining reliefF and mRMR. *BMC genomics*, 9 Suppl 2:S27.

[67] Zhang, Z., Kwoh, C. K., Liu, J., Yin, F., Wirawan, A., Cheung, C., Baskarang, M., Aung, T., and Wong, T. Y. (2011). MRMR Optimized Classification for Automatic Glaucoma Diagnosis. In *IEEE Engineering in Medicine and Biology Society*, pages 6228 – 6231.

[68] Zur, R. M., Jiang, Y., Pesce, L. L., and Drukker, K. (2009). Noise Injection For Training Artificial Neural Networks: A Comparison With Weight Decay And Early Stopping. *Medical Physics*, 36(10):4810–4818.

Apêndice A

Background

In this chapter, both Machine Learning subfields that are used on this work (Inductive Logic Programming and Artificial Neural Networks) will be reviewed, focusing on the topics that relates the most with this work. Afterwards, feature selection and propositionalization techniques will be briefly explained, which are commonly used when dealing with Artificial Neural Networks and Inductive Logic Programming, respectively.

A.1 Inductive Logic Programming

Inductive Logic Programming (ILP) [34] is a recent Machine Learning [32] subfield that makes use of logical languages to induce theory-structured hypothesis in order to achieve two goals [53]:

- Allow inference over unknown data by evaluating it with a generated theory;
- Produce a human-readable logical knowledge that explains the learned concepts and can be easily interpreted for other similar tasks.

One of the most well-known ILP systems, *Progol* [37], performs a sequential-covering algorithm [53] to build candidate hypothesis and uses a score function to decide which one(s) will be part of the resultant theory. To do so in a feasible way, it bounds hypothesis space by using a most saturated clause, called *bottom clause*. In order to properly introduce it, some basic concepts needs to be presented. Assume that in the following, all clauses are in their disjoint normal form. For more detailed definitions, please refer to [32, 37, 53].

- Substitution: a substitution θ is a set of the form $\{X_1/t_1, \ldots, X_n/t_n\}$, where each X_i is a different variable and each t_i is a term distinct from X_i. Additionally, $C\theta$ represents a clause C after substitution θ had been applied on it.
- Unification: an unifier θ is a substitution that, when applied on a pair of clauses C_i and C_j, it makes every literal l_i in C_i identical to a literal l_j in C_j and vice-versa.

- Subset ($\subseteq_c$): consider two clauses $C_i = l_1^i \vee l_2^i \vee \ldots \vee l_m^i$ and $C_j = l_1^j \vee l_2^j \vee \ldots \vee l_n^j$, where $l_1^i, l_2^i, \ldots l_m^i$ and $l_1^j, l_2^j, \ldots l_n^j$ are literals. We say that $C_i \subseteq C_j$ if and only if $\forall k, 1 \leq k \leq m, l_k^i \in \{l_o^j, \forall 1 \leq o \leq n\}$.
- θ-subsumption: consider two clauses C_i and C_j, both of the form $h \vee l_1 \vee l_2 \vee \ldots \vee l_n$, where h is a positive literal and $l_1, l_2, \ldots l_n$ are literals. We say that C_i θ-subsumes C_j if and only if there exists a substitution θ such that $C_i\theta \subseteq C_j$.
- (Literal) Interpretation ($I(l)$): given a literal l, a interpretation for l is an assignment of a truth-value for it, $I(l) \in \{true, false\}$.
- (Clause) Interpretation ($I(C)$): given a clause C, a interpretation for C is a set of interpretations for each of its atoms:

$$I(C) : \{l_1^C, \ldots, l_n^C\} \mapsto \{true, false\}^{|n|}$$

- Models: interpretations that assigns truth values to a given clause or set of clauses (theory):

$$M(C) = \{I \in I(C) | I(C) \mapsto true\}.$$

- Entailment (Logical Consequence) ($\models$): consider two clauses C_i and C_j. C_i entails C_j ($C_i \models C_j$) if and only if $M(C_i) \subseteq M(C_j)$.

A.1.1 ILP Definition

ILP can be defined as a set of techniques that conducts *supervised inductive concept learning*: given a set of labeled examples E and a background knowledge B, an ILP system will try to find a hypothesis H that minimizes a specified loss function $loss(E, B, H)$. For this definition, each used element is:

- B: a set of clauses, which can be *facts* (grounded single-literal clauses that defines what is known about a given task) or *rule clauses*, which defines relations between facts;
- E: a set of ground atoms of target concept(s), in which labels are truth–values[1]. Moreover, E = $E^+ \cup E^-$, where E^+ is the set of positive examples and E^- is the set of negative examples;
- H: a target definite program that entails all positive examples of E and possibly some of negative ones as well, in order to avoid overfitting and allow proper generalization.
- $loss(E, B, H)$: based on the number of positive examples (elements of E^+) entailed by $B \cup H$ and the number of negative examples (elements of E^-) not entailed by $B \cup H$;

[1]This definition is compatible with *learning from entailment* setting, which will be the focus of this work.

An ILP task is a tuple $< E, B, L >$, where E and B are as defined above and L is a set of logic theories that restricts the best hypothesis search and is known as **language bias**. L is different for each problem and what kind of specifications and restrictions it sets upon the search space depends on the ILP system used and in order to explain how it works, one particular ILP system will be chosen: Progol.

As of most of ILP systems, Progol executes a standard sequential-covering algorithm to induce theories, which can be seen in Algorithm A.1. It has the tuple $< E, B, L >$ as input and outputs a theory set H (candidate hypothesis), and uses two auxiliary sets: E_{cov} to store the examples that are already being covered by H and E_{cur}, to keep the remaining ones.

Algorithm A.1 Sequential-covering algorithm

Inputs: E, B, L
Outputs: H

1: $E_{cur} = E$
2: $H = \emptyset$
3: **while** generalization stopping criteria is satisfied **do**
4: $\quad c = h \leftarrow$
5: $\quad$ **while** specialization stopping criteria is satisfied **do**
6: $\quad\quad c = REFINE(c, L)$
7: $\quad$ **end while**
8: $\quad H = H \cup c$
9: $\quad E_{cov} = \{e \in E_{cur} : B \cup H \models e\}$
10: $\quad E_{cur} = E_{cur} - E_{cov}$
11: **end while**
12: **return** H;

Each ILP system has its own way of specifying *REFINE(c, L)*, *specialization stopping criteria* and *generalization stopping criteria* in Algorithm A.1: how to *refine* a candidate hypothesis *c* given the language bias *L*, what exactly is considered as "specific enough" candidate clause (the *specialization stopping criteria*) and what exactly is considered as "general enough" candidate hypothesis (the *generalization stopping criteria*). In Progol, each one of those three concepts are defined as follows:

- $REFINE(c, L)$: iterative literal addition to current rule c, in order to minimize a loss function w.r.t. the examples set *E*, the hypothesis *H* and the background knowedge *B*:

$$loss_{Progol}(E, B, H) = \\ (\sum_{r \in H} |r|) - |\{e \in E^+ : B \cup H \models e\}| + |\{e' \in E^- : B \cup H \models e'\}|,$$

  where E^+ and E^- are subsets of E containing only its positive and negative examples, respectively.

- *specialization stopping criteria*: a given number of negative examples are being covered, at most, by the analyzed rule;
- *generalization stopping criteria*: all positive examples are being covered by the current candidate hypothesis.

More details about language bias and each one of those concepts will be given, respectively, in the next two subsections.

A.1.2 Language Bias

The set of all candidate hypothesis for a given task, which will be called S_H in this work, can be infinite [60]. To ameliorate that, one of the features that constrains S_H in ILP is the *language bias*, L. It is usually composed of specification predicates, which will help define how the search will be done and how far it can go. The most common specification language (used by Progol, for instance) is a set of clauses called *mode declarations* [53], which contains:

- *modeh* predicates, which define what can appear as a head of a clause;
- *modeb* predicates, which define what can appear in the body of a clause;
- *determination* predicates, which relates body and head literals.

Additionally, the *modeb* and *modeh* declarations also specifies what is considered to be an *input variable*, an *output variable* and a *constant* through a symbol "+", "-" and "#" at the front of the each one of its parameters, respectively. An illustration of the modes structure is shown in Figure (A.1).

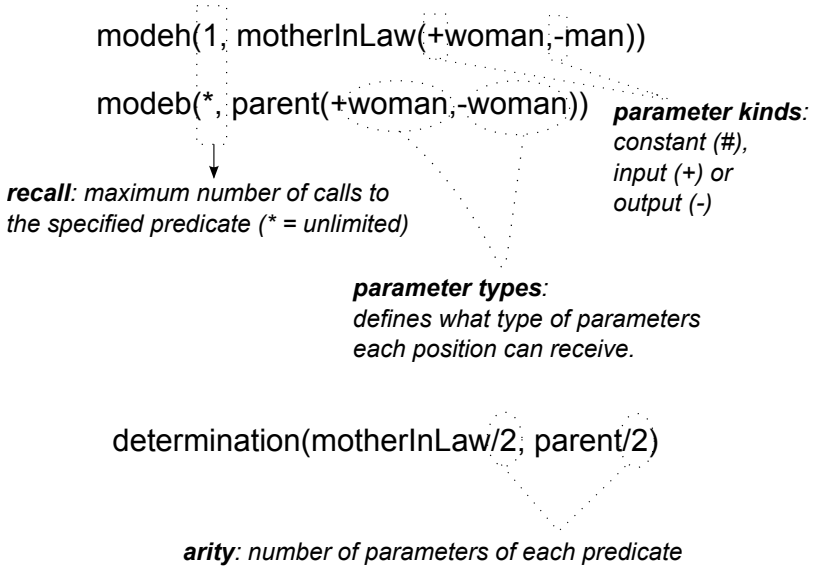

Figure A.1: Mode declarations illustration

The language bias L, through *modeb*, *modeh* and *determination* predicates, can restrict S_H during hypothesis search to only allow a smaller set of candidate hypothesis S_H^L to be analyzed. To illustrate this, let us continue the family environment shown in Section 1.1

Background Knowledge:
parent(mom1, daughter11)
wife(daughter11, husband1)
wife(daughter12, husband2)

Positive Examples:
motherInLaw(mom1, husband1)

Negative Examples:
motherInLaw(daughter11, husband2)

Language Bias:
modeh(1, motherInLaw(+woman,-man))
modeb(, parent(+woman,-woman))*
modeb(1, wife(+woman,-man))
determination(motherInLaw/2, parent/2)
determination(motherInLaw/2, wife/2)

Types:
woman(mother1)
woman(daughter11)
woman(daughter12)
man(husband1)
man(husband2)

The single *modeh* declaration above defines that only the *motherInLaw* relation can appear as a head literal on all elements of S_H and additionally, it receives an input parameter of the type *woman* and outputs an term of type *man*. On the other hand, the *modeb* declarations specifies whether a literal can or cannot be used as a body for elements in S_H and in this example, there are two viable body literals: *parent* and *wife*. Again, input/output types are defined and in this case, both *modeb* declarations have the same typings as *motherInLaw*. Lastly, the *determination* relation is used to relate literals that appears on *modeh* with literals from *modeb* declarations. In the example, *motherInLaw* is set up to use only two literals as bodies: *parent* and *wife*.

Focusing on the parameter types specified by *mode* declarations, during hypothesis construction, a *variable chaining* must be maintained: each input term in a body literal must be an output term in a previous body literal or an input term in the head. For instance, if we take into consideration an hypothesis such as

motherInLaw(A, B) :- *parent(A, C), wife(C, B)*

it can be seen that it satisfies a *variable chaining*: as defined by the *modeh* declaration, since the position of A is of an input variable and it is the only input possible for *motherInLaw*, the only option for the input position of the first body literal is A. As can be seen, the first one is *parent(A, C)*, so the chaining holds. The next one, thus, can have as input A or C. Since it has C [*wife(C, B)*], it still holds.

A.1.3 Bottom Clause

Another feature that is used to restrict hypothesis search space in algorithms based on *inverse entailment* [37], such as Progol, is the *most specific (saturated) clause*, $\perp_e$, known as *bottom clause*. Given an example e, Progol will firstly generate 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 (let the found *modeh* define a predicate h), an initial $\perp_e$ is created:

$$\perp_e : h \leftarrow .$$

Then, passing through the *determination* predicates to verify which of the bodies specified in *modeb* can be added to $\perp_e$, more terms will be added to it (let them be named $b_1, \ldots, b_n$), resulting in a most saturated clause of the form

$$\perp_e : h \leftarrow b_1, b_2, \ldots, b_n.$$

This procedure continues until a number of cycles through the *modeb* declarations have been reached. This number is a parameter of the algorithm, called *depth*. The complete bottom clause generation algorithm, used in Progol, is presented on Algorithm A.2. It receives a positive example e and outputs $\perp_e$, its correspondent bottom clause. It uses a *hash* function to map variables to constants, so each different constant found in the example and in the background knowledge will be assigned to a variable, to generate the most specific clause and an *inTerms* set, to keep all newly and usable variables. Those two elements are the ones that maintain the bottom clause variable chaining property, by creating unique labelled variables and ensuring that body input variables have been already introduced as previous outputs or one of head inputs, respectively.

To illustrate Algorithm A.2, the previously introduced family environment will be used to generate a bottom clause from its only positive example, *motherInLaw(mom1, husband1)*. For this example, assume *depth* $= 1$. Initially, the algorithm variables are initialized as follows:

- $e =$ *motherInLaw(mom1, husband1)*
- $currentDepth = 0$;
- $inTerms = \emptyset$;
- $\perp_e = \emptyset$.

Firstly, all *modeh* declarations are searched to find one is type-compatible with e, in a top-down order, and after finding it, an initial substitution set θ will be generated – this corresponds to lines 1-9 in Algorithm A.2. In our example, there is only one *modeh* candidate: *modeh(1, motherInLaw(+woman, -man))*, which is compatible with e because *mom1* is defined as a woman and *husband1* is defined as a man in the types listing.

Algorithm A.2 Bottom clause generation

1: $currentDepth = 0$, $inTerms = \emptyset$, $\perp_e = \emptyset$
2: Find a head mode declaration h that defines the same predicate as e
3: **if** all terms inside the target predicate of e have compatible types with h **then**
4: Unify the predicate definition of h and e with a substitution θ
5: **else**
6: Find another compatible head mode declaration h
7: It there is not, **return**; otherwise, go back to line 3
8: **end if**
9: Let $head$ be a copy of the predicate defined by h
10: **for all** $v/t \in \theta$ **do**
11: If v is of type $\#$, replace v in $head$ to t
12: Else, replace v in $head$ to v_k, where $k = hash(t)$
13: If v is of type $+$, add t to $inTerms$
14: **end for**
15: Add $head$ to $\perp_e$
16: **for** each body mode declaration b **do**
17: Let $body$ be a copy of the predicate defined by b
18: **for all** substitutions θ of arguments $+$ of $body$ to elements of $inTerms$ **do**
19: **repeat**
20: **if** querying $body\theta$ with substitution θ' succeeds **then**
21: **for** each v/t in θ and θ' **do**
22: If v is of type $\#$, replace v in $body$ to t
23: Else, substitute v in $body$ to v_k, where $k = hash(t)$
24: If v is of type $-$, add t to $inTerms$
25: **end for**
26: Add $body$ to $\perp_e$
27: **end if**
28: **until** $recall$ number of iterations has been done
29: **end for**
30: **end for**
31: Increment $currentDepth$; If it is less than $depth$, go back to line 16
32: **return** $\perp_e$

Thus, a substitution θ = *{+woman/mom1, -man/husband1}* can be used to unify *head* = *motherInLaw(+woman,-man)* with e.

Afterwards, applying lines 10-15, θ = *{+woman/mom1, -man/husband1}* will be variablized, in order to be used to add literals to $\perp_e$ and the variables list *inTerms* will be updated with them. Starting with the first element of θ (*+woman/mom1*), since the right-side is not a constant parameter, it falls on the case of line 12 and thus, a variable will be created in the *hash* to replace *+woman* in *head*. Since it assigns variables in alphabetical order to terms and this is the first one being created, it will be named *A*. Additionally, since it is an input parameter, the newly created variable will be added to *inTerms*. Continuing, the second and last element of θ (*-man/husband1*) will pass through same processing: since the right-side is not a constant, it will be replaced by a newly created variable gene-

rated by the *hash* function, *B* and by not being an input parameter, no changes to *inTerms* will be done because of it. Lastly, the processed $head$ predicate will be added to $\perp_e$, followed by Progol consequence operator ":-". At the final of this processing, the algorithm internal status is as shown below.

- $inTerms = \{mom1\}$
- $\perp_e$ = *motherInLaw(A, B) :-*

Next, from lines 16-30, literals will be added to the body of $\perp_e$ in a similar way the *modeh* had been used to generate the head literal, until $depth$ number of passes through all *modeb* declarations has been done. In our example, there are two *modeb* declarations: *modeb(*, parent(+woman,-woman))* and *modeb(1, wife(+woman,-man))*. For the first one, b = *modeb(*, parent(+woman,-woman))* and those are the actions taken:

1. $body$ receives the predicate defined by b, *parent(+woman,-woman)*;
2. All terms of $inTerms$ will be tested as substitutions for each input parameter of $body$. Since there is only one such input, *+woman*, θ = *+woman/mom1*;
3. Since the recall of b is *, all possible substitutions θ' in the background knowledge that makes $body\theta$ = *parent(mom1, -woman)* true will be tested:
   (a) Unify with *parent(mom1, daughter11)* through substitution θ' = *{-woman/daughter11}*;
   (b) For each element of $\theta \cup \theta$' = *{+woman/mom1, -woman/daughter11}*:
      i. v/t = *+woman/mom1* $\Rightarrow$ v is an input parameter $\Rightarrow$ $hash(mom1) = A \Rightarrow$ *body = parent(A, -woman)*;
      ii. v/t = *-woman/daughter11* $\Rightarrow$ v is an output parameter $\Rightarrow$ $hash(daughter11) = C$ (new element) $\Rightarrow$ *body = parent(A, C)*; *inTerms = {mom1, daughter11}*;
   (c) $\perp_e$ = *motherInLaw(A, B) :- parent(A, C)*;
   (d) No other unifications are possible: stop iteration;

After processing the first *modeb* declaration, those are the current values for $inTerms$ and $\perp_e$:

- $inTerms$ = *{mom1, daughter11}*
- $\perp_e$ = *motherInLaw(A, B) :- parent(A, C)*

For the second *modeb*, b = *modeb(1, wife(+woman,-man))*, this is the algorithm flow:

1. $body$ receives the predicate defined by b, *wife(+woman,-man)*;
2. All terms of $inTerms$ will be tested as substitutions for each input parameter of $body$;

3. First element of $inTerms$: *mom1*;
   (a) θ = +*woman/mom1*;
   (b) Since the recall of b is 1, the first possible substitution θ' in the background knowledge that makes $body\theta$ = *wife(mom1, -man)* true will be tested;
   (c) No possible unifications: end processing of *mom1*;
4. Second element of $inTerms$: *daughter11*;
   (a) θ = +*woman/daughter11*;
   (b) Since the recall of b is 1, the first possible substitution θ' in the background knowledge that makes $body\theta$ = *wife(daughter11, -man)* true will be tested;
   (c) $body\theta$ unifies with *wife(daughter11, husband1)* through substitution θ' = *{-man/husband1}*;
   (d) For each element of $\theta \cup \theta$' = *{+woman/daughter11, -woman/husband1}*:
      i. v/t = +*woman/daughter11* $\Rightarrow$ v is an input parameter $\Rightarrow$ $hash(daughter1) = C \Rightarrow body$ = *wife(C, -man)*;
      ii. v/t = *-man/husband1* $\Rightarrow$ v is an output parameter $\Rightarrow$ $hash(husband1) = B \Rightarrow body$ = *wife(C, B)*; $inTerms$ = *{mom1, daughter11, husband1}*;
   (e) $\perp_e$ = *motherInLaw(A, B) :- parent(A, C), wife(C, B)*;
   (f) Recall = 1: stop iteration;

After processing both *modeb* declarations, $depth$ is verified: if the number of passes through the *modeb* list is equal or higher than it, the algorithm stops. Since for this example $depth$ = 1, it is the case and the generated bottom clause is

$$\perp_e = \text{ motherInLaw(A, B) :- parent(A, C), wife(C, B).}$$

As commented on Chapter 1.1, this bottom clause has an intuitive meaning: a mother-in-law of a man is a parent of his wife. This work intends to explore the bottom clauses capabilities of representing an example, to be used as a propositionalization of a first-order example and as a learning pattern.

To continue this background review, artificial neural networks will be introduced on the next section and at the last section some relevant feature selection techniques will be presented and analyzed, in order to reduce the complexity of generated bottom clauses, and a method for transforming first-order clauses into propositional data called propositionalization will be shown, which aims to allow propositional learners such as neural networks and binary trees to deal with first-order data, usually through a trade-off between efficiency and information loss.

A.2 Artificial Neural Networks

Artificial neural networks are one of the most used machine learning techniques in the literature in a vast range of applications [16, 18, 19, 32, 46, 52]. They have been created as a model that mimics neuronal activity: after receiving a set of input signals from other neurons, it sums it up, apply its internal activation function to this total and check whether the result is enough to trigger its excitatory state, by using a threshold value. If it is, it emits an output that depends on the kind of activation function it has. On the other hand, if it is not, it maintains an inhibitory state and nothing passes through. All relevant topics about it, regarding this work, will be presented in the following subsections.

A.2.1 ANN Definition

A neural network can be defined as a directed graph with the following structure: a unit (or neuron) i 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 $A_i(t)$.

The units of a neural network can be organized in layers. We can define a *n-layer feed-forward network* as an acyclic graph consisted of a sequence of layers and connections between successive layers, containing one input layer, $n-2$ hidden layers, and one output layer, where $n \geq 2$. When $n = 3$, we say that the network is a *single hidden layer network*. When each unit occurring in the i-th layer is connected to each unit occurring in the $i+1$-st layer, we say that the network is *fully-connected*.

A *multilayer feed-forward network* computes a function $\varphi : \mathbb{R}^r \to \mathbb{R}^s$, where r and s are the number of units occurring, respectively, in the input and output layers of the network. In the case of single hidden layer networks, the computation of φ occurs as follows:

1. At time t_1, the input vector is presented to the input layer;
2. At time t_2, the input vector is propagated through to the hidden layer, and the units in the hidden layer update their input potential and activation state;
3. At time t_3, the hidden layer activation state is propagated to the output layer, and the units in the output layer update their input potential and activation state;

4. At time t_4, the output vector is read off the output layer.

In addition, most neural models have a *learning rule*, responsible for changing the weights of the network progressively so that it learns to approximate φ given a number of *training examples* (input vectors and their respective target output vectors).

Lastly, there is a special kind of ANN that will be particularly relevant to this work: recursive (recurrent) networks. Recurrent networks differs from regular MLP models by allowing connections that use outputs of network units of layer m at time t as the input to other units of layer $n \leq m$ at time $t + 1$ [32]. Depending on how the recursion is made, several network models have been created, such as Jordan recurrent networks [46] (which has recurrent connections from output neurons to input neurons) and Elman recurrent networks [12] (which has recurrent connections from hidden neurons to input neurons). Both networks has two kinds of input neurons: *feature* neurons, which are used to insert input vectors and *context* neurons, which receive recursive connections from the hidden (Elman networks) or output (Jordan networks) layers.

The set of connections W_{ij} of a neural network is usually the place where its knowledge is stored. Multi-layered feed-forward neural networks, relational neural networks [63], Elman and Jordan recurrent networks [12, 46] and hybrid neural-symbolic systems such as C-IL^{2}P [16], which combines neural networks and propositional logic programs, are all examples of models that stores learned content into its set of weights. But even having a similar way of storing knowledge, each one has its own way of learning through updating weights. The most relevant ones to this work will be presented in the next subsection.

A.2.2 ANN Learning

Neural networks, as a machine learning sub-area, is intended to be a model that learns from examples. In that point of view, what they differ from each other is how and when they update their weights during training. The model that is related to this work is multi-layered neural networks: this study used a three-layer neural network, structured similarly as the network that the neural-symbolic system C-IL^{2}P [16], which uses propositional logic [53], to learn from ILP examples. Thus, in the following, its main learning algorithms will be reviewed.

For learning from examples, the used network in this work will be based on an well-known algorithm called *back-propagation* [18]: an *error* is calculated as the difference between the network's actual output vector and the target vector, for each input vector in the set of examples. This error $\mathbf{E}$ is then propagated back through the network, and used to calculate the variation of the weights $\triangle\mathbf{W}$. This calculation is such that the weights vary according to the *gradient* of the error, i.e. $\triangle\mathbf{W} = -\eta\nabla\mathbf{E}$, where $0 < \eta < 1$ is called

the *learning rate*. This process continues until a predefined stopping criteria is satisfied. The most common ones [52] are:

1. When $\mathbf{E}$ gets lower than a given value δ;
2. When a sufficient number of training examples are being correctly classified by the model;
3. When a number of unseen, labeled examples are being correctly classified;
4. When a maximum number of training epochs has been reached.

A common concern on neural network training is how to allow generalization, in order to proper infer unknown data [5]. In order to achieve that, it is essential that a proper stopping criteria is chosen for a given dataset. Optionally, a *validation set* can be used to check if the system *generalization* ability is not decreasing during training due to excess training (overfitting). One commonly used technique that helps preventing this on large systems, which will be used on this work, is *early stopping* [18]. *Early stopping* uses a validation set to measure data overfitting: training stops when the validation set error starts to get higher than previous steps or optionally, when training set error is not changing much if compared to generalization loss. When one of those happens, the best validation configuration obtained so far is used as the learned model.

There are several possible criteria to be used with early stopping, based on how permissive one wants to be with regard to validation error and on if training error will be considered as well [47]. Before presenting them, some definitions need to be done firstly.

- Generalization loss (GL) at an epoch t:

$$GL(t) = 100 \cdot \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right), \text{ where}$$

  - $E_{va}(t)$ is the validation error on epoch t;
  - $E_{opt}(t) = \min_{t' \leq t} E_{va}(t')$.
- Training strip of length k: sequence of k epochs numbered from $n+1$, up to $n+k$, where n is divisible by k
- Training strip progress (P_k) at an epoch t:

$$P_k(t) = 1000 \cdot \left(\frac{\sum_{t'=t-k+1}^{t} E_{tr}(t')}{k \cdot min_{t'=t-k+1}^{t} E_{tr}(t')} - 1 \right), \text{ where}$$

  - $E_{tr}(t)$ is the training error obtained at epoch t
  - k is the strip size

The three main stopping criteria with early stopping are:

1. Stop after first epoch t with $GL(t) > \alpha$;

2. Stop after first end-of-strip epoch t with $\frac{GL(t)}{P_k(t)} > \alpha$
3. Stop when the generalization error increased in s consecutive strips.

Stopping criterion 1 takes into consideration generalization loss only: when it exceeds a certain threshold α, training stops. On the other hand, stopping criterion 2 evaluates training error as well, letting training go further if training error is decreasing faster than the increase in generalization loss. Lastly, criterion 3 takes a rather different approach: instead of checking if some error-based function reaches a certain threshold, it counts how many consecutive strips have shown a monotonic increase of validation error. When the allowed maximum number is reached, training stops. Regarding which one to choose, each configuration have its advantages and disadvantages. A simpler criterion like the third one will lower training time complexity but it may cause back-propagation to stop prematurely if validation error is just temporarily raising. Figure A.2 below illustrates this case.

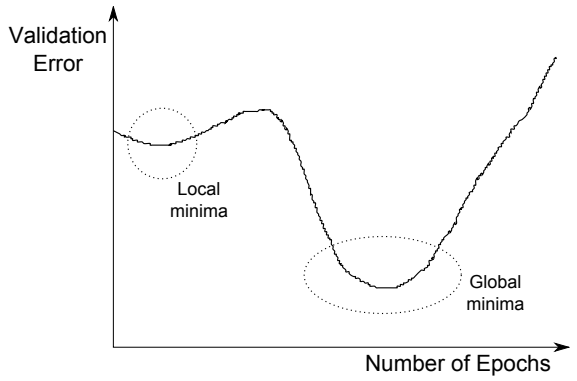

Figure A.2: Validation error behavior during training

Taking this work into consideration, since bottom clauses are extensive representations and thus the input layer of our model is considerably big, it is very likely to overfit training data [5] and a criteria that is focused on evaluating validation error tends to ameliorate this. Thus, the first criteria listed will be the chosen one for be used on this work.

A.3 Feature Selection

As stated in the last section, bottom clauses are extensive representations of an example, possibly having infinite size [60]. Inside bottom clause generation algorithm (Algorithm A.2), one simple way to reduce it is by restricting depth search size, through its $depth$ parameter. By limiting the amount of cycles that the algorithm can use *mode* declarations, it is possible to cut a considerable chunk of literals, although causing considerable information loss.

Clause pruning techniques are another possible approach to get a simpler bottom clause while keeping most of its information and utility as a hypothesis search space

boundary. On [39], two groups of techniques are described: *functional reduction* and *negative-based reduction*. The prior, functional reduction, tries to add the input/output information presented in the language bias (the +/- symbols in parameters) to allow only compatible additions to new clauses. It is imbued on the bottom clause generation algorithm, due to its variable chaining property. The latter, negative-based reduction, is based on analysing not only a positive example, but part of the negative ones as well. After a bottom clause is generated, its last literal is removed: if after this removal, at most *n* negative examples are being covered by it (*n* is a parameter of this reduction method) then the next one is removed as well, until it not covers more than *n* negative examples.

Another approach is to extract features from bottom clauses. Body literals that shares a common body variable can be grouped and be replaced by *structural predicates* [29], which can reduce their size without any loss of information, as long as the structural predicates becomes learning patters as well. In this case, the structural predicate clausal definitions will become part of background knowledge.

Lastly, another viable way of reducing bottom clause size is by using statistical approaches to select most relevant literals, such as Pearson's correlation and Principal Component Analysis (a survey of those methods can be found in [30]). A recent and very interesting method, which has low cost and size complexity while surpassing most common methods in terms of information loss is the mRMR algotithm [10], which has been successfully used in complex problems such as glaucoma diagnosis [67] and gene selection for discriminating different biological samples of different types [66]. It focuses on balancing between minimum redundancy and maximum relevance of features, selecting them by using mutual information I: the mutual information of two variables x and y is defined as

$$I(x, y) = \sum_{i,j} p(x_i, y_j) \log \frac{p(x_i, y_j)}{p(x_i)p(y_j)}, \tag{A.1}$$

where $p(x, y)$ is their joint probabilistic distribution and $p(x)$ and $p(y)$ are their respective marginal probabilities.

Given a subset S of the features set Ω to be ranked by mRMR, the minimum redundancy condition is

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

and the maximum relevance condition is

$$max\ V_I,\ V_I = \frac{1}{|S|} \sum_{i \in S} I(h, i), \tag{A.3}$$

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. To add one feature from Ω_S to S, [10] proposes two ways of combining the two conditions presented above to select features:

- Mutual Information Difference (MID), defined as $\max(V_I - W_I)$, and
- Mutual Information Quotient (MIQ), defined as $\max(V_I/W_I)$.

What differs MID and MIQ is how sensitive they are to slight changes between maximum relevance and minimum redundancy: the difference between V_I and W_I keeps linear in the former, but in the latter it grows up exponentially and thus, distinguishing between candidate features to be added to S is easier. Additionally, results shown in [10, 66, 67] indicates that MIQ usually chooses better features in a considerable amount of datasets. Thus, the latter will be the function chosen to select features in this work and for sake of simplicity, wherever this work refers to mRMR, it is actually referring to MIQ.

Next, propositionalization will be introduced and one of its main algorithms will be reviewed, RSD.

A.4 Propositionalization

Propositionalization is the conversion of a relational database into an attribute-value table, amenable for conventional propositional learners [23]. Propositionalization algorithms use background knowledge and the examples set to find distinctive features, which can differentiate subsets of examples. In other words, they need to be distinct and significative.

There are two kinds of propositionalization: *logic-oriented* and *database-oriented*. The prior aims to build a set of first-order features that are relevant to distinguish between first-order objects and the latter aims to exploit databases 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. This work introduces a new *logic-oriented* propositionalization technique, named Bottom Clauses Propositionalization (BCP), which consists in generating bottom-clauses for each first-order example and using the set of all body literals that occurs on them as possible features (in other words, as columns for an attribute-value table). In order to evaluate how this approach performs, it will be compared with RSD [65], a well-known propositionalization algorithm.

RSD is a system which tackles the *Relational Subgroup Discovery* problem: given a population of individuals and a property of interest of them, find population subgroups that are as large as possible and have the most unusual distributional characteristics. Its input is a complete Progol-formatted dataset, with background knowledge, example set

and language bias and the output is a list of clauses that describes interesting subgroups inside the example dataset. It is composed by two steps: first-order feature construction and rule induction. The prior is a kind of propositionalization which creates higher-level features that are used to replace groups of first-order literals and the latter is an extension of the propositional CN2 rule learner [6] to make it able to be used as a solver to the problem that RSD is trying to clear out.

RSD is used in many recent work as a state-of-the-art propositionalization algorithm [23, 25, 64] and if using it as such, only the first part is of interest: first-order feature construction. It is done in three steps:

- Identifying all expressions that by definition form a first-order feature [65] and at the same time comply to the mode declarations. Such features do not contain any constants and this is done without looking into the example set;
- Employing constants, by copying certain user-defined features with some variables grounded to constants detected by inspecting the example set. Afterwards, a simple filtering is done by excluding all features that has complete or zero coverage over the example set;
- Generating propositionalized representation of each example using the generated feature set, i.e., a relational table consisting of truth values of first-order features.

On the other hand, if using RSD as a subgroup discovery method, after generating the propositionalized data, its extension of the CN2 algorithm will be used to obtain individual-descriptors, which are clauses that defines individual (or targets) in the dataset. In a way, one can say that relational binary classification problems are a particular problem of relational subgroup discovery, having "positive" and "negative" target concepts as individuals.

A final remark regarding RSD needs to be done: because it is intended to deal with relational subgroup discovery problems, which are individual-centered, it is not intended to deal with target predicates with arity higher than 1. This will become clear in the experiments section at Chapter D.

To conclude this background review, another family of propositional learners that will be used as an alternative to ANN's will be presented: decision trees.

A.5 Decision Trees

Decision Trees [53] are models that intends to create a tree that predicts the value of a target variable based on several input variables. Each interior node corresponds to one of the input variables; there are edges to children for each of the possible values of that input variable and each leaf represents a value of the target variable given the values of the input variables represented by the path from the root to it. A tree can be "learned" by splitting

the source set into subsets based on an attribute-value test. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node has all the same value of the target variable, or when splitting no longer adds value to the predictions. This process of top-down induction of decision trees [48] is an example of a greedy algorithm, and it is by far the most common strategy for learning decision trees from data, although bottom-up approaches can be used as well [3].

Decision trees can be divided into two types:

- Classification trees, which analyzes which classes a given example belongs to;
- Regression trees, which can output real values and thus, reduces input dimensionality.

More specifically about classification decision tree algorithms, their goal is to build a decision tree from class labeled training tuples. They have a "flow-chart-like" structure, where each internal (non-leaf) node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf (or terminal) node holds a class label. Among the most common classification decision trees in the literature, two of them will be focused on: ID3 (Iterative Dichotomiser 3) [48] and its successor, C4.5 [49].

ID3 is a top-down classification decision tree that uses a metric called *information gain* to decide between which attributes will be chosen as splitting points when evaluating data. The basic (greedy) algorithm is as follows: starting from the example with the highest information gain metric (which will be the root label of the tree), the examples set will be split up for each possible value of that attribute, generating paths for each one of them. The leaves of each next path are chosen, again, from information gain and this continues to be done iteratively. When the examples covered by a given path does have only identical classes, it stops expanding nodes. The algorithm stops when all paths have stopped generating new nodes.

The information gain value of an attribute A with regard to a set S of attributes is

$$G(S, A) = E(S) - \sum f_s(A_i).E(S_{A_i})_{i=1}^{m},$$

where:

- m is the number of different values of the attribute A in S;
- A_i is the i^{th} possible value of A;
- S_{A_i} is a subset of S containing all items where the value of A is A_i;
- $f_s(A_i)$ is the proportion of the items possessing A_i as value for A in S;
- $E(S)$ is the *information entropy* of the set S, defined as

$$E(S) = -\sum f_s(j).\log_2 f_s(j)_{j=1}^{n},$$

where:

- n is the number of different values of the attribute being calculated;
- $f_s(j)$ is the proportion of the value j in the set S.

C4.5 builds decision trees from a set of training data in the same way as ID3, using the concepts of information entropy and information gain, but it has several improvements:

- It can handle both continuous and discrete attributes. In order to handle continuous attributes, C4.5 creates a threshold and then splits the list into those whose attribute value is above the threshold and those that are less than or equal to it;
- It is also able to deal with training data having missing attribute values. C4.5 allows attribute values to be marked as "?" for missing. Missing attribute values are simply not used in gain and entropy calculations;
- Handling attributes with differing costs (it is able weight some attribute values, giving them more importance if they appear in a pattern);
- It is able to prune trees after creation. It goes back through the tree once it is been created and attempts to remove branches that do not help by replacing them with leaf nodes, through a lower bound on information gain.

C4.5 will be chosen for this work as an alternative algorithm for learning from bottom clauses, together with C-IL^{2}P's recursive ANN.

Having made this quick review regarding decision trees and more specifically, regarding C4.5, all relevant literature concepts have been presented. The main aspects of both components of the neural-symbolic system developed in this work have been presented, bottom clauses and ANN's. The usage of bottom clauses as learning patterns can be seen as a propositionalization method where each feature is a body literal from at least one of them. Because of that, a brief introduction to propositionalization methods have been presented. Bottom clauses are the ILP element that will allow our model, CILP++, to work with relational data and the way this will be done will be explained in the next chapter.

Apêndice B

Neural-Symbolic Systems

In this chapter a quick introduction to neural-symbolic main aspects will be done, followed by an overview of C-IL^{2}P [16], which this work is based on. This overview includes its entire learning cycle: system building, training, testing and knowledge extracting, as well as a quick discussion about applications.

B.1 Introduction

When a father is teaching his little daughter a new concept like, for example, how to drink a glass of water by herself, there are two ways he can do it. One way is to explain with words, illustrations, images, etc., how to grab an empty glass, how to fill it with pure water and how to properly use it (so she will not mess the floor). Another (and maybe easier) approach is just is by drinking it himself while his daughter is looking at him. By doing it over and over, using different glasses from different cabinets and filling with water coming from different sources, she will be able to mimic most of these actions and to learn slight variations of it by herself.

What is just described above is the two most traditional ways that humans solve problems: by mentally describing how it fits into a category of known situations and analogically associate their correct responses to react properly to this new one (like showing symbolically how situations like "grabbing a glass" or "filling glasses with liquids" can fit into the drinking water task), or by showing examples of correct responses to the given problem (like teaching to a child how to drink water by presenting to her different ways of doing it so she can replicate the movements herself).

In the fictional scenario described, there were plenty of information available: the father had background knowledge, relational concepts linking it and enough examples of correct actions of "drinking a glass of water". This is not what usually happens: some problems will fit better with the type of solution firstly shown above, known as *Explanation-Based Learning* [54] problems and they generally carries plenty of formal descriptions

and facts, but lacks a complete set of examples covering all possible situations. On the other hand, there are kinds of problems which are described only by empirical examples of correct and incorrect solutions, which are usually represented by numbers and does not hold formal symbolic descriptions. Those ones are tagged as *empirical* problems and a more "act like this" kind of solution will fit much better. More formally, these two classes of problems and approaches are fields of research in two areas:

- The previous one is explored mainly by *Logic Programming* researchers through relational systems, which tries to generate theories that can explain a given task and answer to slightly different ones with good accuracy;
- The latter one are explored mainly by connectionists and statistical researchers, who models approximated geometrical curves and regions that best fits the set of given examples, so classification and clustering tasks can be done easily, but without any formal knowledge of the resulting model.

As indicated, every kind of problem has its own characteristics and consequently every system that works with them also shares these differences. At Table B.1, a quick comparison between these two kinds of systems are presented. On it, bold fields represents better suitability to the characteristic at stake. It is important to quote that the descriptions shown below are not absolute for every kind of system which belongs to each group. These are just general principles that are inherited by the kind of problems they usually are made for.

	Explanation-Based	Empirical
Multiple Learning	Efficient one–per–time concept learning, slow learning of multiple concepts	**Efficient parallel learning of multiple concepts**
Noise Robustness	Limited, artificial noise–robustness capabilities	**Natural, method–inherent noise treatment**
Concept Clarity	**Learned concepts formally represented by theories and terms**	Concepts are tangled inside matrices of numerical data
Background Knowledge	**Makes partial or total use of background data for training**	Only empirical examples are used, no previous knowledge is applied

Table B.1: Explanation–Based and Empirical systems comparison

But what if a *explanation-based* model needs to efficiently apply parallel learning? And what could be done if an *empirical* system needs knowledge extraction, so it is possible to know formally what has been learned? These questions started a research

towards a new concept in Artificial Intelligence, the concept of *hybrid systems*. What is expected from it is the ability of suppressing the flaws of the individual systems and keep their advantages.

Different techniques have been tried to be used as sources to new hybrid systems [14], including two vastly used machine learning ones: *Artificial Neural Networks* and *Logic Programming*. From those, C-IL^{2}P [16] was made from and it is a hybrid system that achieved solid classification results in comparison to them.

B.2 C-IL^{2}P

The hybrid system that has been developed on this work is focused on (called CILP++) came through improvements of an earlier one called C-IL^{2}P, which is short for *Connectionist and Inductive Learning and Logic Programming*. It is also based in a previous system, KBANN [61] (Knowledge-Based Artificial Neural Network), which models an initial Jordan ANN with a propositional logic program which represents the background knowledge of a given problem. Therefore, they are neural-symbolic systems based on a combination of the main tools used for the two most traditional ways to learn: Logic Programming (learning by induction of theories through examples and background knowledge) and Artificial Neural Networks (further learning with examples). This subsection is intended to shortly describe its basic structure and operations.

The four stages of working with C-IL^{2}P are:

- System building with background knowledge;
- Network training with examples;
- Data testing/inference;
- Knowledge extraction.

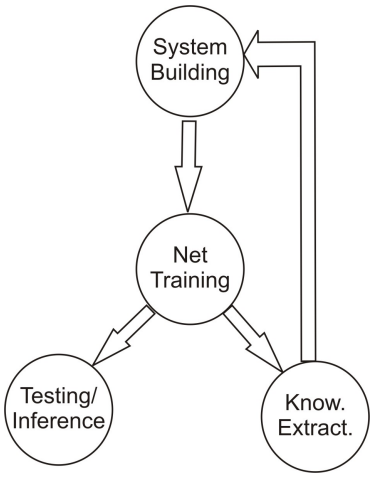

Figure B.1: C-IL^{2}P stages

Their relations with each other can be seen in Figure B.1 and their explanations are given hereafter. In the descriptions of all these four stages, the following notation is used:

Notation	Description
N	Recursive three-layer feed-forward neural network of C–IL^{2}P
n_i	Size of the input layer of N after the building step
n_h	Size of the hidden layer of N after the building step
n_o	Size of the output layer of N after the building step
B	Background knowledge data, with b definite clauses
P	Training dataset, with p clauses
T	Testing dataset, with t clauses

Table B.2: C-IL^{2}P used notation

B.2.1 C-IL^{2}P Building

C-IL^{2}P's building stage is basically a translation of a definite logic program into a recursive ANN. This way, it is then possible to apply parallel learning using standard *back-propagation* algorithm (see 32) and test the newly-learned data using its recursive connections to process chains of reasoning included as sequences of consequences over clauses (for a more theoretical insight of this feature, see 16). The usage of a recursive ANN is one of the enhancements that C-IL^{2}P made upon KBANN, which had to add extra hidden layers on the ANN in order to represent those sequences of consequences. With that, a simpler and more efficient learning can be achieved.

A more detailed build stage algorithm can be found in Algorithm E.1. In a simplified fashion, it is done by applying the following steps:

For each clause c of B, do

1. Add a neuron h in the hidden layer of N and label it as c;
2. Add input neurons to N with labels corresponding to each literal atom in the body of c;
3. Connect them to h with a given weight W if it the associated literals are positive, $-W$ otherwise (see Table E.1);
4. Add an output neuron o in N and label it as the head literal of c;
5. Connect h with o, with a given weight W;
6. If o's label is identical to any of the labels of the input layer ones, add a recursive connection between then with fixed weight 1.

A more visual way of understanding this building process can be seen in Figure B.2.

In the example above, firstly a hidden neuron is created to represent clause 1. It has two literals on its body, so two input neurons representing B and C will be created and

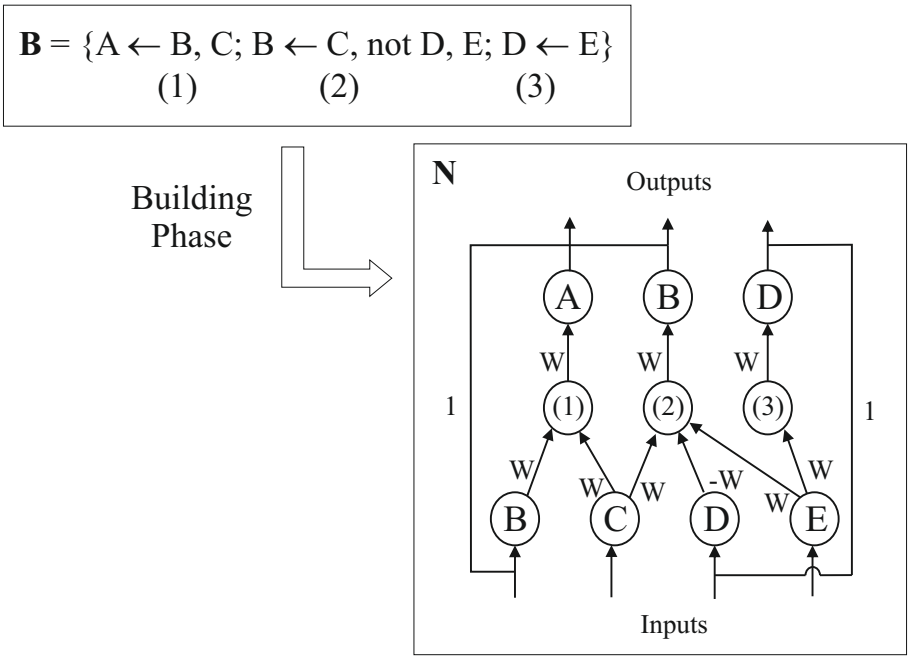

Figure B.2: System building

connected to it with weight W. Since its head is A, it will be connected to a new output neuron labeled A as well, with a weight W on it.

Afterwards, clause 2 is used by the building algorithm: a second hidden neuron is created and all its body literals (C, *not* D and E) are mapped into input neurons. since C already exists, no new neuron is created for it: it is just connected to the newly created hidden neuron with weight W. In the case of *not* D, since it does not exists, a new input neuron will be created and then it is connected in the same way as in the previous case. Additionally, since it is a default negated literal, its connection to the hidden layer will receive weight $-W$. On the other hand, for the body literal E, it is created and connected with weight W. Lastly, a new output neuron is created and associated with the head literal B of clause 1 with weight W. Since it already exists in the network as input neuron, a recursive connection with fixed weight 1 will be created, linking them.

Finally, clause 3 induces the creation of a third hidden neuron, one more input neuron corresponding to its only body literal E and one more output neuron, D. Again, since D already exists in the input layer, a recursive connection will be created. All non-recursive weights of this step will receive weight W and the recursive will receive 1.

C-IL^{2}P can deal with any kind of logic program (even for programs with negated heads, an extended version of the building algorithm has been made in order to deal with it [15]). On the other hand, KBANN cannot deal with logic programs that contains disjunctions with more than one head (e.g. on the logic program P = {A :- B, C; A :- C, D}, the atom A can be true if the first clause **or** the second is satisfied). If two clauses shares a common head atom and have more than one body literal, the logic program needs preprocessed in order to rename them. If KBANN had to use P as background knowledge

to build an initial ANN, firstly all the occurrences of repeated heads would be renamed. Since A is repeated once, it would be renamed A' in the second clause. Further, in order to maintain consistency in the program, a third clause linking both A and A' would be added to the program: A :- A'. Thus, the preprocessed program would be P' = {A :- B, C; A :- A'; A' :- C, D}.

After finishing building the initial structure for N, it is ready to receive training examples.

B.2.2 C-IL^{2}P Training

In the training stage, examples are shown to the C-IL^{2}P's underlying network and a basic back-propagation algorithm is applied, fixing all recurrent weights to 1. These examples are extended logic programming clauses [53], which can work with default negation ("$\sim$"). Additionally, they will indicate which input neurons (or which body literals) will be activated (or set as true), deactivated or ignored, depending on the presence of a non-negated literal, a negated literal or an absence of a literal, respectively. Additionally, if there are concepts that are not present on any input vector which are represented in the network after the building step, a query based on *modus ponens* [59] will be made to complete them before training starts.

An illustration of how the training pattern that C-IL^{2}P works with is can be seen in the two examples shown in Figure B.3.

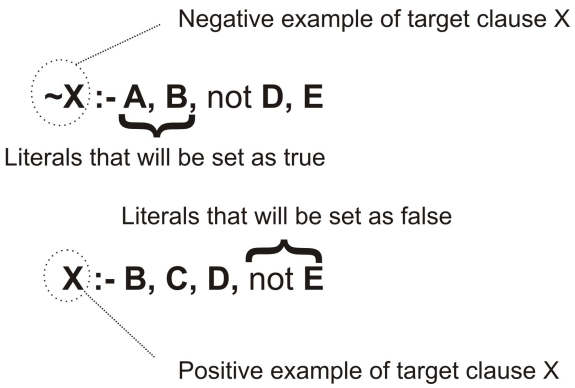

Figure B.3: Training data structure example

The training stage of C-IL^{2}P is basically a *back-propagation* method, but with some pre-processing to transform examples into input vectors and a slightly different stopping criteria. The transformation of an example e consists in representing it by a real-valued vector of dimension n_i, where every position corresponds to an input layer neuron of N, all positions corresponding to input neurons with same label as a positive body literal of e will receive 1.0 and every other position will receive -1.0. A more complete procedure is given in Algorithm E.2.

Following, the stopping criteria for C-IL^{2}P training are three:

- **Stopping criteria 1:** maximum allowed number of training epochs[1] reached;
- **Stopping criteria 2:** a high proportion of training examples has reached a desired general training accuracy;
- **Stopping criteria 3:** enough training accuracy has been reached and it has been not improving for several epochs;

The complete training algorithm can be seen in Algorithm E.4. To give a general idea of the process, it can be seen as a three-step procedure:

1. Convert each example of P to a pair of a input vector and its respective label;
2. Apply sequential[2] *back-propagation* on N training using each input/label pair;
3. Test the updated N using P and use the obtained results to verify stopping criteria.

Every *back-propagation* training parameter used varies with the dataset it is trying to analyze. Because of that, *back-propagation* related variables will be presented only together with their correspondent experiments.

At this point, C-IL^{2}P is ready to test/infer or have its newly learned knowledge to be extract into new clauses.

B.2.3 C-IL^{2}P Testing/Inferring

After C-IL^{2}P's building and training stages, all information provided by background knowledge and examples are combined within the weights of N. As commented earlier, recurrent connections were added to iterate through chains of clauses, allowing it to obtain conclusions that it otherwise would not be able to get. Because of that, a testing procedure would have to iterate through these recurrences to find a stable activation of neurons, so a final answer could be obtained. As can be seen in [16], this stabilization process is guaranteed to achieve a stable state as long as the underlying logic program is consistent. This way, a simple stabilization process to make inferences can be defined in the steps below.

1. Insert the real-valued input vector related to the used testing data;
2. Make one *feed-forward* step to obtain one output vector;
3. Feed each input neuron which receives recurrent connections with their correspondent output neurons activations;
4. *Feed-forward* again the new input vector step to obtain another output vector;
5. If the activation states[3] of the output neurons are identical to its related ones in the input layer, then **go to step 7**;
6. If not, **go back to step 3**;

[1] It is understand as a training epoch a full pass in P, using all examples.
[2] *Sequential* is the name given to *back-propagation* training where each training data is used one by one.

7. Return the current output vector as C-IL^{2}P's answer.

In case of testing, the following steps should be added to the steps above (or should replace the ones with same number):

7. Compare the outputs related to the testing data correct answers in the following way:
   (a) If such output and its related testing answer are both in an active or inactive state, consider it *a hit*;
   (b) Otherwise, consider it *a miss*;
8. Sum up the hits and answer $\frac{hitSum}{sizeOfTestingData}$ as the testing accuracy.

Again, if interested in a more complete testing algorithm, the reader should read Algorithm E.5.

B.2.4 Extracting Knowledge

In the fourth and last stage of C-IL^{2}P's processing cycle, a gathering of all acquired knowledge during building and training stages will be made in the form of logic clauses. All the work done in this way can be seen in [13] and a summary about it will be shown hereafter.

As seen in the building stage of C-IL^{2}P, any definite logic program can be used to build a recursive three-layer neural network, in a way that relations in the program are directly translated into connections between neurons. After that, in the training step, a set of examples is given to the system in order to let it learn new concepts. As one can imagine, all the weight changes caused by the *back–propagation* algorithm will probably change drastically the relational representation between literals in the net. This is what makes knowledge extraction of C-IL^{2}P system a challenging problem and, in a more general way, this parallel weight updating during training makes knowledge extraction of any neural network a troublesome task.

There are two kinds of knowledge extraction algorithms for neural networks: *pedagogical* and *decompositional* ones. The first treats the neural network as a "black box", analyzing different combinations of inputs and the outputs obtained and building "if-then-else" rules. The second tries to extract knowledge from smaller parts of the network and then assembles them together to create a unified theory.

What [13] proposed as a knowledge extraction solution for C-IL^{2}P was a decompositional procedure that splits the network into "regular" ones, which are characterized for

[3]The meaning of *activation state* here is the state of the neuron regarding the value of its activation: if it is above a calculated parameter A_{min}, $0.0 \leq A_{min} < 1.0$ (see Table E.1), then its state is *active*. If it is below $-A_{min}$, it is *inactive*. Otherwise, it is *unknown*.

not having relations coming from the same neuron with different signals. It is shown that the extraction for those nets are sound and complete, and if it was the case of needing to do that splitting (the case of the complete network being non-regular), soundness could be achieved still, but with no guarantees of completeness.

For this stage, only a intuitive algorithm will be given. For a formal and complete algorithm, see [13]. In the following, some of the notations specified in Table B.2 are being used.

1. Split N into a set of smaller, regular subnets in the following way:
   (a) Split the "input-to-hidden" part of N into n_h subnets, each one containing all the input neurons and one hidden neuron;
   (b) Split the "hidden-to-output" part of N into n_o subnets, each one containing all the hidden neurons and one output neuron;
2. Extract all "if-then-else" possible relations from each of these regular networks;
3. Use the soundness and completeness of the extraction above to infer "if-and-only-if" rules from the "input-to-hidden" subnets;
4. With that, assemble rules obtained from "input-to-hidden" and "hidden-to-output" subnets to build an unified set of rules;

B.2.5 Applications

C-IL^{2}P has the ability to build an initial model with a propositional background knowledge and refine it with examples, through back-propagation learning. Because of that, any propositional domain in which preliminary info is available is a potential application.

On its original paper, C-IL^{2}P investigated two of such datasets, from the gene sequences benchmark: Promoter Recognition and Splice-Junction Determination [62]. Both intends to locate distinctive regions on DNA strings: promoter DNA sequences, which identifies the start of a gene and splice-junction spots, which marks where the DNA will be split during protein creation.

On both datasets, C-IL^{2}P got able to achieve expressive accuracy, hitting 92.8% on Promoter Recognition and 94.8% on Splice-Junction Determination and surpassing KBANN, which obtained 92.0% and 94.8%, respectively.

Besides C-IL^{2}P and KBANN, there are other related work which will be discussed next, in the last section of this chapter.

Apêndice C

Using Bottom Clauses in Neural Networks

C-IL^{2}P, which was introduced in Section B.2, is composed by two models: propositional logic programs and ANN's, explained in the Section A.2. By extending it to deal with relational data, it becomes able to be applied on problems dealt by ILP, presented in Section A.1. The way it is done is by using one element of the ILP system Progol theory-induction algorithm: bottom clauses. The extension of C-IL^{2}P that was developed on this work, which is called CILP++, will firstly apply Bottom Clause Propositionalization (BCP) on the examples set and then, use the propositionalized data as learning patterns for its recursive ANN.

As briefly commented in the introduction, bottom clauses literals can be used directly as propositional features. After having presented Progol's bottom clause generation algorithm on Section A.1, this statement can now be explained.

Progol's bottom clause generation algorithm uses a *hash function* to keep track of each correlation between constants and variables. We will modify this algorithm to allow sharing of its *hash* between all examples and with that, if e.g. a variable "A" is associated with a constant "mom1" in a given example, this association ($A \leftrightarrow mom1$) will be the same for all other ones. With that, if two bottom clauses contains a literal with the same predicate description and with the same attributes, it is ensured that their "texts" will be the same and they can be seen a feature which they have in common. For instance, consider two bottom clauses b_1 = [$motherInLaw(A, B) \leftarrow$ *parent(A, C)*, *wife(C, B)*] and b_2 = [$motherInLaw(A, B) \leftarrow$ *parent(A, C)*, *wife(C, D)*]. They share a common literal, *parent(A, C)* which, by the *hash* function sharing modification, is guaranteed to have been generated from the same attributes. Thus, this modification allows body literals to be considered as propositional features.

In this chapter, the CILP++ system will be introduced, which is capable of working with first-order data through BCP and learn from them with back-propagation. Firstly, an introductory section will be presented, showing which differences and contributions

CILP++ have with regard to C-IL^{2}P. Afterwards, BCP will be explained and furthermore, the CILP++ version of C-IL^{2}P's learning cycle which has been implemented will be discussed: building, training and testing. Knowledge extraction involves more research about what the clauses that will be extracted will represent, but an initial thought about how to start this has been done and it will be shown in the discussion of Chapter 6, but all theoretical considerations, implementation and empirical evaluation will be left as future work. To conclude this, some related work will discussed as well.

C.1 CILP++

CILP++ is an extension of the C-IL^{2}P system, described in Section B.2 and the practical result of this work. Its main contributions and improvements are:

- Ability to work with first-order logics through bottom clause propositionalization;
- Improved accuracy, reduced computational time and noise robustness;
- A weight normalization that keeps the integrity of its background knowledge;
- A open-source, configurable and automatic tool (CILP++), containing other minor improvements over C-IL^{2}P, including deduction through recursive feed-forward iteration and stratified folding;

As an extension of C-IL^{2}P, CILP++ also builds a recursive three-layer neural network by using a logical background knowledge. But it is capable of working with first-order clauses as well, using one feature brought up in [37]: search-space restriction through the most specific clause, called *bottom clause*. It is capable of using body literals of bottom clauses directly as propositional features, as "most noisy" inputs for the network. Then, it can optionally select a feature subset for those inputs that can reasonably represent all major concepts embedded in them. It is interesting to make just a "rough" filtering and let a little noise in the data, since zero-noise learning is not as effective as if it was slightly noisy [68], but still reducing considerably input complexity in order to not compromise training approximation [30].

For comparison sake, we ran CILP++ on the Promoter Recognition dataset, in the same conditions described in [16], to see if it is able to successfully reproduce C-IL^{2}P previous results. We obtained 92.1%, which is almost the same as the 92.8% reported for C-IL^{2}P.

Having transformed all first-order examples into pre-processed bottom-clauses, C-IL^{2}P training procedure can be applied, in a similar way as described in [16]. All adaptations and particularities will be described in their corresponding subsections.

C.1.1 Bottom Clauses Propositionalization

The first step for relational learning with CILP++ is BCP. Each instantiated target clause like $target(a_1, \ldots, a_n)$ will be converted to something that a standard neural network can use as input. In order to achieve that, firstly each example is transformed into a bottom clause, which contains all possible concepts that can be related to it, as a starting point to a proper representation and afterwards, it is directly mapped as features on an attribute-value table and numerical vectors are generated for each example. Thus, BCP has two steps:

1. Bottom clauses generation;
2. Attribute-value mapping.

In the first step of BCP, each example is put into Progol's bottom clause generation algorithm [58], in order to create a corresponding bottom clause representation for them. To do so, a slight modification in Progol's bottom clause algorithm [58] is needed. This modified version is shown in Algorithm C.1. This modification was needed because of two reasons:

- To allow the same *hash* function to be shared between all examples, in order to keep consistency between variable associations;
- To allow negative examples to have bottom clauses as well. The original algorithm can deal only with positive examples.

If Algorithm C.1 is executed with $depth = 1$ for the negative example of Section A.1.1, it will generate a bottom clause $\bot_e$ = $\sim$ *motherInLaw(A, B) :- wife(A, C)*. What it did differently from Algorithm A.2 was adding an explicit negation signal ("$\sim$") in the end. With both examples having their bottom clauses generated, the post-processed training set, after the first step of BCP, becomes

$$S_\bot = \{motherInLaw(A, B) : -parent(A, C), wife(C, B);$$
$$\sim motherInLaw(A, B) : -wife(A, C)\}.$$

It is important to notice that the only parameter when using BCP, which is $depth$, controls how far the algorithm will go when generating concepts chaining. Its value controls how many cycles through the $modeb$ declarations will be done, to add more predicates on the generated bottom clause. Therefore, the $depth$ parameter controls how much information loss BCP will have when converting first-order examples into propositional patterns and thus it will be categorized as a heuristic propositionalization method (which was explained on Section A.4.

Algorithm C.1 Adapted Bottom Clause Generation

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

After the creation of the $S_\perp$ set, the second step of BCP takes place: each element of $S_\perp$ (each bottom clause) will be converted to an input vector v_i, $0 \leq i \leq n$, that an ANN can process. The algorithm that was implemented for that is the following:

1. Let $|L|$ be the number of distinct body literals in $S_\perp$;
2. Let S_v be the set of input vectors, converted from $S_\perp$, initially empty;
3. For each bottom clause bot_e of $S_\perp$ do
   (a) Create a numerical vector v_i of size $|L|$ and with 0 on all positions;
   (b) For each position corresponding to a body literal that occur on $\perp_e$, change its value to 1;
   (c) Add v_i to S_v;
4. Return S_v;

As an example, suppose that the bottom clause set $S_\perp$ will be pre-processed by the second step of BCP. $|L|$ would be equal to 3, since the literals found on it are $parent(A, C)$, $wife(C, B)$ and $wife(A, C)$. For the positive bottom clause, a vector v_1 of size 3 would be initialized with 0 and further, it will have all positions corresponding to literals found on the currently processing bottom clause. Thus, its first position (corresponding to $parent(A, C)$) and second position (corresponding to $wife(C, B)$) would receive value 1, resulting in a vector $v_1 = (1, 1, 0)$. With regard to the negative example, it has only the third of the three body literals found in $S_\perp$ and because of that, its final vector would be $v_2 = (0, 0, 1)$.

Regarding what exactly this numerical pattern (and learning from them) means, in ILP terms, training with bottom clauses can be seen as learning to fit all possible ILP candidate clauses to be theories as belonging or not to a given class, from the ILP perspective, due to what a bottom clause is. Bottom clauses are the most specific clauses which are acceptable by an inverse-entailment based hypothesis induction algorithm [37]. When using them as input patterns, their meaning on ANN training still holds: a pattern containing value *1* (meaning that all possible body literals were found in the body of the corresponding bottom clause) in all positions represents the most specific clause that can be represented by all literals from all bottom clauses. On the other hand, a training pattern containing only *0* in all positions (meaning that there is no literal in the corresponding bottom clause) represents the most general clause. Lastly, any pattern with a mix of 0's and 1's represents clauses between those two bounds. When learning from them, CILP++ tries to model an ANN which is capable of correctly label all patterns as best as possible and thus, choosing which hypothesis can be associated with positive examples and which ones cannot.

As a final remark, since bottom clauses are built from a problem specific knowledge base and a (possibly infinite) number of possible instantiations of them, $|L|$ is considerably big. Because of that, an analysis of how feature selection can contribute to learning is done in Section A.3. In the following, the learning cycle of CILP++ will be presented.

C.1.2 CILP++ Building

After BCP has taken place, the original C-IL^{2}P building algorithm will take place to build a starting network, treating each variablized bottom clause literal as a single propositional term. Since C-IL^{2}P uses background knowledge to build the network and only bottom clauses are used for training, it will be necessary to have a proportion of the example set as background knowledge clauses (this subset will be called $S_\perp^{BK}$). It is important to notice that even having used examples to build an initial model, they still need to be reinforced during training: when back-propagation training is running, initial weight configurations tend to influence less and less the final outcome and thus, using examples exclusively to build an initial model would actually reduce training set size and

cripple CILP++'s approximation capabilities [18]. But even after stating that "building examples" would not accomplish anything without its reapplication when learning from examples, this initial configuration is still handy: it allows CILP++ to converge much faster. One can think on this initial model as a "smart start" for the network.

The pseudo-algorithm for the building is as follows.

For each bottom clause $\bot_e$ of $S_\bot^{BK}$, do

1. Add a neuron h in the hidden layer and label it as $\bot_e$;
2. Add input neurons with labels corresponding to each literal atom in the body of $\bot_e$;
3. Connect them to h with a given weight W if the associated literals are positive, $-W$ otherwise;
4. Add an output neuron o and label it as the head literal of $\bot_e$;
5. Connect h with o, with a given weight W;
6. If o's label is identical to a label in the input layer, add a recursive connection between them with fixed weight 1.

Alternatively, we wanted to see how important is this starting set-up, regarding C-IL^{2}P: in the propositional version, it is clearly important because its building algorithm could use it fully, in order to get a well-conditioned network. This is not the case on this work: the subset $S_\bot^{BK} \subseteq \bot_e$ is just a sample of the examples dataset, it is not specifically a background knowledge. Thus, in order to evaluate that, we experimented a second version of the pseudo-algorithm, where only the input and output layers are built and we set up a specific number of initial hidden layer neurons, but just for convergence purposes: no initial background knowledge will be represented.

C.1.3 CILP++ Training

After preparing the bottom clauses and building the network, ANN training will take place. It is mostly similar with C-IL^{2}P [16], learning with back-propagation and not training its recursive connections. Alternatively, CILP++ can use back-propagation with *validation set* (as explained in Subsection A.2) and *early stopping*: we use the validation set to measure generalization error during each training epoch. As soon as it starts to go up the training is stopped and the best model in terms of validation error is returned as the optimal. We apply a more "permissive" version of early stopping [47]: instead of finishing the training immediately after validation starts to go up, we stop when the criterion below is satisfied, where α is the stopping criteria parameter, t is the current epoch number, $E_{va}(t)$ is the average validation error on epoch t and $E_{opt}(t)$ is the least validation error obtained on epochs 1 up to t. The reason that it was chosen by us for this system is because of the complexity of the generated bottom clauses: early stopping is

really good to avoid overfitting [5] when the system complexity is a lot bigger than the numbers of examples. Since it is not guaranteed to converge, an additional criterion is added: the network will stop training after 100 epochs.

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

Given a bottom clause set $S_{\perp}^{train}$, the following steps will be done in order to learn from them:

1. For each bottom clause $\perp_e \in S_{\perp}^{train}$, $\perp_e = h$:- $l_1, l_2, ..., l_n$, do
   (a) Add all $l_i, 1 \leq i \leq n$, that are not represented yet in the input layer 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 so is required to allow proper learning;
3. Make the network fully-connected, by adding weights with zero value;
4. Apply a small disturbance to all weights and biases, to avoid the symmetry problem[1];
5. Normalize all weights and biases;
6. Apply back-propagation training using each bottom clause $\perp_e \in S_{\perp}^{train}$, converted to numeric vectors[2];

To illustrate this process, assume that the bottom clause set $S_{\perp}$ obtained in Subsection A.1.3 is our training data and no background knowledge has been used. In step 1.a, all body literals from both examples [$parent(A, C)$ and $wife(C, B)$ from the positive example and $wife(A, C)$ from the negative one] would cause the generation of three new input neurons in the network, with label identical as its corresponding literals, since our starting network is empty due to the fact that no background knowledge has been used. Following, on step 1.b, an output neuron labeled $motherInLaw(A, B)$ would be added. For step 2, assume that just one hidden neuron will be added. Next, on step 3, add zero-weighted connections from all the three new input neurons to the single hidden neuron and from this neuron to the new output neuron. The disturbance on step 4 needs to be big enough to avoid the symmetry problem but small enough to not significantly influence training and to achieve that, we have used a random non-zero value between $[-0.001, 0.001]$. On step 5, all weights have been normalized in order to obtain a better conditioned network, since it improves considerably back-propagation learning performance [18]. Lastly, on

[2]The symmetry problem [18], in neural networks context, states that identical weights will have similar updates

step 6, back-propagation learning will be used on both examples, firing the input neurons $parent(A, C)$ and $wife(C, B)$ when the positive example is being learned and firing $wife(A, C)$ when the negative one is used.

Specifically about the network normalization [18]: let w_l be a weight in a layer l and similarly, let b_l be a bias. For each l,

$$w_l = w_l \cdot \frac{1}{(|A_l|^{\frac{1}{2}}) \cdot max_w} = w_l \cdot N_l$$

$$b_l = b_l \cdot \frac{1}{(|A_l|^{\frac{1}{2}}) \cdot max_w} = b_l \cdot N_l$$

where,

- max_w: maximum absolute value of w, among all weights in the network;
- N_l: normalization factor;
- t: current epoch;
- $E_{va}(t)$: average validation error on epoch t;
- $E_{opt}(t)$: least validation error obtained on epochs 1 up to t.

However, as pointed out in Section A.2, the activation state of an neuron is defined by A_{min}. This implicates that when weight normalization is done, the activation inequality

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

does not holds anymore. This means that when evaluating a neuron, A_{min} needs to be processed somehow, in order to keep the inequality above correct. Following, it is shown what was done to correct this issue.

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

$$\Rightarrow h_n(\sum_{\forall i} w_i \cdot x_i + b) \cdot h_n(\sum_{\forall i} w_i \cdot N_l \cdot x_i + b \cdot N_l) \geq$$

$$A_{min} \cdot h_n(\sum_{\forall i} w_i \cdot N_l \cdot x_i + b \cdot N_l) \tag{C.2}$$

$$\Rightarrow h_n(\sum_{\forall i} w_i \cdot N_l \cdot x_i + b \cdot N_l) \geq A_{min} \cdot \frac{h_n(\sum_{\forall i} w_i \cdot N_l \cdot x_i + b \cdot N_l)}{h_n(\sum_{\forall i} w_i \cdot x_i + b)} \tag{C.3}$$

From step (1) to (2) each side of the inequality was multiplied by the activation function term, but with normalized weights and bias. Afterwards, in step (3), we isolated the activation with normalized weights and found how A_{min} changed. The term $\frac{h_n(\sum_{\forall i} w_i \cdot N_l \cdot x_i + b \cdot N_l)}{h_n(\sum_{\forall i} w_i \cdot x_i + b)}$ will be the normalization factor for A_{min}: every time a neuron needs

to be evaluated (for example, when one needs to know if an output neuron is considered to be activated or not), this factor must be calculated and A_{min} must be multiplied by it in order to have a correct evaluation of a neuron state.

After training, if one want to follow C-IL^{2}P learning cycle (Figure B.1), the next steps would be inferring unknown data or extracting knowledge from what has been learned from the network. Since the latter was not implemented by this work and is left for future work, only the prior will be explained on the next subsection.

C.1.4 CILP++ Testing/Inferring

Inference in CILP++ is done only in the ANN level: the pseudo-algorithm below describes in details how this is done (assume that the set of testing data is $S_{\perp}^{test}$):

For each bottom clause $\perp_e$ of $S_{\perp}^{test}$, do

1. Feed each input neuron corresponding to a body literal of $\perp_e$ with 1 and all other inputs with 0;
2. $Stabilized$ = $false$;
3. While $Stabilized$ is $false$ do
   (a) Let I be the set of activated[3] input neurons;
   (b) Apply a standard feed-forward iteration in the network, as described in A.2;
   (c) Let O be the set of activated output neurons;
   (d) If $I \equiv O$, $Stabilized$ is $true$;
   (e) Otherwise, for each recursive connection in the output layer, propagate their output values back to the input layer;
4. Return the activation states of all output neurons;

Having presented CILP++, some main related work with regard to it will be presented.

C.2 Related Work

This section will be divided into three related work categories: other approaches that use bottom clauses, other propositionalization approaches and other hybrid systems. The first one will cover the most relevant applications which uses bottom clauses for any other application different from hypothesis search boundary; the second one will group up all main propositionalization methods in the literature; and the third one will have a survey over the most important hybrid systems other than KBANN or C-IL^{2}P.

[3] A neuron is considered to be *activated* (or deactivated) when its activation potential is higher (or lower) than A_{min} (or -A_{min}).

C.2.1 Approaches that Uses Bottom Clauses

Bottom clauses have already been used on ANN's as learning patterns before [9], but to build an efficient hypothesis evaluator, not to actually classify first-order examples. On it, a subset of examples is chosen, their bottom clauses are generated and they are used together with auxiliary parameters related to it such as the label of target predicate, number of literals, number of distinct variables on it, and so on. This work is motivated by the fact that literal choosing heuristics for theory refinement is a known bottleneck for ILP learning algorithms [37].

Also, the QG/GA system [42] introduces a new hypothesis search algorithm in ILP, bottom-up, called Quick Generalization (QG), which performs random single reductions in bottom-clauses to generate candidate clauses for hypothesis and additionally, it suggests using Genetic Algorithms (GA) in those generated clauses to further explore the search space, converting them to numerical patters in the same way we do: each occurrence of a literal is mapped as "1" and non-occurences as "0".

Besides using them as learning patterns, the work on [11] used them for their original purpose (hypothesis search boundaries), but to deal with theory revision [50]. Theory revision is the task of updating and improving pre-existent domain theories, using a set of *refinement operators* which are responsible for adding and removing literals from the theories, with a set of training examples and it relies on the premise that one cannot expect that theories are entirely complete or correct. The contribution of the work on [11] was the integration of bottom clauses and language bias to the refinement process: they were used by the refinement operators to constrain what could or could not be added or removed from a theory clause and a considerable performance improvement over previous methods was found, together with a more comprehensible set of revised theories.

C.2.2 Other Propositionalization Approaches

Propositionalization, as explained in Section A.4, is the extraction of propositional features from a relational dataset, allowing common machine learning models to learn from them. Among other propositionalization approaches, three will be explained hereafter: RelF [26], which is a recent approach that is considered to be the successor of RSD; LINUS and its successors [28] and RELAGGS [24].

As explained in Section A.4, RSD's propositionalization algorithm generates an exhaustive set of features, constrained by a language bias, and complying with user-defined constraints. Afterwards, it discards features that are not related to any of the examples set or that are true for all examples. Its successor, RelF, takes a more "classification-driven" approach, by only considering features that are interesting for distinguishing between classes. It also discards features which θ-*subsumes* any previously generated feature. Those two reductions are not done as a separate pruning step, like RSD: in RelF,

it is done while generating features.

In opposite to the youth of RelF, LINUS [28] was the first system to introduce propositionalization. It worked with acyclic and function-free Horn Clauses, like Progol, BCP and RSD, but differently from them, it constrained the first-order language further to only accept *DBHB* clauses, allowing the possible variable occurrences in an acceptable clause to have head variables only. Its successor, DINUS [22], allows a larger subset of clauses to be accepted (determinate clauses), allowing clauses having body literals with variables not appearing in the head literal to be accepted, but still allowing only one possible instantiation of those variables. Finally, SINUS [22] improved on DINUS by dealing with the same clauses type as BCP, making use of language bias to constraint 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 simplifying the final number of features. LINUS and DINUS treat body literals as features, which is a similar approach to BCP. However, BCP can deal with the same language as Progol, thus having none of the language restrictions of LINUS/DINUS.

Lastly, RELAGGS takes a completely different approach for propositionalization, categorized as *database-oriented* [23]. It uses standard SQL aggregate functions (basically four: *average*, *minimum*, *maximum* and *sum*, but depending on the problem, additional aggregations such as *standard deviations* or *ranges* can be used) as features and foreign keys as chaining between relations (predicates).

C.2.3 Other Hybrid Systems

There is a vast diversity of hybrid systems and subareas that aims to learn relational data with numerical models, such as Markov Logic Networks [51] (Markov Random Fields + ILP, MLN), Statistical Relational Learning [17] and SVILP [40] (Support Vector Machines + ILP).

The first one, MLN, is a collection of formulas from first order logic, to each of which is assigned a real number, the weight. Taken as a Markov Random Field [21] (or Markov network), the vertices of the network graph are atomic formulas, and the edges are the logical connectives used to construct the formula. Each formula is considered to be a clique, and the Markov blanket is the set of formulas in which a given atom appears. A potential function is associated to each formula, and takes the value of one when the formula is true, and zero when it is false. The potential function is combined with the weight to form the partition function for the Markov network and thus allow it to infer unknown data.

For the other two approaches: the second one, Statistical Relational Learning, uses (a subset of) first-order logic background knowledge to describe relational properties of a domain in a general manner (universal quantification) and draw upon probabilistic graphical

models (such as Bayesian networks or Markov networks) to model the uncertainty; and the third one, SVILP, is an approach that combines logic-based rules with support vector numerical prediction, using logic rules as inputs to generate a kernel for use by SVM.

Besides MLN, another recent relational hybrid system that deals with uncertainty worth mentioning is called BLOG [31]. Instead of bringing relational learning to a probabilistic framework, like what MLN does, BLOG works in the other way around by adding probabilities to possible worlds (possible background knowledge definitions). By doing that, the learning task of BLOG is, instead of performing top-down approaches such as sequential covering [53] or bottom-approaches such as GOLEM [38], to calculate posterior distributions for each class, given an example to be queried and a possible world.

A different approach for integrating ANN's and first-order logic is through relational databases: a theory can be represented structurally by a network linking each relation. With this approach, each layer would represent entries of a given entity and each connection would represent a relation between them [63]. Two models are compared, RelNN's and GNN's. They mainly differ on computational effort and what kind of relation graphs they can process: the prior can efficiently process acyclic graphs, but not cyclic ones (although the work explains a sound way of dividing it into a set of relational trees). On the other hand, the latter can process all kinds of graphs but it is more time-consuming than the first approach.

An additional work aiming at neural-symbolic integration using relational networks based in ARTMAP models has been done on [4] to learn from first-order data. Those models are composed by two ART networks [18], where one is responsible for clustering body literals and the other clusters head literals, and a mapping module which creates relations between those networks to build candidate hypothesis.

Having introduced all relevant topics to this work, the next chapter will show what CILP++ has been able to achieve empirically, comparing with both Aleph and by using BCP together with Quinlan's C4.5 algorithm instead of CILP++'s recursive ANN.

Apêndice D

Experimental Results

In this section we will present our experimental methodology and results of CILP++ compared to a freely-distributed and well-known ILP system, Aleph. Aleph is a freely distributed ILP system, which has a considerable amount of state-of-the-art algorithms as built-in, such as Progol (by default). We have chosen two benchmarks: the mutagenesis dataset [57] and the *alzheimer* benchmark [20], which consists of four datasets: *amine*, *acetyl*, *memory* and *toxic*. From now on, we will refer to these datasets as *alz-amine*, *alz-acetyl*, *alz-memory* and *alz-toxic*, to distinguish those datasets better from other benchmarks, e.g. mutagenesis. We will also present an alternative to the recursive ANN of CILP++: we will just apply BCP on the training data and use the C4.5 decision tree (presented on Section A.5) as classifier. We will call this approach "BCP+C4.5".

We ran CILP++ on them in several different configurations, to make comparisons between different models and evaluate better how our approach for first-order logic neural-symbolic integration performs. Three analysis will be made:

- **Accuracy vs. time**, using the original Alzheimer datasets;
- **Accuracy when doing feature selection**, evaluating how CILP++ behaves with selection of features in a both logical way, by constraining the maximum clause length when building bottom clauses, and a statistical way by using mRMR.
- **Comparison with RSD**, comparing accuracies of both CILP++ and RSD when classifying with their native learners (CILP++'s recursive ANN and RSD's CN2 rule learner).

Additionally, some considerations about dealing with noise will be done as well. When dealing with ILP and nominal datasets such as the mutagenesis and the four alzheimers datasets, it is not possible to apply additive noise and only the multiplicative one is viable.

Two important remarks need to be made:

1. ANN's are known as robust on additive noise, but this is not true for multiplicative noise: one needs to know the noise level applied on classification and how the

real correlation between data and label is, in order to proper set the model up [7]. Because of that, it is not expected for CILP++ to have good performance int the presence of this kind of noise. The noise analysis is presented on this work for the sake of completeness. As part of future work, we intend to proper evaluate CILP++'s noise robustness capabilities on continuous data, where additive noise is viable;

2. With regard to both alzheimers and mutagenesis benchmarks, a varied number of accuracy results using Aleph have been reported in the literature, such as [8, 20, 27]. As a result, we have decided to run our own comparisons between CILP++ and Aleph. We built 10 folds for all experiments and both systems shared it.

In the following, firstly an introductory description of both datasets and experiments will be made, then the obtained results regarding the three settings enumerated before will be presented.

D.1 Datasets Description

Most of used ILP datasets as testbeds for new models or improvements of existent ones belongs to the Bioinformatics area, including both mutagenesis and alzheimers. Both of them will be presented with more details in this section.

The first benchmark, mutagenesis, was created in order to let ILP systems become able to predict the mutagenicity of a set of 230 aromatic and heteroaromatic nitro compounds. The prediction of mutagenesis is important as it is relevant to the understanding and prediction of carcinogenesis. Not all of the 230 examples from the benchmark are used by ILP models when evaluating their performance [57]: from them, 188 examples can be fitted using linear regression and the remaining 42, cannot. We will follow their steps and use only the 188 linearly fittable examples.

The second one, the four alzheimers datasets, contains data that describe four drug properties that newly-made Alzheimer's disease treatment drugs should have:

- Inhibit amine re-uptake (*alz-amine* dataset);
- High acetocholinesterase inhibition (*alz-acetyl* dataset);
- Good reversal of scopolamine induced deficiency (*alz-scopolamine* dataset);
- Low toxicity (*alz-toxic* dataset).

When this problem became an ILP benchmark, there was no proved pattern that should be followed by those drugs manufacturers in order for the final product to obtain such properties and ILP was then tested on this benchmark to try to predict when a given drug configuration would inherit such properties.

Dataset	Positive Examples #	Negative Examples #	Predicates #	BCP Features # [1]
mutagenesis	125	63	34	1115
alz-amine	343	343	30	1090
alz-acetyl	618	618	30	1363
alz-memory	321	321	30	1052
alz-toxic	443	443	30	1319

Table D.1: Datasets Statistics

To conclude this overview, some general statistics regarding both dataset are given below.

D.2 Experiments Description

The main goal of these experiments is to demonstrate how CILP++ behaves on different situations, compared with an ILP benchmark. It will be evaluated on four different scenarios: with standard (original) dataset, with noise robustness, after doing feature selection and comparing it against other hybrid methods. For all processing speed results, complete runtimes is what will be considered (complete runtimes are referred as the sum of the time spent on all ten folds, including set-up, building the initial network, training and testing). About feature selection, this work will apply two methods of the ones described on Subsection A.3: mRMR and "logical" reduction/extension through changing the bottom clauses generation algorithm *depth* parameter, on algorithm C.1. Lastly, for other approaches comparison, one of the propositionalization methods presented in subsection A.4, RSD and BCP will be compared against each other, with their native learners and sharing a common learner as well, Quinlan's C4.5 decision tree.

In the first set of experiments, four CILP++ configurations will be tested:

- st: standard back-propagation stopping criteria;
- es: early stopping;
- $n\%bk$: the network is built with a subset of $n\%$ of the example set;
- $2h$: hidden layer with only 2 neurons.

The fourth configuration, $2h$, is explained with details in [18]: neural networks with only two neurons in the hidden layer generalizes binary problems approximately as well as more complex others. Furthermore, if a neural network has too many features to evaluate (if the input layer has too many neurons), it has already enough *degrees of freedom*, thus further increasings on it by adding hidden neurons would just increase the overfitting probability. Additionally, since bottom clauses are just "rough" representations of examples, overfitting can be particularly bad, since we just want it to model the general characteristics of each example and thus, a simpler model would be required [5].

[1] This value is the maximum number obtained when applying BCP on each one of the ten folds of the dataset

For all experiments with Aleph, the same configurations as [27] will be used for the alzheimers datasets (any other parameter not specified below was used with Aleph's default value):

- Variable depth: 3;
- Lower bound on the number of positive examples to be covered by an acceptable clause: 2;
- Lower bound on the minimum accuracy of an acceptable clause: 0.7;
- Lower bound on the minimum score of of an acceptable clause: 0.6;
- Upper bound on the number of literals in an acceptable clause: 5;
- Upper bound on the number of negative examples allowed to be covered by an acceptable clause: 300.

With regard to mutagenesis, based on [43], the following parameters have been set up (again, if a parameter is not listed down below, its default value on Aleph has been used):

- Upper bound on the number of literals in an acceptable clause: 4;
- Upper bound on the number of negative examples allowed to be covered by an acceptable clause: 100.

For CILP++, we used $depth = 3$ (see Algorithm C.1) for BCP and for backpropagation training, the following parameters have been used on *st* configurations:

- Learning rate: 0.1;
- Decay factor: 0.995;
- Momentum: 0.

Lastly, on all experiments with *es* configurations, those are the used parameters:

- Learning rate: 0.05;
- Decay factor: 0.999;
- Momentum: 0.1.
- Alpha (early stopping criteria): 0.01

D.3 Results

In this section, all results regarding learning relational data with CILP++ will be presented. They will be divided according to the kind of experiment done, as mentioned in the beginning of this chapter: **accuracy results**, **results with feature selection** and **results with other hybrid approaches**. Although it is not the focus of this work, results regarding **noisy data** will be presented as well.

D.3.1 Accuracy Results

In this experiment, CILP++ will be evaluated mainly in terms of accuracy vs. speed against Aleph. Four results tables will be presented: two with accuracy averages and standard deviations over 10-fold cross-validation for *mutagenesis* and each *alzheimer* dataset, for both *st* and *es* configurations, and two others with complete runtime results (by "complete runtime" we refer to the summed total of building, training and testing times, for both systems), for both configurations as well. In the accuracy table, values in bold are the highest ones obtained and the difference between them and the ones in italic are statistically significant by two-tailed, paired t-test. In the runtime table, only markings for highest values are done. All experiments were run on a 3.2 Ghz Intel Core i3-2100 with 4 GB RAM.

Table D.2: Accuracy/standard deviation results and runtimes for *st* configuration (in % for accuracy and in *hh:mm:ss* format for runtimes). We can see that Aleph and CILP++ are comparable in the *st,2.5%bk* configuration, having better accuracy on three out of five datasets while being faster on four of them. The *st,2h* configuration, on *alz-toxic*, obtained better accuracy than Aleph and was three times faster. This indicates that the standard configuration (the same neural network training settings of C-IL^{2}P) are adequate for relational learning, in both terms of accuracy and speed.

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

It can be concluded from here and taking in consideration both *st* and *es* configurations that there is a trade-off between speed and accuracy when using them: if it is interesting for a given classification problem to learn as quickly as it can, even if not learning as best as possible, the *es* configuration can be an option. If the opposite is what is of concern, the *st* configuration should be the opted one.

D.3.2 Results with Feature Selection

It was been discussed on Subsection A.3 that due to the extensive size of bottom clauses, feature selection techniques may obtain interesting results when applied after BCP.

Table D.3: Accuracy/standard deviation results and runtimes for *es* configuration (in % for accuracy and in *hh:mm:ss* format for runtimes). In the *es* configuration, CILP++ got even faster performance, with the drawback a considerable decrease on accuracy, with Aleph managing to outclass CILP++ on almost all datasets, only falling behind on mutagenesis. This may indicate that the early stopping main focus, which is overfitting avoidance, possibly is not useful for bottom clauses: it does not consider how training is performing and only looks at validation error. Due to that, a stopping criteria that sacrifices a portion of the training set to use as validation in order to avoid something that may not be happening possibly is not the best one for this kind of problem.

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

We also proposed two ways of doing it: by changing the variable depth (see Algorithm C.1) and with a statistical method, which will be mRMR.

For the first approach, we changed the the used *depth* value in the first set of results, which was 3, to 2 and to 5, to analyze how a reduction or an increase in the number of generated features could affect CILP++ performance. To do so, we have chosen two datasets: *alz-amine* and *alz-toxic*. We opted for those two because CILP++ performed just well on then, not outstandingly well (like on *mutagenesis*), neither considerably bad (like *alz-acetyl*). Additionally, we have chosen the best *st* configuration (which was $st, 2.5bk$) and the best *es* configuration (which was $es, 2h$).

The conclusion from here is that feature selection can be helpful for BCP, but after bottom clause generation, because variable depth change did not perform well in our experiments but on the other hand, when using mRMR, a trade-off of over 90% feature reduction in exchange of less than 3% accuracy could be seen by the results. This can be helpful if efficiency is required in an application for CILP++.

D.3.3 Comparative Results With RSD

In this section, comparative results against RSD will be done. We will only use the mutagenesis dataset to do the comparison due to a limitation on it: it is unable to deal with target predicates having arity higher than 1. Since all four alzheimers datasets have target predicates having arity 2, they could not be used. Below, the accuracy and runtimes

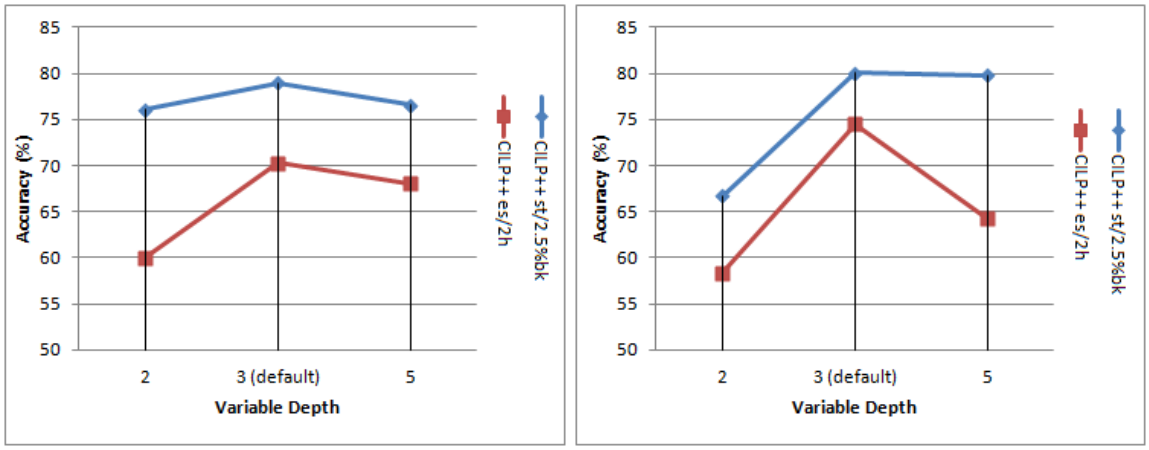

Figure D.1: Accuracies over variable depth changes on *alz-amine* (left) and *alz-toxic* (right). The results with different variable depths indicates that the default value we have chosen for variable depth is satisfactory: neither increasing or decreasing on its value helped on performance. As stated in Section A.1, variable depth controls how far the bottom clause generation algorithm will go when generating concept chaining and it is a way of controlling how much information loss the propositionalization method will have. From this and from the obtained results, it should be intuitive that higher variable depth should mean better performance, but together with useful features, it brings redundancy as well: not all generated chainings will be useful to describe an example, since the bottom clause generation algorithm picks up all possible ones during a cycle through the language bias.

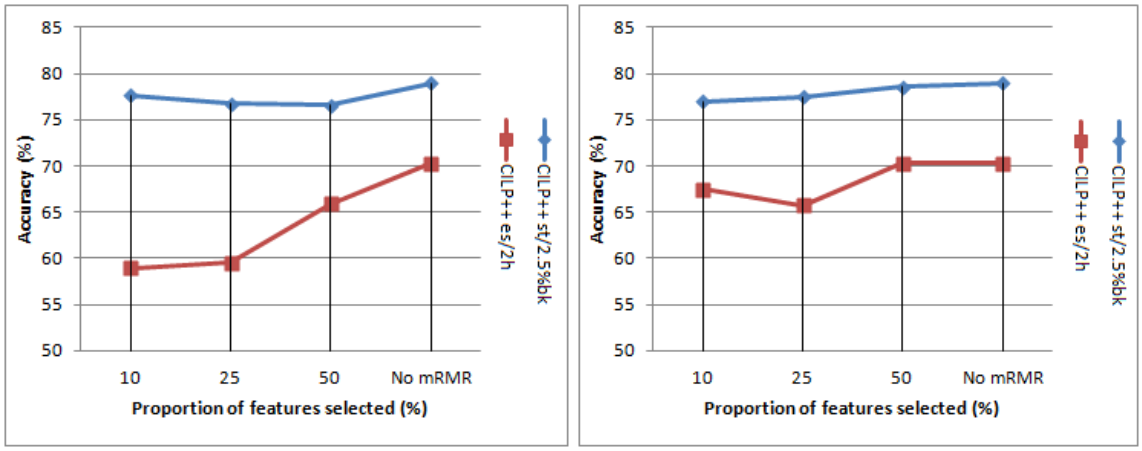

Figure D.2: Accuracies with mRMR on *alz-amine* (left) and *alz-toxic* (right). When applying mRMR, it can be seen that this method can be used to enhance efficiency: for the *alz-amine* dataset, for instance, a reduction of 90% of the number of features caused only a loss of less than 3% on accuracy with the *st,2.5%bk* configuration.

table is shown. We will compare both BCP and RSD propositionalization methods when generating training patterns for CILP++'s recursive ANN (labeled *ANN* in the table) and

C4.5 decision tree. Aleph results will be shown as well, to be used as baseline. We will use the CILP++ configuration that obtained the best results so far: *st,2.5%bk*.

Table D.4: Accuracy and runtime results for *mutagenesis* and *krk* datasets (in % for accuracy and in *hh:mm:ss* format for runtimes). The results show that CILP++ as a whole (*BCP+ANN* configuration) is better than RSD, while keeping highly competitive results with Aleph, but RSD performed as well as BCP when using C4.5 as learner. Regarding runtimes, CILP++ stayed on a par with Aleph, having got the upper hand in one dataset each, but it is faster than RSD. This confirms our expectations: we continue obtaining comparable accuracy results with Aleph in KRK, but faster results (mutagenesis is the only odd one out). But in the comparison with RSD, CILP++ outperformed it in all models. The results also show that BCP matches very well on both learners, but excels with our ANN. On the other hand, RSD did not fit well with ANN. Regarding runtime when comparing with RSD, we were faster.

Dataset	***Aleph***	***BCP+ANN***	***RSD+ANN***	***BCP+C4.5***	***RSD+C4.5***
muta	$80.85 * (\pm 10.51)$	**91.70**(± 5.84)	$67.63 * (\pm 16.44)$	$85.43 * (\pm 11.85)$	$87.77 * (\pm 1.02)$
krk	**99.6**(± 0.51)	$98.41 * (\pm 1.03)$	$72.38 * (\pm 12.94)$	$98.84 * (\pm 0.77)$	$96.1 * (\pm 0.11)$
muta	0:08:15	0:10:34	0:11:11	**0:02:01**	0:02:29
krk	0:11:03	0:02:37	0:06:21	**0:01:59**	0:05:54

Our goal in this scenario is to show that BCP, as a standalone propositionalization method, is fast and capable of generating accurate features for learning. Our results show that it excels when integrated with our ANN model, inside CILP++, but it performs on a par with a well-known propositionalization algorithm, RSD. Nevertheless, we show that it is also faster for all tested learners on all datasets. We kept Aleph among results to use as a baseline: we can see that no accuracy results with regarding to Aleph are statistically significant, which allows us to confirm that BCP with other learners is also comparable.

D.3.4 Results on Noisy Data

On this scenario, CILP++ will be evaluated in terms of robustness in the presence of multiplicative noise. For this, the two best configurations of CILP++, which is *st,2.5%bk* for the *st* results table and *es,2h*, from the *es* results, will be used again. Those configurations, will be ran on the same datasets of the feature selection results, *alz-amine* and *alz-toxic*. With this, we want to analyse if the slightly better results obtained will be improved or will decrease in the presence of multiplicative noise. It is known that neural networks deals better with additive noise [7], but it is not possible to do this kind of tweak on relational datasets while keeping the experiments fair on both Aleph's and CILP++'s sides.

In order to simulate multiplicative noise on those datasets, 10%, 20% and 30% of training data will have its label "flipped" (positive examples will become negative and vice-versa) and two results plots, one for each dataset, will be presented in the following, comparing those two configurations against Aleph and using 10-fold cross validation.

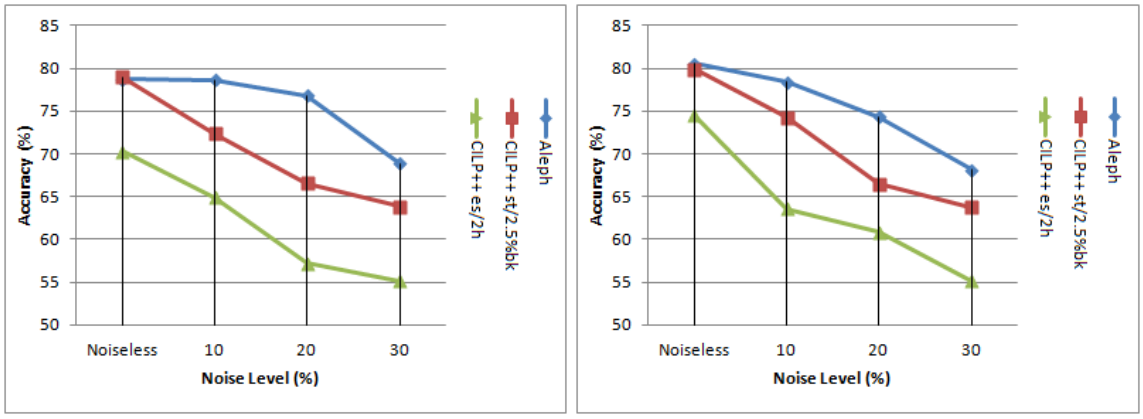

Figure D.3: Noisy Results on *alz-amine* (left) and *alz-toxic* (right). From the results, Aleph seems more robust to multiplicative noise. Since we used multiplicative noise on the data, this somewhat explains those results. It needs also to be taken into consideration that heuristic propositionalization methods are not expected to perform as well as ILP algorithms, since there is information loss involved in the process. Even so, when noise level starts to get higher (with 20% and 30% of noise level), CILP++'s *st,2.5%bk* model curve inclination starts to get small quicker than Aleph. This behavior can be seen with *es,2h* as well, but it starts to learn almost nothing on both datasets (its accuracy becomes close to 50%).

The conclusion from this analysis is that as it is currently, CILP++ is not robust to multiplicative noise. As future work, a deeper analysis of this behavior, with continuous datasets (where additive noise can be put into the features), should be done. An analysis of if a more robust propositional model can deal better than CILP++'s recurrent ANN or C4.5 with that kind of noise is part of future work as well.

D.4 Discussion

Summarizing, our experimental results consisted on four scenarios:

1. Accuracy vs. runtime;
2. Accuracy with feature selection;
3. Accuracy when compared with RSD;
4. Accuracy in the presence of multiplicative noise.

In the first analysis, we could see that BCP is an interesting approach for propositionalization, taking advantage of an ILP element which until now, has been used just as a hypothesis search boundary by Progol to generate a set of features of which propositional learners can take advantage of. We got comparable accuracy with a state-of-the-art ILP system, Aleph, performing better than it on three out of five datasets.

The second analysis, regarding feature selection, shown that there is an optimal variable depth parameter and"more is not merrier". With regard to mRMR, it obtained a really close accuracy if compared with not using it, even with a selection of only 10% of features (a reduction in the number of features of 90%). This indicates that this approach for reducting BCP's feature number can be promising.

With regard to the third approach, it has been shown that CILP++, as a whole, is superior in terms of accuracy when compared with RSD when using our ANN model, but if BCP and RSD are compared with each other with the C4.5 decision tree, they are on a par. But on both comparison, the approaches with BCP performed faster.

The last scenario took multiplicative noise into consideration. As stated before, ANN's are not necessarily robust to this kind of noise and we know that this experiment would be challenging. Even so, when noise became more severe, the *st,2.5%bk* configuration started to slow down its accuracy decreasing, if compared with Aleph. The *es,2h* model shown similar configuration, but it should not be taken into consideration because it was on the brink of learning nothing, since it was becoming closer and closer to 50% accuracy.

Taking all into consideration, CILP++ seems promising. As a first attempt of extending C-IL^{2}P to deal with relational data, it performed reasonably well when compared with Aleph, a state-of-the-art ILP system. Specifically regarding BCP, it is shown to be an easily configurable, practical and simple propositionalization method. It takes advantage of Progol-like language bias (composed by *mode declarations* and *determinations*) and generates bottom clauses which shares the same variable renaming and thus, it keeps consistency between then and their literals can directly be used as features for a propositional learner and obtaining good results with it is possible, as we have shown empirically. Still, there is a lot to be done as future work (they will be explained in the next chapter), which can increase even further this approach's capabilities.

Apêndice E

C–IL^{2}P Algorithms

Algorithm E.1 C–IL^{2}P's building algorithm

for all $c_i^B \in B$ **do**
  Add a neuron h to the hidden layer of N;
  Associate h with a label equal to the clause represented by c_i^B;
  Set the activation threshold of h as θ_l (see Table E.1);
  Split c_i^B into a set of body literals $L = \{b_1, \ldots, b_n\}$ and a head literal $head$;
  for all $b_i \in L$ **do**
    Add a neuron e to the input layer of N;
    Connect e to h, with a calculated weight W (see Table E.1);
    Associate e with a label equal to the atom represented by b_i;
    Set the activation function of e as linear [1];
    Set the activation threshold of e as 0.0;
  end for
  Add a neuron o to the output layer of N;
  Connect h to o, with a calculated weight W;
  Associate o with a label equal to the head literal represented by $head$;
  Set the activation threshold of o as θ_A (see Table E.1);
  Set the activation function of both h and o as semi–linear bipolar [2];
  if exists an input neuron e' with same label as o **then**
    Connect o to e', with a fixed weight 1.0;
  end if
end for

[1] Linear function: $f(x) = x$
[2] Semi–linear bipolar function: $f(x) = \frac{2}{1+e^{-\beta.x}} - 1$, where β is a slope parameter, usually set to 1.0

Some of the network parameters are calculated during its building stage. At Table E.1, each one of them will be briefly described. All the bounds and equations quoted in there are demonstrated in [16].

Parameter	Description
$(k_1, \ldots, k_i, \ldots, k_b)$	Number of body literals of each i background knowledge clause
$(\mu_1, \ldots, \mu_i, \ldots, \mu_b)$	Number of clauses in B with same head as each of the i background knowledge clauses
A_{min}	Primary C–IL^{2}P parameter. It must be below 1 and above a calculated lower bound[1]
W	Absolute starting value for all non–recursive weights, having a calculated lower bound[2]
θ_l	Threshold value for hidden neurons[3]
θ_A	Threshold value for output neurons[4]

Table E.1: Building stage calculated parameters

Algorithm E.2 C–IL^{2}P's input conversion function

1: **function** `convertInput(`$\mathtt{l}_+$`)`
2: Let i be an integer, starting with 1.
3: Let r be a vector, $r \in \Re^{n_i}$, with value -1.0 in all positions
4: **while** $i \leq n_i$ **do**{n_i is the size of the input layer}
5: **if** the label of the *i*th input neuron equals any literal in $\mathtt{l}_+$ **then**
6: Change the value of $r[i]$ to 1.0;
7: **end if**
8: Increase i by 1;
9: **end while**
10: **return** r;
11: **end function**

Table E.2 presents the used stopping criteria parameters used hereafter in Algorithm E.3, to give a better notion of what exactly they mean.

[1] $A_{min} > \frac{max_B(k_1,\ldots,k_b,\mu_1,\ldots,\mu_b)-1}{max_B(k_1,\ldots,k_b,\mu_1,\ldots,\mu_b)+1}$.
[2] $W \geq \frac{2}{\beta}.\frac{\ln(1+A_{min})-\ln(1-A_{min})}{max_B(k_1,\ldots,k_b,\mu_1,\ldots,\mu_b)(A_{min}-1)+A_{min}+1}$.
[3] $\theta_l = \frac{\ln(1+A_{min})(k_i-1)}{2}W$.
[4] $\theta_A = \frac{\ln(1+A_{min})(1-\mu_i)}{2}W$.
[1] "Rangeness" in the scope of C–IL^{2}P is the value of the standard error function $e(output, target) = \frac{|output-target|^2}{2}$.

Algorithm E.3 C–IL^{2}P's stop criteria check function

function `checkStopCriteria(`$index$`,` $stableEpochs$`,` $best$`)`
  Let $numberInMargin, numberOfHits$ be integers with starting value 0;
  if $numberOfEpochs \geq MAX$ **then**
    return $true$;
  end if
  for each $c_j^P \in P$ **do**
    Let $inputVector$ be the result of `convertInput(`l_+`)` for c_j^P [1];
    Apply *feed–forward* operation[2] on N with $inputVector$ as input;
    Let s be the network response vector to the *feed–forward* operation;
    Increment $numberInMargin$ and $numberOfHits$ by 1;
    for each $s_i \in s$ **do**
      if $e(s_i, targetOutput$[3]$) > correctnessRange$ **then**
        Decrement $numberInMargin$ by 1;
        break loop;
      end if
    end for
    for each $s_i \in s$ **do**
      if $e(s_i, targetOutput) > stabilizationRange$ **then**
        Decrement $numberOfHits$ by 1;
        break loop;
      end if
    end for
  end for
  if $\frac{numberInMargin}{p} \geq correctnessProportion$ **then**
    return $true$;
  end if
  if $\frac{numberOfHits}{p} \geq stabilizationProportion$ **then**
    if $\frac{numberOfHits}{p} \geq best$ **then**
      Assign $\frac{numberOfHits}{p}$ to $best$;
      Assign 0 to $stableEpochs$;
    else
      Increment $stableEpochs$ by 1;
    end if
    if $stableEpochs >= stabilizationMaxEpochs$ **then**
      return $true$;
    end if
  end if
  return $false$;
end function

[1] The same splitting and conversion for each example applied in Algorithm E.4 must be done
[2] See [32]
[3] This is the same target output calculated for each example in Algorithm E.4

Algorithm E.4 C–IL^{2}P's training algorithm

1: Let $isFinished$ be a boolean variable that indicates when training is finished;
2: Let $index$ counts the amount of epochs executed so far and start it with 0;
3: Let $stableEpochs$ represents the current number of consecutive epochs without improvements in $stabilizationProportion$ and start it with 0;
4: Let $bestPerformance$ represent the best epoch accuracy so far and start it with 0.0;
5: **while** $(isFinished \equiv false)$ **do**
6: Increment $numberOfEpochs$ by 1;
7: **for all** $c_i^P \in P$ **do** {P is the set of training examples}
8: Split c_i^P into two sets of body literals (l_+ of positives and l_- of negatives) and a head literal $head$;
9: Let $inputVector$ be the result of `convertInput(`l_+`)`[1];
10: **if** if $head$ is a positive literal **then** {c_i^P is a positive example}
11: Let $targetOutput$ be 1.0;
12: **else**
13: Let $targetOutput$ be -1.0;
14: **end if**
15: Apply *back–propagation* updating[2] on N using $inputVector$ as input and $targetOutput$ as target output;
16: **end for**
17: Assign to $isFinished$ the result of `checkStopCriterion(`$index$`,` $stableEpochs$`,` $bestPerformance$`)`[3];
18: **end while**

[1] See Algorithm E.2
[2] See [32]
[3] See Algorithm E.3

Algorithm E.5 C–IL^{2}P's testing algorithm

Let $numberOfHits$ counts the number of right answers among T and start it with 0;

Let $stabilized$ indicate when the activations are stabilized[1] and start it with $false$;
for all $c_i^T \in T$ **do** {T is the set of testing examples}
  Split c_i^T into two sets of body literals (l_+ of positives and l_- of negatives) and a head literal $head$;
  Let $testVector$ be the result of `convertInput(l+)`;
  Apply *feed–forward* operation on N with $testVector$ as input;
  Let s be the network response to the *feed–forward* operation;
  for all $s_i \in s$ **do**
    if neuron i has a recurrent connection to an input neuron n **then**
      Use it to make s_i be the next input of n;
    end if
  end for
  while $stabilized \equiv false$ **do**
    Apply *feed–forward* operation on N;
    Let s' be the network response to the *feed–forward* operation;
    Assign $true$ to $stabilized$;
    for all $s'_j \in s'$ **do**
      if (neuron j is connected to an input neuron n') **then**
        if `returnStatus(input, n')` $\neq$ `returnStatus(output, j)` **then**
          Assign $false$ to $stabilized$;
        end if
        Use it to make s'_j be the next input of n';
      end if
    end for
  end while
  if ($head$ is a positive literal) **and** (the output neuron with $head$ as label is $active$) **then**
    Increment $numberOfHits$ by 1;
  end if
  if ($head$ is a negative literal) **and** (the output neuron with $head$ as label is $inactive$) **then**
    Increment $numberOfHits$ by 1;
  end if
end for
return $\frac{numberOfHits}{t}$ [2] as testing accuracy;

[1] A network is considered to be *stable* if all output neurons activation states are identical to its correspondents in the input layer, if exists

[2] Recall that t is the size of the testing dataset, T

Parameter	Description
MAX (criterion 1)	Maximum number of epochs allowed
$correctnessRange$ (criterion 2)	Maximum value of which a N's output will be considered as "within range"[1] of a desired target
$correctnessProportion$ (criterion 2)	Minimum proportion of examples of P that, when used as a *feed–forward* input, needs be within *corectnessRange* in order to stop the training
$hitMargin$ (criterion 3)	Margin of a desired network answer that an output will be considered as correct
$stabilizationProportion$ (criterion 3)	Minimum proportion of examples of P that, when used as a *feed–forward* input, needs to be within *hitMargin* in order to the current epoch to be considered as having enough accuracy
$stabilizationMaxEpochs$ (criterion 3)	Maximum consecutive training epochs that will be executed without improvements in its proportion of correctness, given that their proportions are equal of above *stabilizationProportion*

Table E.2: Training stage stopping criteria parameters

Algorithm E.6 C–IL^{2}P's activation analysis function

```
1: function returnStatus(layer, index)
2:     Let activation be the activation of the index neuron of the layer layer;
3:     if activation ≥ Amin[1] then
4:         return active;
5:     else if activation ≤ −Amin then
6:         return inactive;
7:     else
8:         return unknown;
9:     end if
10: end function
```

[1]Recall that $Amin$ was one of the calculated parameters on C–IL^{2}P's building stage