Self-Organized Inductive Learning in a Multidimensional Graph-like Neural Network Framework

Ana Carolina Melik Schramm

Supervisor: Prof. Edjard Souza Mota, Ph.D.

Institute of Computing
Federal University of Amazonas

This Thesis is submitted for the degree of
Master in Informatics

Manaus-AM August 2022

Ana Carolina Melik Schramm

Self-Organized Inductive Learning in a Multidimensional Graph-like Neural Network Framework

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas, como requisito parcial para a obtenção do título de Mestre em Informática.

Master's thesis submitted to the Graduate Program in Informatics of the Computer Science Institute of the Federal University of Amazonas, as partial requirement for obtaining the title of Master in Informatics.

Supervisor: Prof. Edjard Souza Mota, Ph.D.

Manaus / AM
2022

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

S377s

Schramm, Ana Carolina Melik
   Self-organized inductive learning in a multidimensional graph-lke neural network framework / Ana Carolina Melik Schramm . 2022
   59 f.: il. color; 31 cm.

   Orientador: Edjard Souza Mota
   Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

   1. Artificial intelligence. 2. Neural-symbolic integration. 3. Inductive clausal learning. 4. Multidimensional graph neural network. I. Mota, Edjard Souza. II. Universidade Federal do Amazonas III. Título

UFAM

FOLHA DE APROVAÇÃO

"Self-Organized Inductive Learning in a Multidimensional Graph-like Neural Network Framework"

ANA CAROLINA MELIK SCHRAMM

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

[illegible]

Prof. Edjard de Souza Mota - PRESIDENTE

Paulo César da Rocha Fonseca

Prof. Paulo Cesar Fonseca - MEMBRO EXTERNO

[illegible]

Prof. Marco Antônio Pinheiro de Cristo - MEMBRO INTERNO

Manaus, 21 de Março de 2022

Acknowledgements

First and foremost, I would like to express my gratitude to my thesis advisor prof. Edjard Mota for all his help and patience, as well as to UFAM, IComp, Capes and FAPEAM for the opportunity.

I would also like to thank the committee, profs. Paulo Cesar Fonseca and Marco Cristo for their inputs, which helped me to improve this work.

Finally, thank you to my family for their love and support.

Resumo

A cognição humana depende fortemente do aprendizado indutivo, um processo que o campo do aprendizado de máquina visa replicar de maneira artificial em hardware/software. Embora os métodos de aprendizagem conexionistas tenham produzido grandes resultados pragmáticos nessa área, eles ainda carecem de um modelo de hierarquia de aprendizagem para explicar seus resultados. A computação Neural Symbolic (NeSy) busca desenvolver uma integração efetiva entre a aprendizagem conexionista e simbólica. Como esforço para alcançar essa integração, Neural Multi-Space (NeMuS) é uma estrutura gráfica multidimensional, originalmente concebida com quatro espaços de elementos codificados de lógica de primeira ordem, que aprende padrões de refutação e realiza raciocínio oracional indutivo para induzir hipóteses que explicam exemplos não especificados anteriormente em um conhecimento prévio. Recentemente, houve um experimento em que um conhecimento de fundo conectado foi treinado usando Self-Organizing Map (SOM) para gerar similaridade de conceitos de acordo com seus atributos e sua respectiva posição dentro dos conceitos. Neste experimento, a indução foi realizada por análise humana no mapa organizacional de conceitos. Neste trabalho, buscamos um método adequado para gerar padrões de vizinhança a serem usados para aprendizado indutivo e raciocínio a fim de reduzir o espaço de busca de hipóteses. Adicionalmente, definimos um viés de linguagem capaz de lidar com invenção de predicados, para orientar o processo de geração de tais hipóteses.

Palavras-chave: Neural-symbolic, Inductive Logic Programming .

Abstract

Human cognition heavily relies on inductive learning, a process that the field of machine learning aims to replicate in artificial hardware/software. While connectionist learning methods have yielded great pragmatic results in this area, they still lack a model hierarchy of learning to explain their results. NeSy computing seeks to develop effective integration between connectionist and symbolic learning. As an effort to achieve this integration, NeMuS is a multi-dimensional graph structure, originally conceived with four spaces of codified elements of first-order logic, that learns patterns of refutation and performs inductive clausal reasoning to induce hypotheses that explain examples non-previously specified in a background knowledge. Recently, there was an experiment in which a connected background knowledge was trained using SOM to generate similarity of concepts according to their attributes, and their respective position within the concepts. In this experiment, induction was performed by human analysis on the organizational map of concepts. In this work, we seek a suitable method to generate neighbourhood patterns to be used for inductive learning and reasoning in order to reduce the search space of hypotheses. Additionally, we define a language bias able to handle predicate invention, to guide the process of generating such hypotheses.
Keywords: Artificial Intelligence, Neural-Symbolic Integration, Inductive Clausal Learning, Multidimensional Graph Neural Network.

Table of contents

List of figures

List of tables

ML	Machine Learning
NeSy	Neural Symbolic
NeMuS	Neural Multi-Space
SOM	Self-Organizing Map
SOIL	Self-Organised Inductive Learning
ANNs	Artificial Neural Networks
ILP	Inductive Logic Programming
ICL	Inductive Clausal Learning
SOIR	Self-Organizing Inductive Reasoning
LP	Logic Programming
BK	Background Knowledge
AI	Artificial Intelligence
MIL	Meta-Interpretive Learning
CILP	Connectionist Inductive Logic Programming
ABL	Abductive Learning
NTP	Neural Theorem Prover
LRNN	Lifted Relational Neural Network
GNN	Graph Neural Network
CNN	Convolutional Neural Networks
GCN	Graph Convolutional Networks
GAE	Graph Autoencoders
SAE	Sparse Autoencoder
GraIL	Graph Inductive Learning
FOL	First-Order Logic
FOIL	First-Order Inductive Learning

Chapter 1

Introduction

One of the main ways by which humans learn is by observing instances of things in the real world and generalising for what is yet unseen, a process called inductive learning. One of the most important topics in machine learning is the study and modeling of inductive learning, a task considered as the "key to an improvement of methods by which computers can acquire logic"[4]. Even more recently, it has been said in [5] that the aim of Machine Learning (ML) is to automate induction. To this end, various models have been created, and while many are able to induce admirably, there is still a wide gap between them and the human brain. The reason for this gap is easily explained: a human only needs one interaction with, say, a poisonous plant, to know to avoid that same plant in the future. That is, human brains are able to generalise from very few examples, sometimes only one. Creating computational systems that generalise from few examples is still a great challenge in artificial intelligence.

Connectionist learning methods prevail currently in ML. Connectionism comes from the field of cognitive science, where it is defined as "an approach to the study of human cognition that utilizes mathematical models, known as connectionist networks or artificial neural networks" [6]. Artificial Neural Networks (ANNs) are composites of simple processing elements (artificial neurons) found in different layers, including one input, one output, and possibly multiple hidden layers[7]. They aim to create models that accurately represent the cognitive processes that constitute learning, and have repeatedly been proven to successfully model solutions to many real-world problems. However, unlike humans, they need many examples to be able to perform induction. Other limitations of ANNs include their black-box nature, tendency to overfit as their size and expressivity increase, and the idea of propositional fixation based on an argument of John McCarthy[8] (that neural networks are limited to propositional logic and cannot represent relational knowledge). Taking these limits into consideration has given rise to the idea that pure connectionism does not explain intellectual abilities precisely.

The symbolic approach to cognitive science precedes the connectionist one, and consists of rule-based models in which learning and concept representation are seen as symbol processing and manipulation. Although symbolic methods predominated in the early days of computer modeling of the mind and have been applied to areas such as problem solving and language processing, they were eventually abandoned due to their severe limitations. Currently, logic programming methods such as Inductive Logic Programming (ILP) have shown potential in addressing the issues with ANNs cited above, as they can be more easily understood by humans due to learning explicitly symbolic programs. However, they cannot be applied to the same variety of problems that ANNs can, and are not robust to noise and mislabelled data[9].

Human cognition, instead of being either purely connectionist or purely symbolic, seems to consist of processes belonging to both approaches. Moreover, connectionist and symbolic approaches both have limitations that are addressed by each other. Therefore, it stands to reason that an interesting and potentially more accurate model would be one that integrated connectionist learning with symbolic reasoning. This consitutes the field of NeSy computing, which this work fits into.

In [10], Diniz and Mota began the development of a platform to integrate connective and symbolic learning. The proposed Shared NeMuS learns patterns of complementary literals by training a knowledge base, in order to embody resolution within the process of learning and reasoning as matching of the coded query against regions of these patterns. The structure was then extended in [11] for symbolic Inductive Clausal Learning (ICL). This type of learning differs from classic ILP, as it is not limited to Horn clauses and does not check every possible option in the search space. Instead, it leans on the inductive momentum mechanism (which prunes away candidates to generalized body literals if they have been reached from the bindings of terms of the negative examples). Additionally, it uses linkage patterns, which are paths, within graphs, that connect constants and the predicates where they appear.

1.1 Problem Description

In [1] a first experiment, called Self-Organizing Inductive Reasoning (SOIR), was carried out by Barreto and Mota in which a connected background knowledge was trained using a SOM approach. Unlike [10], this SOM training experiment was not meant to generate patterns of refutation, but rather of similarity of concepts according to their attributes, and their respective position within the concepts.

The knowledge base used in the experiment[1] was composed of unit clauses with predicates of the form $father/2$ and $mother/2$, with the objective of learning the *ancestor* rule. The NeMuS constant space, as is called the space of individuals in the universe of discourse, was fed to the SOM training phase, yielding a map where circles represented instances of $father$ and $\times$'s represented instances of *mother*. After training, the SOM was used to "learn to induce" rules that defined the *ancestor* target predicate. Below are the first 12 unit clauses in the knowledge base, as seen in [1]:

$father(Jake, Bill)$	$father(Bill, Ted)$	$father(John, Harry)$
$father(Jake, John)$	$father(Bill, Megan)$	$father(John, Susan)$
$mother(Matilda, John)$	$mother(Alice, Ted)$	$mother(Mary, Harry)$
$mother(Matilda, Bill)$	$mother(Alice, Megan)$	$mother(Mary, Susan)$

Table 1.1: A background knolwedge to induce ancestor relation.

According to [1], "$ancestor(a,b)$ and $ancestor(c,d)$ are located in the mother and father regions respectively. Therefore, we can say that an ancestor can be a father or a mother. [Figure 1.1] shows the SOM after the induction $ancestor(X,Y)$ knowing $ancestor(Jake, John)$ and $ancestor(John, Harry)$. The triangle v_0 represents the induction vector of $ancestor(Jake, John)$ and v_1 represents the example $ancestor(John, Harry)$. **From the organization of the map, we see that both vectors are near father instances so we can assume that Jake is father of somebody that is ancestor of Harry**".

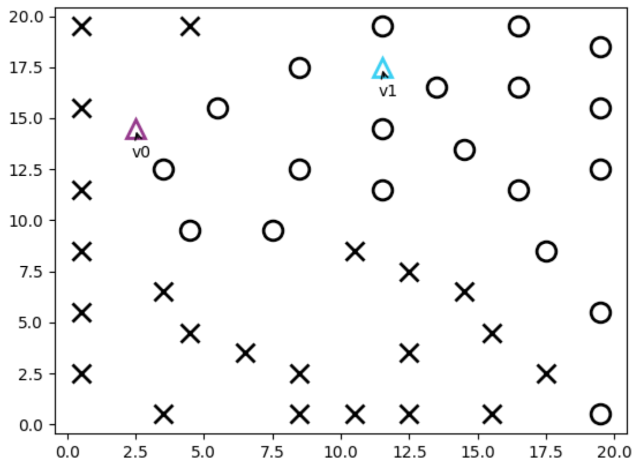


Figure 1.1: From [1], the SOM after inducing the *ancestor* rule only with positive examples

[1]Access to git repository can be provided upon request

The biggest limitation in the experiment was that the generation of the hypothesis that explains positive examples of the target concept, i.e. $ancestor(X,Y)$, had to be written out of the visual observation of the map. In other words, we had to observe the locations of positive examples, and then discover a rule based on the proximity of these examples to concepts on the map (as shown in the emphasized sentence at the end of the last paragraph). To automate this process it is necessary to endow our previous work, [12], with a mechanism able to learn in a self-organised inductive way. However, before that a far more difficult challenge needs to be tackled: to figure out a concept of *neighbourhood* to be used in finding *patterns* of groups of individuals (elements of constant space), that are connected to one another in the same way, via their attributes. The above concept, which we will call *neighbourhood pattern* is the central point of this work.

Furthermore, the learning of hypotheses in the way we envision requires the settings of the inductive process to be configured in order to reduce the search space of hypothesis and enable the development of heuristics to generate better hypotheses. The literature shows (e.g. Muggleton [13], Inoue[14]) that this can be done via the use of meta-level approaches, which (roughly) define hypotheses schemes in logic programming. In this work, we aim to achieve this through using language bias for the reduction of hypotheses space and perform some experiments to figure out an useful way to explore recursion patterns, if there are any.

1.2 Objetives of this Work

The objective of this work is to pinpoint a suitable method to generate *neighbourhood patterns* to be used for inductive learning and reasoning, based on the NeMuS neural-symbolic framework. As such, the idea is to maintain connectionist ML's efficient classification and learning from large-scale data, while bringing the advantageous aspects of symbolic ML to fill in the gaps in connectionism as is the main idea of NeSy Artificial Intelligence (AI).

We aim to do this by:

- Reducing the search space of hypotheses by making use of the "bridging concepts" mechanisms that are set by a proper language bias;
- Integrating ICL with language bias and inductive momentum, from [11], into this new framework;
- Providing predicate invention based on bridging concepts, as presented in [12];
- Performing experiments to figure out organization patterns of recursion by means of analysis of neighbourhoods of concepts.

Chapter 2

Related Works

2.1 Inductive Logic Programming

ILP was first proposed by Muggleton in [15], shortly after the introduction of relevant concepts to the area such as Inverse Resolution and Predicate Invention[16]. As a research field, ILP aimed to combine the logical knowledge representation from Logic Programming (LP) with ML. According to Muggleton in [17], as the connectionist inductive learning-based systems at the time had trouble dealing with propositional fixation, the core task in ILP was (and has remained) to learn from datasets in which not the values in themselves, but the relationships between individuals, are what provide the generalizations to learn from. As well as the problems with dealing with relational knowledge, ILP also tackles other limitations of connectionist systems at the time: logic programs are more expressive than the network and graph-structures predominantly used at the time, and are more easily interpretable by humans.

Today ILP is a form of ML that learns relations instead of functions, and in which data is represented using logic programs instead of vectors[18]. An ILP task is composed of three sets of logic programs: Background Knowledge (BK), positive examples (E^+) and negative examples(E^-). Given these sets, an ILP system is supposed to learn a hypothesis that entails all or as many positive examples as possible, and none or as few negative examples as possible.

There are generally two approaches to ILP systems: bottom-up and top-down. Bottom-up ILP starts by examining the examples provided and creating very specific hypotheses, which are then generalised. In contrast, top-down ILP uses a method called generate-and-test, in which hypotheses are generated first and then tested against the positive and negative examples provided. These hypotheses are as general as possible, and are further specified by, when a contradiction is found during testing, applying substitution or adding a literal to one of its clauses, or adding a new clause. One example of this approach being used is Metagol, a state-of-the-art ILP system.

2.1.1 Meta-Interpretive Learning and Metagol

As ILP learns from examples, the information contained in them is often not enough to generalize rules. When given an extensive or complex BK, as well as few examples, the resulting hypothesis space (that is, the set of all hypotheses a system can return) may be too large and inconclusive without the use of bias. In order to control the hypothesis space, ILP uses inductive bias, also known as priors. These are the set of assumptions a learner uses to compute predictions beyond what is in the training set. It is a key characteristic of ILP to use background knowledge as a way to impose strong inductive bias onto systems, which allows them to generalize from few examples, often a single one. Additionally, other forms of bias can be used, such as metarules. Used in a form of ILP known as Meta-Interpretive Learning (MIL), metarules are a form of declarative bias that consists in second-order Horn clauses with existentially quantified predicate variables. They are often included in the BK and help define the structure of the hypotheses to be learned.

Metagol [19] is a state-of-the-art ILP system based on Prolog that requires as input, as well as the logic programs previously mentioned, metarules that define the form of clauses permitted in a hypothesis. These are of the form $metarule(Name, Subs, Head, Body)$, where the symbols in *Subs* are the second-order variables, and the fields $Head$ and $Body$ contain, respectively, the head and the body of the metarule in the form [X, Y_1, ..., Y_n], where X is the second-order variable and all Y_is are the first-order variables that compose it. One example of this is the chain metarule $(P(A,B) \leftarrow Q(A,C), R(C,B))$, that in Metagol is written as $metarule(chain, [P,Q,R], [P,A,B], [[Q,A,C], [R,C,B]])$.

With this input, Metagol uses iterative deepening on the number of clauses in the solution to find adequate substitutions for the second-order variables in order to find a hypothesis. Following the proof of a set of goals, Metagol forms a logic program by projecting the meta-substitutions onto their corresponding metarules[20]. Metagol is also notable for its support of automatic predicate invention and its ability to learn recursive programs. It is, however, limited in the sense that it cannot handle noise.

2.2 Neural-Symbolic Computing based Inductive Learning

NeSy computing seeks to develop effective integration between connectionist learning and symbolic reasoning, possibly taking advantage of all statistical methods that can be applied to the features of data perceived or on the logical structure of symbolic information[21]. A system based on this integration could then transfer knowledge across domains and use background constraints to identify what is relevant in a particular context and so, restructure the problem (possibly inventing concepts) to make it easier to solve.

2.2.1 CILP

Garcez and Zaverucha propose in [22] Connectionist Inductive Logic Programming (CILP), which maps logic programs into a single layer artificial neural network. Each rule has a neuron in this layer, input signals to each hidden neuron represent propositional literals in the body and their heads are the output signals. Although they have found a very efficient NeSy solution, even proving that after training the equivalent ANN its corresponding logic program is correct, it is limited to propositional logic. Later, Garcez and others [23] proved that CILP and its extension to Learning by induction (C-IL^{2}P), could deal with possible world semantics which is a sort of non-propositional logic.

In [24], França et al. propose CILP++, an extension of CILP that allows for efficient and accurate extraction of first-order logic rules from trained networks. The learning process begins by generating bottom clauses (described as "the most specific clause that can be considered a candidate hypothesis", these clauses serve as boundaries in the hypothesis search space) for each example provided, mapping them onto features on an attribute-value table and generating numerical vectors that can be taken as input by a neural network. For learning, CILP++ uses resilient backpropagation with early stopping to train the network, and then applies an adapted version of the TREPAN rule extractor algorithm (adaptations include allowing the generation and query of first-order rules into Prolog, as well as simplifications to improve efficiency and readability) to perform relational knowledge extraction.

Recently, Garcez et al. [25] reported the results from the NeSy Computing community that has sought to integrate the views from AI, cognitive sciences, machine learning, ANN, computational vision and natural language processing, and point out the main lines NeSy should go to meet human-like computing. This is also know as Human-Like AI initiative.

2.2.2 ∂ILP

Differential ILP (∂ILP) assumes a that a logic program R is composed by the union of the BK and positive examples logic programs, a target predicate T, is the *intensional predicate* that must be learned, P_e, the set of *extensional predicates* in R, P_a is a set of auxiliary intensional predicates, the additional invented predicates used to help define the target predicate, and P_i = $P_a \cup T$. Then, a language template τ is of the form

$$\tau = \begin{cases} P_e \\ P_i \end{cases}$$

Evans and Grefenstette in [26] propose ∂ILP, a reimplementation of ILP with a differentiable implementation of deduction through forward chaining on definite clauses. ∂ILP's approach to ILP is top-down: it generates clauses from a language template and tests the generated programs against the positive and negative examples. To make it a satisfiability problem, it assigns Boolean flags to each clause indicating whether it is on or off. Then, to solve this satisfiability problem, it uses a SAT solver to find a truth-assignment to the generated clauses, such that the clauses that are "on" together with the BK entails all the positive examples and none of the negative ones.

In order to keep the search space of clauses manageable, it is necessary to impose certain limitations. To do so, ∂ILP uses concepts such as rule templates.

A rule template describes the range of clauses that can be generated. Specifically, the number of variables allowed and whether the atoms in the clause can use intensional predicates or only extensional ones.

∂ILP limits clause generation by establishing that each intensional predicate p is defined by exactly two clauses specified by two rule templates. Other limitations include not allowing the following:

- Constants in any of the clauses;
- Clauses with predicates of arity above 2;
- Circular clauses;
- Duplicated clauses;
- Clauses in which a variably present in the head is not in the body;
- Clauses that contain intensional predicates when their rule template does not allow them.

In turning an induction problem into a satisfiability one, ∂ILP makes it possible to minimize loss by applying stochastic gradient descent and learn the correct truth-assignment to the clauses generated, even if there is noise or if the data is ambiguous. Furthermore, ∂ILP uses continuous weights to determine a probability distribution over clauses, instead of Boolean flags to choose a subset of clauses. Thus, ∂ILP "implements differentiable deduction over continuous values". For this reason, it can handle ambiguous or mislabelled data, a common limitation in ILP. However, it still includes limitations such as its requirement of significant memory resources, and the necessity of use of templates to limit search space.

2.2.3 Other Approaches to NeSy

Dai et al. in [27] present a Abductive Learning (ABL), a NeSy approach using abductive learning as a way to unify machine learning and logical reasoning in a way that benefits both techniques. In it, a ML model interprets data into primitive logical facts, while a logical model reasons about missing or incorrect information by using background knowledge to obtain a consistent final output.

Rocktäschel and Riedel present in [28] the construction of a Neural Theorem Prover (NTP) based on differentiable backward chaining and unification, that can perform first-order inference in vector space like a discrete theorem prover would do on symbolic representations and can also learn representation of symbols and first-order rules of predefined structure. The key idea is to recursively construct neural networks by replacing operations on symbols in backward chaining with differentiable operations on distributed representations. To do that, they separate goals and substitutions into vector representations of involved predicates and constants, and structures that define the connections of a neural network.

Šourek et al. [29, 30] and Železný et al. [31] present Lifted Relational Neural Network (LRNN), which uses weighted relations rules for learning feed-forwards neural networks. They're different from standard neural networks because their structure is derived from symbolic rules and thus has an intuitive interpretation, and because the weights of the network are tied to the first-order rules and are thus shared among different neurons.

LRNNs can be used to learn a latent category structure that is predictive in the sense that the properties of an entity can be largely determined by the category to which that entity belongs, and the entities satisfying a property can be largely determined by the category to which that property belongs. This enables reasoning based on the idea that similar entities have similar properties.

2.3 GNN-based Inductive Learning

The Graph Neural Network (GNN) model was introduced by Scarselli et al. in [32] to accurately represent graph structures, in order to better deal with real-world applications in which data is represented in this form.

Scarselli et al. describe a graph G as a pair (N, E) where N is the set of G's nodes, while E is the set of G's edges. Each node denotes a concept, which is defined by its own features and related concepts. Relationships between concepts are denoted by the edges between the nodes, and each edge e and node n has its own label l, represented by a vector that includes its features. Therefore, each node n has a state x_n that depends on n's neighbourhood and features, and represents the concept associated with it. More formally, if f_w is a parametric function that expresses the dependence of a node n on its neighbors, then:

$x_n = f_w(l_n, l_co[n], x_ne[n], l_ne[n])$

where $co[n]$ is the set of arcs that have n as a vertex, and $ne[n]$ is the set of n's neighbors.

Tasks for learning on graphs can be node-focused or graph-focused. In graph-focused tasks, the learning doesn't depend on any particular nodes, but is associated with the entire graph, while node-focused tasks are associated with individual nodes and the learning depends on each of their properties. To deal with graph-focused tasks, [32] includes a "supersource node" that represents the whole graph and from which it is possible to reach every node.

With the rise of deep learning, applying it to graphs was shown to be a difficult task due to their irregular structures, heterogeneity and large scale in real applications. The recent advances in Convolutional Neural Networks (CNN) brought GNNs back into view, and they have thus been improved upon. Many other deep learning methods on graphs have been created, enough for there to be multiple up-to-date surveys on the subject. Zhang et al. [33] divide these methods in five categories: graph recurrent neural networks (the classic GNN as well as its improvements), Graph Convolutional Networks (GCN), Graph Autoencoders (GAE), graph reinforcement learning, and graph adversarial methods. For the scope of this work, only GCNs and GAEs will be touched upon, as these are the categories of methods that can be applied to the inductive learning setting.

2.3.1 Graph Convolutional Networks (GCNs)

The task of defining convolution and pooling operations on graphs, known as the geometric deep learning problem, is not as straightforward as on images or texts, because graphs don not have a grid-like structure. One main direction of research on this topic is, according to [34], to "define graph convolutions from the spectral perspective". Spectral methods perform convolutions by transforming node representations into the spectral domain. In graphs, this means using the graph Laplacian to find the combination of orthogonal elements that compose the graph - a task known as eigen-decomposition. Like in conventional convolutions, spectral convolution in graphs consists of passing input signals (the set of node features is the input layer) through a set of learnable filters to aggregate the information, and then doing some nonlinear transformation. In practice, this leads to limitations such as the need to learn O(N) parameters, possibility of filters not being localized in the spacial domain, high time complexity and inability to share parameters between graphs with different sizes and structures. All these problems are addressed by different methods. While most of these problems are addressed and alleviated in methods that still mostly use spectral convolutions, the multiple graphs aspect requires methods based in spatial convolutions - those that perform convolution by considering node neighbourhoods

Applying GCNs to inductive learning has been verified to be possible in some models like GraphSAGE[35], GAT[36], GaAN[37] and FastGCN[38], but only for graphs with explicit features. The task of performing inductive learning for graphs without those is known as the out-of-sample problem[39].

2.3.2 Graph Autencoders (GAEs)

Sparse Autoencoder (SAE) began the use of autoencoders for graphs, based on the idea that autoencoders can be used as dimensionality reduction techniques to learn low-dimensional node representations, if one regards the adjacency matrix (or its variations) of a graph as raw features of nodes. Despite outperforming non-deep learning baselines at the time, SAE was proven by Zhang [40] to be based on an incorrect theoretical analysis, and the reason for its effectiveness is unknown. However, the need to fill in the puzzle and further develop this line of research motivated the creation of other GAE methods, including ones that apply variational autoencoders (that combine dimensionality reduction with generative models) to graph data.

GAEs have been shown to be applicable to inductive learning by using a GCN as the encoder or by directly learning a mapping function from node features.

2.3.3 GNN Example

Example 2.1

Teru et al. [2] present Graph Inductive Learning (GraIL), a GNN-based relation prediction framework that learns entity-independent relational semantics with strong inductive bias. The paper gives an example of how learning methods that operate on embeddings (latent representation of entities and relations) work to predict missing edges on graphs. The training graph (Figure 2.1) and graph illustrating inductive inference (Figure 2.2) used in the paper, can be translated into clauses in the following way:

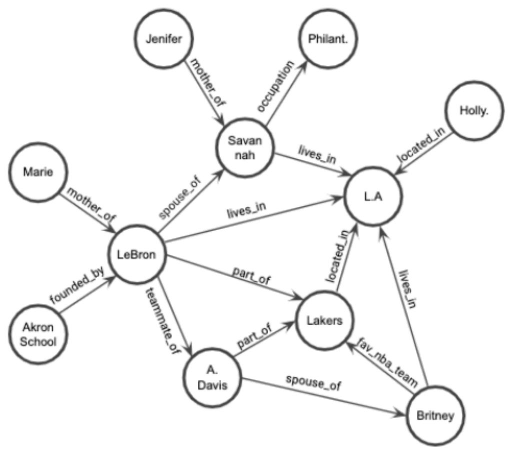


Figure 2.1: Training graph for example in [2]

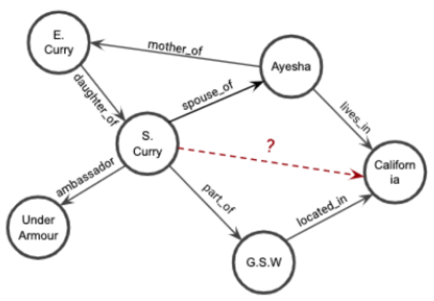


Figure 2.2: Graph illustrating inductive inference in [2]

From Fig. 2.1	From Fig. 2.2
$mother\_of(marie, lebron).$	$mother\_of(ayesha, e\_curry).$
$mother\_of(jenifer, savannah).$	$spouse\_of(s\_curry, ayesha).$
$spouse\_of(lebron, savannah).$	$ambassador(s\_curry, underarmour).$
$spouse\_of(a\_davis, britney).$	$lives\_in(ayesha, california).$
$lives\_in(lebron, la).$	$part\_of(s\_curry, gsw).$
$lives\_in(savannah, la).$	$located\_in(gsw, california).$
$lives\_in(britney, la).$	
$part\_of(lebron, lakers).$	
$part\_of(a\_davis, lakers).$	
$located\_in(lakers, la).$	
$located\_in(holly, la).$	
$occupation(savannah, philant).$	
$founded\_by(akron\_school, lebron).$	
$teammate\_of(lebron, adavis).$	
$fav\_nba\_team(britney, lakers).$	

In the example, the inductive inference is to learn the missing relation between s_curry and $california$, which would be $lives\_in(s\_curry, california)$ by taking into account the clauses $spouse\_of(lebron, savannah)$, $lives\_in(lebron, la)$ and $lives\_in(savannah, la)$. The rule learned in this case would be the following:

$$lives\_in(X,Y) \leftarrow spouse\_of(X,Z) \wedge lives\_in(Z,Y)$$

All approaches briefly described above seek to generate rules like the one presented in the example, which is the ultimate goal of a relational machine learning system. None of the works researched seem to solve the problem we are tackling: to perform induction without the need to walk the entire path that leads to, for example, a recursion. This is due, firstly, to the fact that none of the observed works have tried to perform inductive reasoning based on self-orgazing maps of concepts. Secondly, because none of them have logic-like structure interconnecting spaces of concepts. However, the limitations of the first versions of NeMuS structure need to be overcome by an extension on its space boundaries, and a more concise definition of its core component or building block needs to be provided. Thus, we end this chapter by addresing the rationale for doing this.

2.4 FOIL

Proposed by J.R. Quinlan in [3], First-Order Inductive Learning (FOIL) is a system where objects are described using relations. Given examples (positive and negative) of a target relation, FOIL is tasked with finding a rule that defines this relation.

FOIL accepts Horn clauses of the form $C \leftarrow L_1, L_2, ..., L_n$, where C is a predicate and each L_i is either a predicate or the negation of a predicate. Variables start with capital letters, and all other atoms are constants.

The FOIL algorithm consists of one outer operation and one inner operation. They work as follows:

Outer operation:

- 1) Establish the training set consisting of constant tuples, labelled `(+)` or `(-)`;
- 2) Until there are no `(+)` tuples left in the training set:
  - 2.1) Find a clause that characterizes part of the target relation;
  - 2.2) Remove from the training set all tuples that satisfy the right-hand side of this clause.

Inner operation:

- 3) Initialize a training set T_i, local to this operation. Let $i = 1$;
- 4) Until there are no `(-)` tuples left in T_i:
  - 4.1) Find a literal L_i to add to the right-hand side of the clause

- 4.2) Produce a new training set T_{i+1} based on the tuples in T_i that satisfy L_i. If L_i introduces new variables, each tuple from T_i can form several expanded tuples in L_{i+1};
- 4.3) Increment i and continue.

To visualize how this algorithm works, consider the following example:

Example 2.2

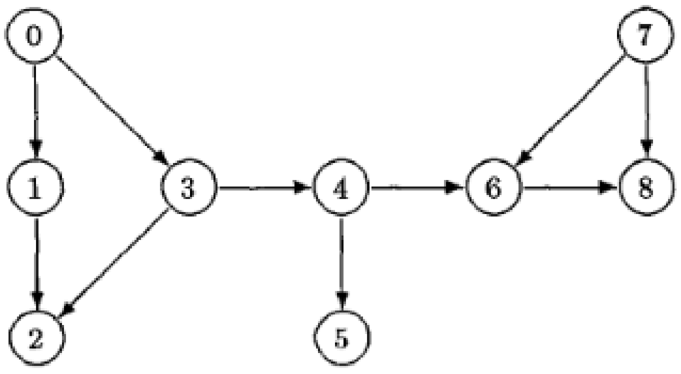


Figure 2.3: Graph that illustrates this example, found in [3]

Consider the target relation can-reach(X,Y), and the following BK, illustrated in Figure 2.3:

```
linked-to(0,1).  linked-to(4,5).
linked-to(0,3).  linked-to(4,6).
linked-to(1,2).  linked-to(6,8).
linked-to(3,2).  linked-to(7,6).
linked-to(3,4).  linked-to(7,8).
```

Note that in the training sets that follow, `<X,Y>` = `linked-to(X,Y)`.

- 1) Establish the training set:

  ```
  (+): <0,1> <0,2> <0,3> <0,4> <0,5> <0,6> <0,8> <1,2> <3,2>
  <3,4> <3,5> <3,6> <3,8> <4,5> <4,6> <4,8> <6,8>
  (-): <0,0> <0,7> <1,0> <1,1> <1,3> <1,4> <1,5> <1,6> <1,7>
  <1,8> <2,0> <2,1> <2,2> <2,3> <2,4> <2,5> <2,6> <2,7> <2,8>
  <3,0> <3,1> <3,3> <3,7> <4,0> <4,1> <4,2> <4,3> <4,4> <4,7>
  <5,0> <5,1> <5,2> <5,3> <5,4> <5,5> <5,6> <5,7> <5,8> <6,0>
  ```

<6,1> <6,2> <6,3> <6,4> <6,5> <6,6> <6,7> <7,0> <7,1> <7,2> <7,3> <7,4> <7,5> <7,7> <8,0> <8,1> <8,2> <8,3> <8,4> <8,5> <8,6> <8,7> <8,8>

- As there are (+) tuples in the training set, proceed to step 2.1;
- 2) As there are (+) tuples in the training set, proceed to step 2.1;
- 2.1) Ideally, the clause set in this step will satisfy many (+) tuples and few (-) ones. Here, it is set as `can-reach(X,Y)` $\leftarrow$ `linked-to(X,Y)`. This clause satisfies the (+) tuples <0,1>, <0,3>, <1,2>, <3,2>, <3,4>, <4,5>, <4,6>, <6,8>, <7,6> and <7,8>, while also not satisfying any of the (-) ones;
- 2.2) Remove the clauses mentioned in the previous step from the training set. This leaves it as:

  (+): <0,1> <0,2> <0,3> <0,4> <0,5> <0,6> <0,8> <1,2> <3,2> <3,4> <3,5> <3,6> <3,8> <4,5> <4,6> <4,8> <6,8>

  (-): <0,0> <0,7> <1,0> <1,1> <1,3> <1,4> <1,5> <1,6> <1,7> <1,8> <2,0> <2,1> <2,2> <2,3> <2,4> <2,5> <2,6> <2,7> <2,8> <3,0> <3,1> <3,3> <3,7> <4,0> <4,1> <4,2> <4,3> <4,4> <4,7> <5,0> <5,1> <5,2> <5,3> <5,4> <5,5> <5,6> <5,7> <5,8> <6,0> <6,1> <6,2> <6,3> <6,4> <6,5> <6,6> <6,7> <7,0> <7,1> <7,2> <7,3> <7,4> <7,5> <7,7> <8,0> <8,1> <8,2> <8,3> <8,4> <8,5> <8,6> <8,7> <8,8>

  Note that the clause generated in this outer operation is complete (since it satisfies (+) tuples and does not satisfy any (-) tuples). Therefore, the clause set by the inner operation is a different one, and not a continuation of `can-reach(X,Y)` $\leftarrow$ `linked-to(X,Y)`. On entering the inner loop, the right-hand side of the clause is empty;
- 3) Enter the inner loop with the training set from the previous step as T_1;
- 4) As there are (-) tuples in T_1, proceed to step 4.1;
  - 4.1) Let the literal added to the right-hand side of the clause be `linked-to(X,Z)`. None of the pairs <2,...>, <5,...> or <8,...> will satisfy this new literal, because in the original set of relations there are no `linked-to` relations with 2, 5 or 8 as `X`;

- 4.2) Since `linked-to(X,Z)` adds a new literal to the base, the tuples in T_2 will be triples instead of pairs. Each pair `<A,B>` will generate one triple `<A,B,Ci>` for each pair `<A,Ci>` in the relation linked-to. For example, `<0,2>` gives way to `<0,2,1>` and `<0,2,3>`, because of the relations `linked-to(0,1)` and `linked-to(0,3)`. T_2 looks like:
```
(+): <0,2,1> <0,2,3> <0,4,1> <0,4,3> <0,5,1> <0,5,3>
<0,6,1> <0,6,3> <0,8,1> <0,8,3> <3,5,2> <3,5,4> <3,6,2>
<3,6,4> <3,8,2> <3,8,4> <4,8,5> <4,8,6>
(-): <0,0,1> <0,0,3> <0,7,1> <0,7,3> <1,0,2> <1,1,2>
<1,3,2> <1,4,2> <1,5,2> <1,6,2> <1,7,2> <1,8,2> <3,0,2>
<3,0,4> <3,1,2> <3,1,4> <3,3,2> <3,3,4> <3,7,2> <3,7,4>
<4,0,5> <4,0,6> <4,1,5> <4,1,6> <4,2,5> <4,2,6> <4,3,5>
<4,3,6> <4,7,5> <4,7,6> <6,0,8> <6,1,8> <6,2,8> <6,3,8>
<6,4,8> <6,5,8> <6,6,8> <6,7,8> <7,0,6> <7,0,8> <7,1,6>
<7,1,8> <7,2,6> <7,2,8> <7,3,6> <7,3,8> <7,4,6> <7,4,8>
<7,5,6> <7,5,8> <7,7,6> <7,7,8>
```
- 4.3) Set $i = 2$ and go back to the beginning of step 4;

• 4) As there are still `(-)` tuples in T_2, proceed to step 4.1;

- 4.1) Let the new literal in the clause be `can-reach(Z,Y)`;
- 4.2) As no new variables were added, the tuples in the next training set will remain as triples. T_3 consists of the tuples in T_2 that are satisfied by the literal added in the previous step. This leaves T_3 as:
```
(+): <0,2,1> <0,2,3> <0,4,3> <0,5,3> <0,6,3> <0,8,3>
<3,5,4> <3,6,4> <3,8,4> <4,8,6>
```

As there are no `(-)` tuples in T_3, the inner loop will not be repeated. The clause generated is `can-reach(X,Y)` $\leftarrow$ `linked-to(X,Z), can-reach(Z,Y)`, and all `(+)` tuples in the original training set are reached by either this clause or the one generated in the outer loop. Therefore, the definition of can-reach is complete.

In the last iteration of step 4.1, choosing the literal `linked-to(Z,Y)` would also lead to a training set with no `(-)` tuples, which would lead to the clause `can-reach(X,Y)` $\leftarrow$ `linked-to(X,Z), linked-to(Z,Y)`. However, FOIL's heuristic for evaluating literals (explained in section 2.4.1) led it to choose `can-reach(Z,Y)`, as this literal was considered more useful.

Limitations of FOIL include being restricted to Horn clauses and not being capable of predicate invention. Additionally, it is based on a greedy algorithm for assembling clauses from literals. When valuating a possible literal, FOIL searches almost exhaustively through possible combinations of variables. Furthermore, once a literal is selected, no other alternatives are explored.

2.4.1 FOIL's heuristic for choosing literals

In FOIL, the utility of a literal L_i is measured in terms of information gained by adding it to the clause. Since clauses characterize a subset of the `(+)` tuples in a relation, this information gained is in regard to the `(+)` tuples in the training set T_i obtained by adding L_i to the clause. The information given by T_i labelling any of its tuples as $(+)$ is given by the following formula, supposing that T_i^+ is the number of `(+)` tuples in T_i, and that T_i^- is the number of `(-)` ones:

I(T$_i$) $= -log_2(T_i^+/(T_i^+ - T_i^-))$

Now, suppose that T_i^{++} of the tuples in T_i^+ generate one or more tuples in T_{i+1}. Then:

Gain(L$_i$) $= T_i^{++} \times (I(T_i) - I(T_{i+1}))$

To understand this, consider the previous example. In T_2, T_2^+ = 18 and T_2^- = 54. Applying these values to the first formula, $I(T_2)$ = 2. T_3 is then obtained by adding the literal `can-reach(Z,Y)`. 10 `(+)` tuples from T_2 are represented in T_3,each giving way to one tuple, as no new variables were added. So, T_2^{++} =T_3^+ = 10. As there are no `(-)` tuples in T_3, T_3^- = 0. Consequently, $I(T_3)$ = 0, which leads to:

Gain(`can-reach(Z,Y)`) = 10 $\times(2-0) = 20$

Had `linked-to(Z,Y)` been chosen, $I(T_3)$ would still be 0, but T_3 would have only 6 tuples (`<0,2,1>`, `<0,2,3>`, `<0,4,3>`, `<3,5,4>`, `<3,6,4>` and `<4,8,6>`), so T_2^{++} =T_3^+ = 6, and:

Gain(`can-reach(Z,Y)`) = 6 $\times(2-0) = 12$

This means that more information was gained by adding `can-reach(Z,Y)` to the base than by adding `linked-to(Z,Y)`. Hence, the former literal being added to the clause, rather than the latter.

2.5 Leveraging Information from Logical Ground Terms

Other symbolic or neural-symbolic logic-based inductive learning frameworks consider constants only for unification purposes, but do not pay attention to their degree of importance for logical inference. From the human learning perspective, it is intuitive to generate all possible hypotheses and then keep only those that succeed in proving positive examples and rejecting negative ones. However, while generating hypotheses, one can also "look ahead" to see whether a given example has chances of either failing or succeeding with the hypothesis in question. This is only possible if one understands how the positional bindings of objects within literals may offer information to foresee such a failure or success.

With this in mind, and aiming to consider the degree of importance of each First-Order Logic (FOL) object, Mota and Diniz, [10], proposed a Neural Multi-Space (NeMuS) representation with four spaces of codified elements of first-order logic. Variables and constants are paired in the first space, then spaces for functions, predicates and clauses. From the space of variables and constants, each element has a vector of bindings to upward elements with a weight of importance were they appear as attributes. The clause space has no bindings upward, and so might be used as outputs. This structure is illustrated in Figure 2.4(a) and Figure 2.4(b). The upward arrows in Figure 2.4(b) represent weights of influence from lower elements upon higher ones in which they appear as a composite part.

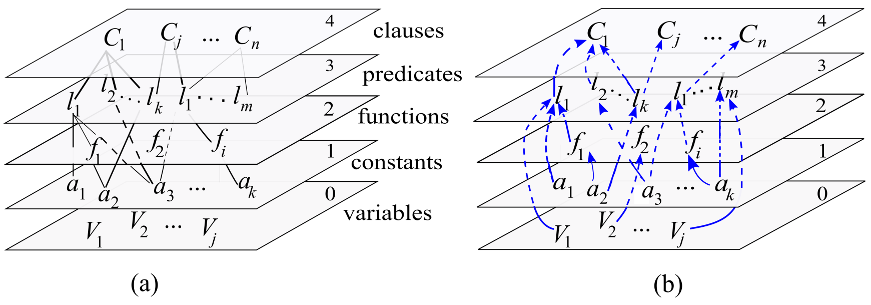


Figure 2.4: (a) First-Order logic space. (b) Weighted First-Order logic space

NeMuS was further extended, Mota et alia in [11], for symbolic inductive learning, then [12] separeted variable and constant spaces into two new ones and showed that predicate invention can be efficiently performed in such a structure. However, even the self-organised inductive reasoning experiment reported by Barreto and Mota, [1], was limited by the fixed number of multi-spaces allowed. Another limitation of the experiment is the fact that it considers only positive clauses and examples. Including negative ones would enable the use of the inductive momentum mechanism, introduced in [11]. Inductive momentum is a mechanism that iteratively "selects" only those atoms not likely to entail negative examples.

(IM) takes into account the positional information of attributes from positive an negative examples to perform such selection. What we aim to achieve is to gather additional information from ground term and how they are distributed across the BK. This is strongly related to neighbourhood definition because constants that appear in instances of the same concept would be in the same neighbourhood. This distribution should obey a certain pattern, which could be used to generate hypotheses in a more efficient way. Although NeMuS does not offer straight information to this, it can be used to acquire it.

Within the problem of generating hypotheses, when it comes to figuring out recursive hypotheses, the previously mentioned ILP methods need to walk the entire path between individuals, as one does in a graph. We aimed to address the problem of shortening this process in [12] in two ways:

1. By defining a language bias able to define "bridging concepts" (which will be explained in the next section), set the size of hypotheses to make simpler clauses, and point to how *auxiliary* concepts can be introduced via invention.

2. By assuming a heuristic that was supposed to be defined in the development from [1]. This would consist in finding an organization pattern associated with recursion, like clusters of concepts. In this way, the task of "walking" a path in a graph would be transformed into checking if two individuals are in the same neighbourhood.

However, the challenge in 2 was not tackled any further. That is mainly due to the concept of neighbourhood not being well defined. Therefore, defining it is the main contribution we expect to make in this work. In the next chapter we will introduce the problem of finding a recursive hypothesis, bridging concepts, and how we plan to make experiments using NeMuS to figure out how the definition of neighbourhoods should be addressed.

Chapter 3

Neural-Symbolic Approach to Inductive Learning

The problem of inductive learning involves a knowledge base of predicates, called background knowledge (*BK*), a set E of examples that the logical description H of the target concept (t) should prove (positive examples, e^{+}) and a set of examples that the target concept should not prove (negative examples, e^{-}). From BK, only a (not necessarily small) portion of its constant terms form what is called Herbrand Base, that may explain how H entails a positive example for the concept t. To reduce the search space of hypotheses generation is a central theme in this field. In this chapter, we shall present a simple logical language to represent hypotheses and description of language biases, as well as give and an informal introduction to our neural-symbolic framework. Using this, we will describe the inductive learning mechanism developed, and the point in which its integration to a connectionist approach can aid the discovery of information about patterns that could help guide the search for useful hypotheses.

3.1 The Logical Language for BK and Hypothesis

The logical language used (for this work) is a fragment of first-order logic based in a clausal formal system [41], in which clauses are divided into two categories. 1) *Initial Clauses*, say B (or the BK), are those belonging to the set of axioms plus the negation of the query; 2) *structured Clauses* are the ones derived by a sort of Linear Resolution [42]. Roughly, if s is a sentence or query, in clausal form, and B is the set of initial clauses, then a deduction of s from B corresponds to deriving an empty clause, $\sqcup$, from $\{\sim s\} \cup B$, or according to

Herbrand theorem, to prove that $\{\sim s\} \cup B$ is *unsatisfiable* and it yields the most general unifier for s.

A set of logical formulae is represented by clauses of literals according to the following terminology. Predicates and constant or atomic symbols start with lowercase letters like $p, q, r, \ldots$ and $a, b, c, \ldots$, respectively. Variables start with capital letters, like $X, Y, \ldots$. A term is either a variable, a constant symbol or a function $f(t_1, \ldots, t_k)$ in which f represents a mapping from terms $t_1, \ldots, t_k$ to an "unknown" individual. If p is a symbol representing a predicate relation over the terms $t_1, \ldots t_n$, then $p(t_1, \ldots, t_n)$ is a valid atomic formula. Predicates and functions are compound symbols with similar structure, but with different logical meaning. A *literal* is either an atomic formula, L, or its negation $\sim L$, and both are said to be complementary to each other. A *Deduction Rule* is a disjunction of literals $\mathtt{L}_1, \ldots,$ $\mathtt{L}_n$, written as $\mathtt{L}_1; \ldots; \mathtt{L}_1$.

There may exist more than one positive literal, and so any Horn clause is represented by `Head;`$\sim$ `Body`, in which literals of the body are called assumptions. So, any standard Logic Program, as defined in [43] , can be represented in the logical language we use. Besides, we use logical negation.

Example 3.1 The Figure 3.1 depicts a hypothetical Family Tree that we then write in this simple logical language.

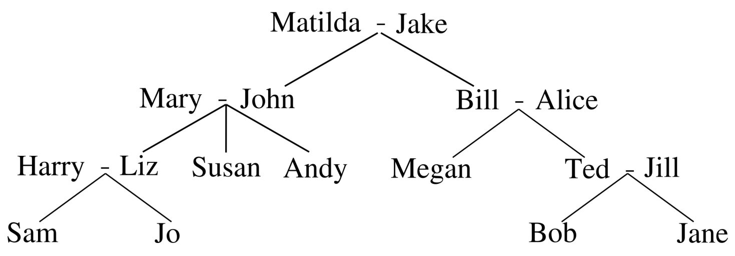


Figure 3.1: A simple family tree.

This family tree can be represented by a sequence of clauses in our language. The aNeMuS system compiles it and each logical element is uniquely indexed and composes a NeMuS structure which we describe in section 3.3.

```
father(jake,bill).    father(ted,jane).       mother(mary,harry).
father(jake,john).    father(harry,sam).      mother(mary,susan).
father(bill,ted).     father(harry,jo).       mother(mary,andy).
father(bill,megan).   mother(liz,jo).         mother(alice,megan).
father(john,harry).   mother(matilda,john).   mother(jill,bob).
father(john,susan).   mother(matilda,bill).   mother(jill,jane).
father(ted,bob).      mother(alice,ted).      mother(liz,sam).
```

In the following sections we shall use this example to give an informal presentation of the NeMuS framework and the method of Inductive Clause Learning (ICL) proposed by Mota *et alia* [12], that was developed in the first steps of this present work. Then we describe the the main problems we aimed to tackle as described in the objectives of our research, section 1.2, our solution for language bias and the need for the use of connectionist approach to enhance the learning mechanism of the aNeMus agent.

3.2 Logic Expressions as Multi-Spaces

Sequences of clauses (or logic programs) are parsed and translated into an internal structure of shared data connected via memory address pointers. This representation is very efficient for dealing with symbols, and the idea of sharing data could be used to create computational efficient neural representations of clauses. Formal logic languages are structurally well defined, and as such can be designed as a hierarchical structure of indexes. Instead of training a neural network with bare data like other approaches, e.g. Komendantskaya [44], an efficient encoding of shared structures was used, and turn them into spaces of index to build up a logic neural multi-space. The investigation of this work concerns specifically examples of neural multi-space for first-order logical expressions, but the structure presented here is not restricted to it.

For this purpose, we use a symbolic hash mapping [45], that maps symbolic objects of the language to a hash key within a finite range. Such a key is not the one used for learning because collisions may happen. So, a separate chaining is used to place keys that collide in a list associated with index, in which every node contains the kind of occurred symbol. Counters were added so that to every new symbol parsed and "hashed", a code hash mapping function generates the next natural number, starting from 1. In this way, every single symbol has a unique index, and such an index shall be the one used for neural learning mechanism.

Thus, a *Logic Expression Coded Corpus (LECC)* is an n-tuple[1] of associative hash mappings $\langle f_C, f_F, f_P, f_W \rangle$, such that $f_C : \mathscr{C} \rightarrow \mathbb{N}$, $f_F : \mathscr{F} \rightarrow \mathbb{N}$, $f_P : \mathscr{P} \rightarrow \mathbb{N}$ and $f_W : \mathscr{W} \rightarrow \mathbb{N}$, in which $\mathscr{C}$, $\mathscr{F}$, $\mathscr{P}$ and $\mathscr{W}$ be a finite sets of constants, functions, predicates, and all other categorical concepts above predicates (e.g. worlds), respectively. Note that the uniqueness of a mapping is only within a corpus space, i.e. the code "1" will be the index of the first predicate found, as well as the first atom four in the case of formula $p(a)$ be the first clause parsed.

[1]Here n is set to 4

3.3 NeMuS Definition

In general, NeMuS is an ordered space for components of logical languages. However, the class of first order-logic applications, for simplicity sake, we consider the interpretation of only five indexed spaces for elements of a first-order language: variables (space 0), atomic constants of the Herbrand Universe (space 1), functions (space 2), predicates with literal instances (space 3), and clauses (space 4). For higher-order logics one just needs to consider spaces above (say 5 as possible worlds) as composed of clause spaces below (in this case 4), and so on. In what follows vectors are written v, and $v[i]$ or v_i is used to refer to an element of a vector at position i.

Each logical element is described by a vector called T-Node, and in particular each element is uniquely identified by an integer code (an index) within its space. In addition, a T-Node identifies the lexicographic occurrence of the element, and (when appropriate) an attribute position. For the purpose of this work we shall omit the dimension attribute of T-Node which is meant to represent negative literals, which are needed for representation of deduction rules. Although most ILP standard methods make use of them because they perform deduction while running their learning algorithms, the method we propose does not need them as it anticipates deduction for positive and negative examples.

Definition 1 (T-Node). *Let $c,a,i,h \in \mathbb{Z}$. A* T-Node (target node) *is a quintuple (h,c,i,a) that identifies an object at space h, with code c and occurrence (or instance) i, at attribute position a. If p is a T-Node, $\zeta(p) = h$, $\kappa(p) = c$, $\alpha(p) = a$ and $\iota(p) = i$. If x is a vector of T-Nodes with size n, and c is a code index of an object occurring in an element of x, then $o(c,x) = \{k_1,\ldots,k_m\}$ is a set of all orders (or indexes) of c within x, $0 \leq k \leq (n-1)$. The set of all T-Nodes will be called $\mathscr{T}_N$.*

Definition 2 (Binding). *If p is a T-Node and $z \in \mathbb{Z}$, then a* Binding *is pair $(p,w)_k$, which represents the influence w of object k over occurrence $\iota(p)$ of object $\kappa(p)$ at space $\zeta(p)$ in position $\alpha(p)$.*

The neural multi-space (NeMuS) is ingeniously built upon a building block we call *Neural Compound* (*nComp* or compound). It is used to mount spaces into bundles of vectors of vectors of vector of compounds. A *nComp* is made of a vector of attributes (T-Nodes), and a vector of bindings to indicate where such instances appear within NeMuS. All other elements follow from it, as we have presented before. Formally we have:

Definition 3 (Neural Compound and Compound Space - cSpace). *A* Neural Compound *– nComp, is a pair (x_a^i, β^i), in which $x_a^i = [c_1,\ldots,c_m]$ is a vector of T-Nodes, each one representing an attribute instance of compound instance i, and β^i is a vector of bindings* *from compound i to all upper compounds where i appears. A* Compound Space (c-Space) *is simply a vector c of compounds.*

A compound space is a vector where all instances of a concept (an object, function, predicate, world, etc) are stored. Note that variables and constants are treated as compounds, and the logical scope of a variable, say X, is identified by the instances of its bindings. Variables can be used as inputs for NeMuS. The function f_β maps a constant i to the vector of its bindings $\vec{x}_i$, as above. For each literal there is a compound space because the same literal, even ground, may appear in more than one formula. In the case of constants and variables, such a space is unique, i.e. a vector with only one element but it is still a vector. This means that we can build logical terms, expressions and formulas only with vectors of vectors of vectors of compound spaces, or a vector bundle-like structure. As elements of such spaces are also interconnected by bindings of influence, from bottom to top, each space can be interpreted as layers. Formally, this can be defined as follows.

Definition 4 (NeMuS - Deep NeMuS). *Let $c^i = [\overrightarrow{s_1^i}, \ldots, \overrightarrow{s_k^i}]$ a vector of cSpaces, or simply a* Conceptual Space *(or layer). Then $d_1 = [c_1^1, c_1^2, \ldots, c_1^n]$ is a vector of conceptual spaces or dimension. A* Deep NeMuS *is a vector of dimensions.*

As mentioned above, negative literals are not used in the examples of this work. However, the *aNeMuS* artificial agent[2] keeps a dimension for negative space of literals in logical formulae.

Example 3.2 Consider the following part the of clauses from Example 3.1

```
father(jake,bill).  mother(matilda,bill).
father(bill,ted).   mother(jill,jane).
```

Codes for the logical elements in the order they are read or scanned are:

[2]*aNeMuS* is a C++ implementation of an artificial cognitive agent the performs symbolic reasoning that is being extended to deal with neural-symbolic learning based on a NeMuS structure.

constant	literal it appears	code	instance of constant
jake	father(jake,bill)	1	1
bill	father(jake,bill)	2	1
bill	father(bill,ted)	2	2
ted	father(bill,ted)	4	1
matilda	mother(matilda,bill)	12	1
bill	mother(matilda,bill)	2	4
jill	mother(jill,jane)	15	2
jane	mother(jill,jane)	9	2

For the predicate space here is part of its coded elements.

predicate	clause it occurs	predicate	instance of predicate
father(jake,bill)	1	1 (father)	1
father(bill,ted)	3	1 (father)	3
mother(matilda,bill)	12	2 (mother)	2
mother(jill,jane)	18	2 (mother)	8

Using these codes, we consider these clauses from the BK in example 3.1. The part of the aNeMuS structure that shows what each vector is composed of is presented as follows, with interspersed explanations for how each space should be read:

```
Clause space:
father(jake,bill): [4,1,1,1]
father(bill,ted): [4,3,1,1]
mother(matilda,bill): [4,12,1,1]
mother(jill,jane): [4,18,1,1]
```

This space encodes the occurrences of the clauses. Each entry is a compound space that encodes a clause, and in this case, all compound spaces have only one compound. The third entry, for example, encodes the clause `mother(matilda,bill)`. This clause is made up of only one term, which is an instance of the predicate `mother`. Therefore, the vector of attributes in this compound is composed of one T-Node: `[4,12,1,1]`. This T-Node is read as follows:

- The first element (4) places the term in the third space. As that is the space of clauses, this means that the term is a clause;
- The second element (12) indicates that this is the twelfth clause in the BK;
- The third element (1) indicates that the term is the first (and only) instance of said clause;
- The fourth element (1) indicates that this term is the first (and only) in the clause.

```
Predicate space:

father: [ [3,1,1,1], [3,1,1,2] ], [ [3,1,3,1], [3,1,3,2] ]
mother: [ [3,2,2,1], [3,2,2,2] ], [ [3,2,8,1], [3,2,8,2] ]
```

This space encodes the occurrences of the predicates. Entries 1 and 2 are compound spaces that encode, respectively, the predicates `father` and `mother`. Analysing the first entry, the vector for `father` has two compounds: one for `father(jake,bill)` and one for the occurrence of `father(bill,ted)`. The T-Node in the `father(bill,ted)` compound, for example, has two element, because this predicate only has two terms (`bill` and `ted`). Inside this compound, the T-Node for `ted` is `[3,1,3,2]`:

- The first element (3) places the term in the third space. As that is the space of predicates, this means that the term appears in a predicate;
- The second element (1) indicates that the term is in the first predicate in the base, that is, `father`;
- The fourth element (3) indicates that the term is in the third instance of said predicate;
- The fifth element (2) indicates that this term is the first in the predicate.

```
Constant space:
    jake: [1,1,1,1]
    bill: [1,2,1,2],[1,2,2,1],[1,2,4,2]
    ted: [1,4,1,2]
    jane: [1,9,2,2]
    matilda: [1,12,2,1]
    jill: [1,15,2,1]
```

This space encodes the occurrences of the constants. Each entry is a compound space made up of a single compound that encodes the occurrences of a specific constant. As there are 3 occurrences of `bill` in the BK, the vector of bindings for `bill` is composed of three T-Nodes. The T-Node in the second binding (`[1,2,2,1]`), for example, is thus read:

- The first element (1) places the occurrence in the first space. As that is the space of constants, this means that `bill` is a constant;
- The second element (2) indicates that this is an occurrence of the second constant in the base (that is, `bill` is the second constant);
- The third element (2) indicates that this is the second instance of said constant;
- The fourth element (1) indicates that this instance of `bill` is the first term of the predicate instance where it happens

This hierarchy of indexed terms is used to perform inductive learning as it is described in the following sections.

3.4 NeMuS-based Inductive Learning

ICL is based on the concept of Least Herbrand Model (LHM), established by John Lloyd [43] to anticipate the elimination of inconsistent hypotheses, at each induction step, before they are fully generated. This is done by *colliding*, via computing the *inductive momentum*, atoms obtained from bindings of arguments from e^+ (candidates to compose LHM) and e^-. Then, a pattern of linkage across verified literals is identified, anti-unification (adapted from [46]) is applied, and a conjecture is generated. The process repeats until the conjecture becomes a closed and consistent hypothesis.

In Figure 3.2 it is depicted a Herbrand Base that may explain how the hypothesis how H entails a positive example for the concept t. In its turn, t may have just one attribute ($p(a_k)$), two attributes ($p(a_k, a_{k1})$), etc.

From the bindings of t's attribute, i.e. $\beta(a_k)$ and possibly $\beta(a_{k1})$, t is connected with attribute mates' bindings. Such connections *bridge* all concepts they may appear in (p_1, q_1, etc.), like a path $\{c_1, \ldots, c_j\}$ in a graph, until it reaches the binding concept of t's last attribute. The interconnected concepts form a *linkage pattern*. For instance, when p_1, q_1 etc., are all the same concept, then a recursive hypothesis may be generated. Invention and hypotheses generation will always take place in the *bridging concepts*' region.

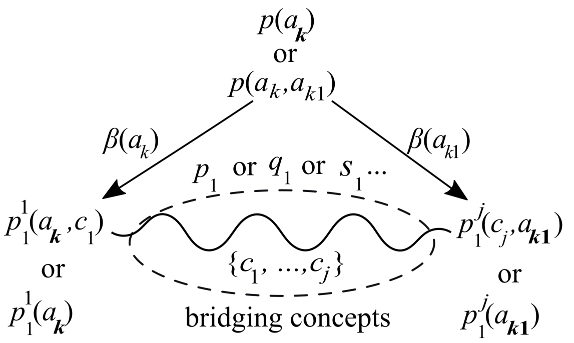


Figure 3.2: Portion of the HB with bindings of t's attributes.

The induction method is based on the following aspects: (a) *Inductive momentum* that iteratively "selects" only those atoms not likely to entail e^-, (b) *linkage patterns* among atoms passed (a), based on internal connections (bridging concepts); (c) *anti-unification* substitutes constants in an atom from the Herbrand Base by variables (and optionally (d) *category-first ordering* as described in our previous work, Schramm *et alia* [47], useful when *BK* contains monadic definitions of categories).

3.4.1 Hypothesis Information

A special form of intersection ρ identifies the common terms between both literals. For instance, in Figure 3.2, suppose $j = 1$, i.e. bridging concepts has just two literals of the same concept p_1: $p_1(a_k,c_1)$ and $p_1(c_1,a_{k1})$. Then $\rho(p_1^1,p_1^2) = c_1$.

Definition 5 (Linkage and Hook-Terms). *Let p and q be two predicates of a BK. There is a* Linkage *between p and q if a same constant, t_h, appears (at least) once in ground instances of p and q. We call t_h a* hook-term, *computed by $t_h = \rho(p,q)$. The* attribute mates *t_h w.r.t. an atom p, written $\overline{\rho(p,q)}_p$ is a set of terms occurring in p, but not in q.*

With p_1^1, p_1^2 as above, c_1 is their hook term and $\overline{\rho(p_1^1,p_1^2)}_{p_1^1} = \{a_k\}$ and $\overline{\rho(p_1^1,p_1^2)}_{p_1^2} = \{a_{k1}\}$. This would form the definite clause $p(a_k,a_{k1}) \leftarrow p_1(a_k,c_1) \wedge p_1(c_1,a_{k1})$, which is not generated but it is built using anti-unification to generalize over its hooked ground literals. In the following definition we use the standard notion of a substitution θ as a set of pairs of variables and terms like $\{X_1/t_1,\ldots,X_n/t_n\}$.

Definition 6 (Anti-substitution and Anti-unification). *Let G be a first-order expression with no constant term and $X_1,\ldots,X_n$ be free variables of G, e is a ground first-order expression, and $t_1,\ldots,t_n$ are constants terms of e. An* anti-substitution *is a set $\theta^{-1} = \{t_1/X_1,\ldots,t_n/X_n\}$ such that $G = \theta^{-1}e$, and G is called a simple* anti-unification *of e. The* anti-unification

f_θ^{-1} *maps a ground atom e to its corresponding* anti-substitution *set that generalizes e, i.e.* $f_\theta^{-1}(e) = \theta^{-1}$

Note that in the original definition of anti-unification proposed by Idestam-Almquist, [46], G should be the generalisation of two ground expressions. Here, as we build a hypothesis by adding literals from a definite clause the definition is

Definition 7 (Anti-unification on linkage terms)**.** *Given two literals $p(a_k, a_z)$ and $q(a_z, a_{k1})$. A* linkage term*, say Z_0, for their hook term a_z, is a variable that can be placed, by anti-unification, in the hook's position wherever it appears in the ground instance that will produce two non-ground literals.*

The definite clause $p(a_k, a_{k1}) \leftarrow p_1(a_k, c_1) \wedge p_1(c_1, a_{k1})$, from Figure 3.2 ($j = 1$), $\theta^{-1} = \{a_k/X, a_{k1}/Y, c_1/Z\}$ is incrementally anti-unified, and so the following general clause is found: $p(X,Y) \leftarrow p_1(X,Z) \wedge p_1(Z,Y)$.

This concept is the fundamental operation to generate a hypothesis because it generalizes ground formulas into universally quantified ones. Before describing how negative examples are used we have the following definition.

Definition 8 (Hypothesis)**.** *Let H be a formula with no constant term, S be a set of ground atoms formed by concepts and constants from a Herbrand universe H_u, t is a ground atom with k terms (which belongs to the base of constants, H_0) such that $t \notin S$. We say that H is a* hypothesis *for t with respect to S if and only if there is a set of atoms $E = \{e_1, \ldots, e_n\}$ and a θ^{-1} such that for*

1. *every a of t, there is some $e_i \in E$ and $\rho(t, e_i) = a$*
2. *every e_i and e_{i+1} $\rho(e_i, e_{i+1})$ is not empty.*
3. *when 1 and 2 hold, then $H = \theta^{-1}(\{t\} \cup E)$.*

An open hypothesis *is one that at least one term of t have not been anti-unified. Thus, 1, 2 and 3 will always generate a closed least hypothesis.*

Recall that *learning should involve the generation of a hypothesis and to test it against positive and negative examples*. Such a "test" could be done while the hypothesis is being generated. This is the fundamental role of the following concept. Note that, in the interest of saving space, the following sections shall use logic notation. It is important, however, to recall the definitions given in section 3.3, as the method described runs on top of the NeMuS structure in the *aNeMuS* agent.

3.4.2 Inductive Momentum

In the following definition, a_k is a constant originating from the path of a positive example, and b_k is another constant originating from a negative example.

Definition 9 (Inductive Momentum). *Let (x_a^i, w_i) and (x_a^j, w_j) be two spaces of atoms i and j, representing l^+ and l^- atomic formulas (literals) in the Herbrand base. If $\exists k$ and m, from e^+ and e^-, such that $l^+ \in \beta(k)$ and $l^- \in \beta(m)$, i.e. k is an element of x_a^i and m is an element of x_a^j, then the* inductive momentum *between l^+ and l^- with respect to k and m is*

$$I_\mu(x_a^i, x_a^j)_m^k = \begin{cases} inconsistent(0) & \text{if } i = j, \text{ and} \\ & \quad \iota(k, x_a^i) = \iota(m, x_a^j) \\ consistent(1) & otherwise \end{cases}$$

When $i = j$, then it is assumed $|x_a^i| = |x_a^j|$ since they are the same code in the predicate space. When it is clear in the context we shall simply write $I_\mu(l^+, l^-)_m^k$ rather than the T-Node vector notation.

Inconsistent hypotheses are eliminated by computing the I_μ, i.e. the *collision* of atoms, obtained from bindings of arguments from e^+, via computing the *Inductive momentum*,.

Example 3.3 The *BK* is formed by ground instances of binary and monadic predicates (not limited to them), atoms and Herbrand universe H_u as follows.

1. $\{p_1(a, a_1), \dots, p_k(a_k, a)\}$,
2. $\{q_1(a_1, b_1), \dots, q_j(b_j, a_1)\}$,
3. $\{r_1(c_1, a_k) \dots, r_m(a_k, c_m)\}$,
4. $\{t_1(b_j), \dots, s_1(c_1), \dots, v_1(c_m), \dots\}$,
5. target $p(X)$, with e^+: $p(a)$ and e^-: $\sim p(b)$.

As depicted in Figure 3.3 Atom candidates to form consistent hypotheses, via anti-unification [46], from the Least Herbrand Model (LHM) [43], pass I_μ. In Figure 1, on the right, $q_j(a_1, b_1)$ pass I_μ while $q_j(b_j, a_1)$ do not.

When the BK is compiled its correspondent NeMuS structure is also built. The induction mechanism, at each step, adds to the premise of a hypothesis the next available atom from the bindings of a constant only if such an atom “resists" the inductive momentum.

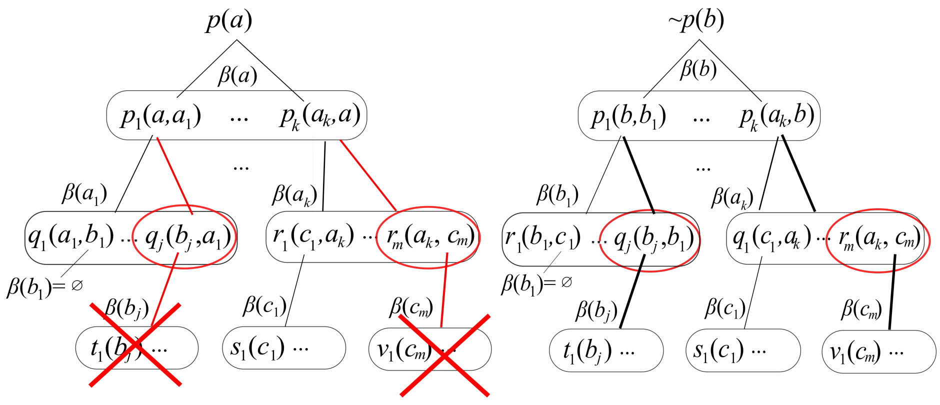


Figure 3.3: figure
Search space formed by $\beta(a)$ and $\beta(b)$ yielded from I_μ

Step	partial hypothesis	Inductive Momentum (I_μ)
1	$p(X) \leftarrow p_1(X,Y)$	not applied
2	$p(X) \leftarrow p_1(X,Y) \wedge q_1(Z,Y)$	$I_\mu(q_1(a_1,b_1), r_1(b_1,c_1))^{a_1}_{b_1}$ = *consistent*
3	$p(X) \leftarrow p_1(X,Y) \wedge q_2(Z,Y)$	$I_\mu(q_1(a_1,b_2), r_2(b_2,c_2))^{a_1}_{b_1}$ = *consistent*
...	...	...
n	$p(X) \leftarrow p_1(X,Y) \wedge q_j(Z,Y)$	$I_\mu(q_j(b_j,a_1), q_j(b_j,b_1))^{a_1}_{b_1}$ = *inconsistent*

For a partial hypothesis would be $p(X) \leftarrow p_1(X,Y) \wedge q_j(Z,Y)$ with $\theta^{-1} = \{a/X, a_1/Y, b_j/Z\}$, but the equivalent path from negative example would reach $q_j(b_j, b_1)$. This would allow $p(b)$ to be also deduced, which is not what it is expected from a sound hypothesis. Thus, this hypothesis is dropped. For this example, a sound hypothesis could be $p(X) \leftarrow p_k(Y,X) \wedge r_1(Z,Y) \wedge s_1(Z)$. Had the target concept be $s(X)$ and positive example $s(c_1)$, then a possible hypothesis generated would be
$s(X) \leftarrow s_1(X) \wedge r_1(X,Z_0) \wedge p_k(Z_0,Z_1)$.

3.5 Inductive Clause Learning from the Herbrand Base

The following method joins all ideas described in section 3.4. We shall use standard logic program notation for clauses just for readability sake, but recall that Amao language treats $q \leftarrow p$ as $q \vee \neg p$. The general idea of ICL can be summarised in three mains steps.

1. to walk across the linkages found in the Herbrand Base in order to select atoms as candidates for composing hypotheses, as well as those to oppose the compositions
2. to compute I_μ of atoms as candidates for anti-unification that were selected from positive and negative linkages.

3. to generalize, via anti-unification, only atomic formulas likely to build consistent hypotheses, i.e. those composed by atoms consistent with respect to I_μ

However, this general description lacks a proper treatment for language biases as away to guide the search for useful relations to be added in the hypothesis formation. As pointed out in section 1.2, language biases has been recently treated as meta-level approaches, which (roughly) define hypotheses schemes in logic programming. We partially achieve this using language bias for the reduction of hypotheses space. Furthermore, it gives no clue at which point a connectionist method of pattern recognition could be used as heuristics for the selection of shorter hypothesis, e.g. recursive hypotheses. In the following description we shall consider a dyadic theory with no function terms, in order to make it simple the understanding of our proposed method of bridging concepts to be added in the inductive learning mechanism.

3.5.1 Bridging Concepts

Predicate invention, according to ILP definition, is a *bias* defined by the user via a declarative language. It is a way to deal with predicates missing from the *BK* for lack of information.

Consider a BK defined by the set $B = \{father(a_1, c_1), mother(b_1, c_1), father(a_2, c_2), mother(b_2, c_2), \ldots, father(a_n, c_n), mother(b_n, c_n)\}$, and suppose that the target predicate is $ancestor(X, Y)$. Without information on how the concepts of parenthood or ancestry work, it is still possible to detect that (a) there are two distinct concept relations where a constant c_i participates as a second attribute; and (b) there can be many instances of both $father$ and $mother$, and no constant appears as first argument of both. Therefore, there seems to be a new concept that *captures the property shared by all individuals/objects when appearing as the first attribute of either relation.* This concept bridges two regions of concepts via a path across their attributes, as can be seen in Figures 3.4 and 3.5, where $B = \{father(jake, alice), mother(matilda, alice), \ldots\}$.

Bridging concepts consists in, when two or more different predicates essentially perform the same "role" in a base, inventing a predicate that generalises them. This makes the hypotheses space smaller, and makes it simpler to find a recursive hypothesis because it is much simpler to deal with a single concept than it is to deal with many, as far as recursion is concerned.

The concept of $ancestor$ is connected with the bindings of other constants that appear along with $jake$ or bob in predicates that are already part of the BK (we call these constants attribute mates). Such connections bridge all concepts related to these constants, like a path in a graph, until the binding concept of $ancestor(jake, bob)$'s last attribute (that is, bob) is

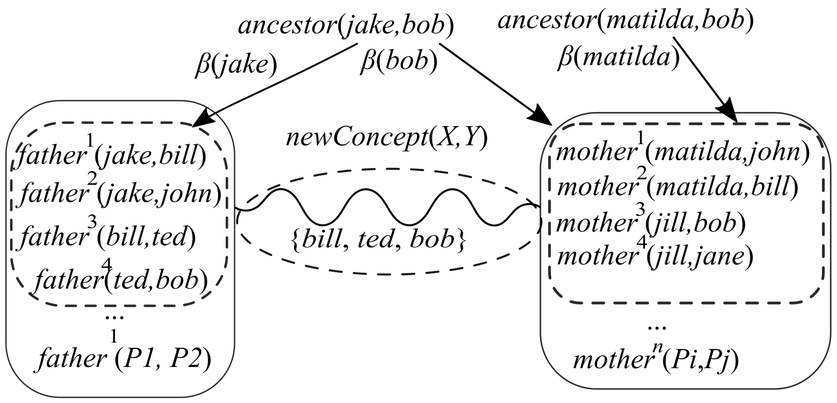


Figure 3.4: *newConcept* bridges two regions of concepts via a path.

reached. In the current example, *jake* and *bob* are linked via only one concept: the one discovered in the previous step, which we know to be *parent*.

On a closer look at Figure 3.5, it is possible that our general approach to predicate invention generates hypotheses that do not look like what we expect. For example $ancestor(X,Y)$ given $ancestor(jake,bob)$, might generate $ancestor(X,Y) \leftarrow father(X,Z_0) \wedge p_1(Z_0,Y)$. It is assumed that two concepts, say c_1 and c_2, are "specialisations" of another concept c whenever there are objects appearing as second argument of both, but can only appear as first in one of them.

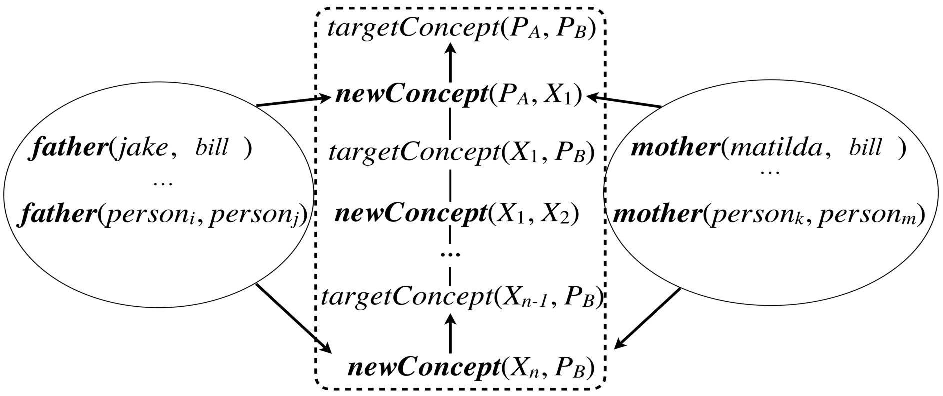


Figure 3.5: Bridging concept reduces hypothesis expression.

We say that an *invented predicate bridges two regions of concepts*, allowing for a simpler generalisation of ground rules into hypothesis. This is illustrated in Figure 3.5

Every time either or both concepts are involved in a hypothesis generation, the new concept is used to intentionally define the target predicate. So, from Figure 3.5 the rule base would be

$$newConcept(X,Y) \leftarrow (father(X,Y) \vee mother(X,Y))$$

The new concept is *parent*, which is not a target concept to consider induction, but shall be used as a bridge or as a base form of a hypothesis, while the target shall be a linear or recursive linkage pattern (in our approach), or tail recursive.

3.5.2 'Bias" as Invention of Predicates

aNeMuS performs a similarity training on NeMuS's weights using the vector representation for each constant as well as for literals. Those with similar linkages end up with similar weight values associated to the argument they have and their position within them. Besides, bias may be used to add non targeted new predicates.

Non user bias: "automated" invention

For this, it is necessary "to invent" a predicate, say p_0, such that H_1 becomes a closed hypothesis. For the sake of space θ^{-1} will be suppressed when anti-substituions are clear.

User defined bias for invention

When an assumption that r_l defines another concept, say p_b, then H_1's body would have p_b and H_2's head would have p_b, rather then p_0. This would be something like `assuming` $r_l(X,Y)$ `defines` $p_b(X,Y)$. As there can be many bindings, we close an open hypothesis for each possible combination of bindings. Then, we keep computing the momentum and expanding a new branch for each combination (as explained in Sections 3.5.3 to 3.5.2).

Suppose we wish to induce a hypothesis that defines the ancestry relation from this base. The general form of requesting this, according to the literature, is simply written:

`ancestor(X,Y) given` (**knowing**) `E`$^+$ `and E`$^-$

However, using a language bias can guide the search towards more suitable candidates to add to the hypothesis according to:

- the desired size of the hypothesis, which is usually set as **dyadic** theory (Chomsky-like normal form grammars, i.e. clauses with, at most, two literals in the body); this can lead to
- invention of predicates, which can be done either by automatically generating indexed labels for new predicates (i.e. $\mathtt{p}_0$, $\mathtt{p}_i$, ...) or making known to the induction process the possible forms of bridging concepts (i.e. `assuming mother(X,Y) or father(X,Y) defines parent(X,Y)`

Considering these Informally, the language bias we propose is a set of strings formed by one of the following possibilities:

1. `induce pX`

   Free induction, the general case seen above with `ancestor`

2. `induce pX assuming qX [or rX] defines sX (or inventing sX from qX or rX).`

   induction with concept biased invention

3. `induce pX assuming qX [or rX] defines sX (or inventing sX from qX or rX) a dyadic theory.`

   induction with dyadic concept biased invention

4. `induce pX knowing pa [`$\sim$`pb]`

   induction by examples

5. `induce pX knowing pa [`$\sim$`pb] assuming qX [or rX] defines sX`

   induction by examples with invention biased by concepts

6. `induce pX [knowing pa [`$\sim$`pb]] [assuming a dyadic theory.]` (default is automatic invention)

   induction by examples with invention biased by concepts

7. `induce pX assuming/inventing general dyadic theory`

   induction by examples automatically inventing dyadic theories

8. `induce pX knowing pa [`$\sim$`pb] assuming qX [or rX] defines sX a dyadic theory`

   induction by examples with concept biased invention of dyadics.

Based on this informal grammar description, the recognition of a well-formed induction request is performed by the automaton in Figure 3.6

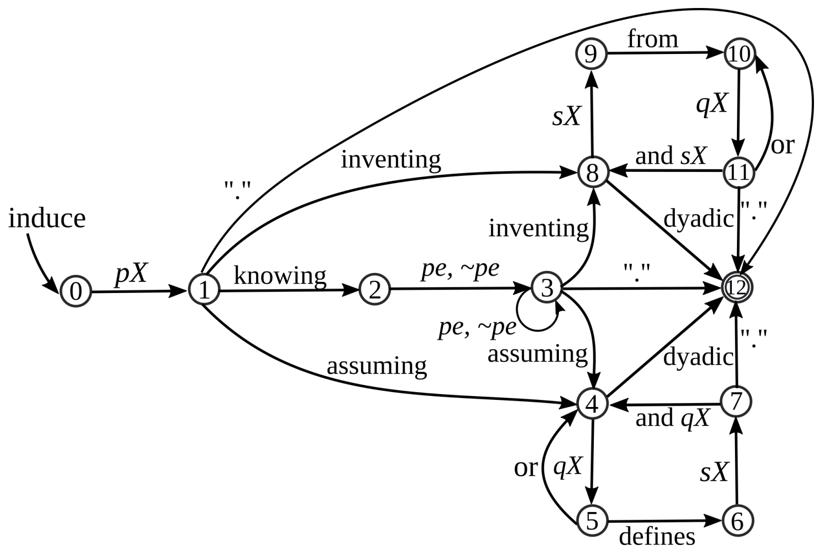


Figure 3.6: Automaton that performs the proposed language bias

Continuing with the running example, a valid request for *aNeMuS* would be:
induce ancestor(X,Y) **knowing** ancestor(matilda,bob) **assuming** father(X,Y) **or** mother(X,Y) **defines** parent(X,Y) a dyadic theory.

The first step would be to associate `ancestor(matilda,bob)` with `ancestor(X,Y)`. This is done by anti-substituting `matilda` for `X` and `bob` for `Y`. We represent this as $\theta_0^{-1} =$ `{matilda/X, bob/Y}`. This θ_0^{-1} is associated to the initial conjecture (that is, the initial incomplete hypothesis), which is generated as $\mathtt{C}_0 =$ `ancestor(X,Y)`.

The second step is to get β`(matilda)` and β`(bob)`, that is, the set of all bindings of `matilda` and all bindings of `bob`, respectively. The request made to aNeMuS does not include any negative examples, and so the inductive momentum automatically succeeds.

i	atom/hypothesis	θ_i^{-1} / $\beta(a)$
0	$ancestor(matilda, bob)$	$\{matilda/X, bob/Y\}$
C_0	$ancestor(X,Y) \leftarrow$	$\beta_1(matilda)$ and $\beta_1(bob)$
1	$I_\mu = consistent$, no hook	$mother(matilda, john)$ and $mother(matilda, bill)$

Now, for each pair $p_i \in \beta$`(matilda)` and $q_i \in \beta$`(bob)`, we check if there is any **hook-term** (Definition 5), i.e. a constant common to both p_i and q_i. If one such constant is found it establishes a "chain" that connects `matilda` to `bob`. The first case checked would be $p_i =$ `mother(matilda,john)` and $q_i =$ `father(ted,bob)`, where there is no hook-term. Checking all other cases, it is clear that this is the case for all values of p_i and q_i. However, both constants that appear in β`(matilda)` (that is, both `john` and `bill`) also have other bindings, through which it may still be possible to reach bob. Therefore, we continue the process of generating a hypothesis.

The next step, then, is to check for bias for p_i. For example, if $p_i =$ mother(matilda,john), by anti-unifying $\theta_1^{-1} =$ {matilda/X, john/Y} we have parent(X,Y) $\leftarrow$ mother(X,Y) as the first closed conjecture, or hypothesis. That is, we reach a hypothesis that states, in informal terms, that every mother is a parent. This then becomes one of the hypotheses generated, which should now be used to extend C_0 to C_1. We note, though, that in C_0's θ_0^{-1}, we have bob/Y. As bob and john are different constants, it is not possible to assign them both to Y. Therefore, we rename john/Y to john/Z_0.

For the purpose of visualization:

β(matilda) = {mother(matilda,john), mother(matilda,bill)}

β(bob) = {father(ted,bob), mother(jill,bob)}

i	atom/hypothesis	θ_i^{-1} / $\beta(a)$
0	$ancestor(matilda,bob)$	$\{matilda/X, bob/Y\}$
C_0	$ancestor(X,Y) \leftarrow$	$\beta_1(matilda)$ and $\beta_1(bob)$
1	$I_\mu = consistent$, no hook	$mother(matilda, john)$ and $mother(matilda, bill)$
bias	$mother(X,Y)$	match both β_1
for	$parent(X,Y)$	rename variable
H_0^i	$parent(X,Y) \leftarrow father(X,Y)$	$\{matilda/X, bob/Y, john/Z_0\}$
C_1	$ancestor(X,Y) \leftarrow parent(X,Z_0)$	$\beta_1(john)$, $mother(alice,ted)$

Now, the next conjecture $C_1 = C_0 \cup$ parent(X,Z_0) $=$ ancestor(X,Y) $\leftarrow$ parent(X,Z_0). This conjecture is not closed yet because there is still a variable in the head of the clause that has not been reached by a predicate (that would be bob/Y), thus we must continue. The same happens in the case of $p_i =$ mother(matilda,bill), where (after the renaming process) $\theta_1^{-1} =$ {matilda/X, bill/Z_0} and parent(X,Z_0) $\leftarrow$ mother(X,Z_0). Therefore, in this case, the next conjecture would be $C_1 = C_0 \cup$ parent(X,Z_0) , that is, $C_1 =$ ancestor(X,Y) $\leftarrow$ parent(X,Z_0). As both versions of C_1 are the same, we can proceed with only one of them.

Once again, to ease visualization:

β(john) = {father(jake,john), father(john,harry), father(john,susan), mother(matilda,john)}

β(bill) = {father(jake,bill), father(bill,ted), father(bill,megan), mother(matilda,bill)}

β(bob) = {father(ted,bob), mother(jill,bob)}

We now repeat the process of checking for constants in common both for each pair $p_i \in \beta$(john) and $q_i \in \beta$(bob), and for each pair $p_i \in \beta$(bill) and $q_i \in \beta$(bob). In the latter case, when checking $p_2 =$ `father(bill,ted)` and $q_1 =$ `father(ted,bob)`, a hook-term is found: `ted`.

Following the other previously described steps, we establish `ted/Z`$_1$, and reach $C_2 =$ `ancestor(X,Y)` $\leftarrow$ `parent(X,Z`$_0$`)` $\wedge$ `parent(Z`$_0$`,Z`$_1$`)` $\wedge$ `parent(Z`$_1$`,Y)`. Unfortunately, we cannot do so, as the induction is in dyadic theory mode, and therefore C_2 has already reached its maximum body size of 2 at the inclusion of `parent(Z`$_0$`,Z`$_1$`)`. Due to this, we must invent a new predicate `P`$_0$`(Z`$_0$`,Y)` $\leftarrow$ `parent(Z`$_0$`,Z`$_1$`)` $\wedge$ `parent(Z`$_1$`,Y)`, which aggregates the last 2 literals included in C_2 into one single definition.

In order to proceed, we notice that finding a hook-term has led to the discovery of a shallow recursive path `bill` $\rightarrow$ `ted` $\rightarrow$ `bob`. This allows us to replace `P`$_0$`(Z`$_0$`,Y)` with `ancestor(Z`$_0$`,Y)`, since `ancestor(bill,bob)` $\leftarrow$ `parent(bill,ted)` $\wedge$ `parent(ted,bob)`.

As a consequence of that, we have the following 2 clauses:

1. `ancestor(X,Y)` $\leftarrow$ `parent(X,Z`$_0$`)` $\wedge$ `parent(Z`$_0$`,Z`$_1$`)` $\wedge$ `parent(Z`$_1$`,Y)`

2. `ancestor(Z`$_0$`,Y)` $\leftarrow$ `parent(Z`$_0$`,Z`$_1$`)` $\wedge$ `parent(Z`$_1$`,Y)`

Then, replacing clause 2 in clause 1, we reach the hypothesis we want:

$C_2 =$ `ancestor(X,Y)` $\leftarrow$ `parent(X,Z`$_0$`)` $\wedge$ `ancestor(Z`$_0$`,Y)`

Although the reached hypothesis is valid, it is not helpful in establishing the ancestry relation between individuals in the BK, as it is recursive and has no base case. In this example, the base case would be `ancestor(X,Y)` $\leftarrow$ `parent(X,Y)`. This clause can be added to the hypothesis by either:

1. Including another language bias in the request, establishing that `ancestor(X,Y)` $\leftarrow$ `parent(X,Y)`, that is, that `parent(X,Y)` defines `ancestor(X,Y)`; or

2. Including another positive example in the request that would help the induction process to reach the conclusion that `ancestor(X,Y)` $\leftarrow$ `parent(X,Y)`. Some options for such examples would be `ancestor(jake,john)` or `ancestor(mary,susan)`.

Note that in the construction of the hypothesis, the reason why `P`$_0$`(Z`$_0$`,Y)` could be replaced with `ancestor(Z`$_0$`,Y)` was that a shallow recursive path was found through the constants in the base. The other situation in which this replacement could be made would be if a deep recursive path was found. Current methods find this path by iteratively walking through it, starting at `X` and stopping only when `Y` is reached. We propose instead that it can be found via a pre-processing of the data in the BK that allows for analyzing regions of neighbourhoods.

3.5.3 Hypotheses' space formation via Selection of Candidates

Induce concept $t(X,Y)$, given $e^+ = \{t(a_k,a_{k1})\}$ and $e^- = \{t(b_k,b_{k1})\}$, θ^{-1} is the anti-substitution set. From NeMuS get $\beta(a_k)$ and $\beta(a_{k1})$. The initial view of the space of possible hypotheses that can be formed using atoms from the Herbrand Base and anti-unification is illustrated in Figure 3.7.

Each $\beta(a_k)_i$ in a triangle represents a hypothesis formation branch that can be expanded following the bindings of the attribute in e^+. Some of them may allow the deduction of e^-, and thus *inductive momentum* is applied to validate fetched atoms. After adding an anti-unified literal from $\beta(a_k)_1$ into the premise of the hypothesis being generated, say H_1, the next induction step will take a branch from the attribute-mates of a_k to compute I_μ, generalize and so on. In the breadth-first walk the generation of H_1 is postponed until all triangle branches have been initially exploited. For completeness sake it is implemented breadth-first.

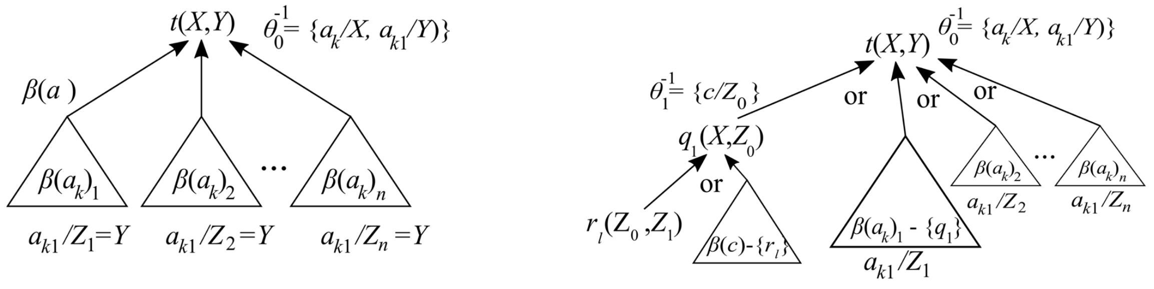


Figure 3.7: (a) Hypotheses' space from e^+; (b) Anti-unify atoms along with invention (steps *a* to *m*).

Let q_1 be a positive literal coming from a positive example, and η a negative literal from a negative example.According to Definition 9, a_k is attribute of q_1 and b_k is an attribute of η.

```
1. If $I_\mu(q_1,\eta)_{b_k}^{a_k} = 1$ for all $\eta \in \beta(b_k)$, then
2.     If $q_1(a_k,a_{k1}) \in \beta(a_k)$ and $q_1(a_k,a_{k1}) \in \beta(a_{k1})$, then
3.         $H_1$: $t(X,Y) \leftarrow q_1(X,Y)$,        $\theta_1^{-1} = \{a_k/X, a_{k1}/Y\}$,
4.     else if $q_1(a_k,c) \in \beta(a_k)$ and $c \neq a_{k1}$, then
5.         $H_1$: $t(X,Y) \leftarrow q_1(X,Z_0)$,        $\theta_1^{-1} = \{a_k/X, c/Z_0\}$
6. else get another $q_j \in \beta(a_k)$ and repeat 1-5, until consistent hypotheses candidates are found
```

For a consistent H_1, $\exists r_l \in \beta(c)$, and

```
a. if $\exists\ r_l \in \beta(c)$ and $r_l \in \beta(a_{k1})$ and $r_l \neq q_1$, then anti-unify $r_l(c,a_{k1})$ against previous $\theta^{-1}$
b.     $H_1$: $t(X,Y) \leftarrow q_1(X,Z_0) \wedge r_l(Z_0,Y)$. (Chain in ILP)
c. else if no bias is given and $I_\mu(r_l,\eta') = 1$ then invent $p_0$ and $H_2$
```

```
d.    $H_1$: $t(X,Y) \leftarrow q_1(X,Z_0) \wedge p_0(Z_0,Y)$
e.    $H_2$: $p_0(X,Y) \leftarrow r_l(X,Z_0)$. The search now is guided by $\beta(c)$
f. **else if** $r_l(X,Y)$ defines $p_b(X,Y)$ and $r_l \neq q_1$, **then invent** $p_b$ in $H_1$ and $H_2$
g. **else if** $r_l(X,Y)$ defines $p_b(X,Y)$ and $r_l = q_1$, **then**
h.        **if** $q_1(c,a_{k1}) \in \beta(a_{k1})$, **then** $H_1$: $t(X,Y) \leftarrow q_1(X,Z_0) \wedge q_1(Z_0,Y)$ and
i.            **invent** $H_2 : t(X,Y) \leftarrow q_1(X,Y)$, $H_3 : t(X,Y) \leftarrow q_1(X,Z_0) \wedge t(Z_0,Y)$
j.        **else if** $q_1(c,a_{k1}) \notin \beta(a_{k1})$ and $q_1(a_k,c)$ and $q_1(c,a_{k1})$ $\boxed{\textbf{region's weights are similar}}$,
k.            **invent** $H_1$: $t(X,Y) \leftarrow q_1(X,Y)$
m.                $H_2$: $t(X,Y) \leftarrow q_1(X,Z_0) \wedge t(Z_0,Y)$
```

Table 3.1 show all steps of the hypothesis generation that our method suggests, in which we spot the parts that lacked either a formal definition (in the case of language biases), or the exact point at which connectionist method will enhance the inductive learning process.

i	atom/hypothesis	θ_i^{-1} / $\beta(a)$
0	$ancestor(jake,bob)$	$\{jake/X,bob/Y\}$
C_0	$ancestor(X,Y) \leftarrow$	$\beta_1(jake)$ and $\beta_1(bob)$
1	$I_\mu = consistent$, no hook	$father(jake,alice)$ and $father(ted,bob)$
bias	$father(X,Y)$	match both β_1
for	$parent(X,Y)$	rename variable
H_0^i	$parent(X,Y) \leftarrow father(X,Y)$	$\{jake/X,bob/Y,alice/Z_0\}$
C_1	$ancestor(X,Y) \leftarrow parent(X,Z_0)$	$\beta_1(alice)$, $mother(alice,ted)$
bias	$mother(X,Y)$	match both $\beta_1(alice)$
for	$parent(X,Y)$	rename variable
H_1^i	$parent(X,Y) \leftarrow mother(X,Y)$	reaches maximun body size, do not add
H_1	another$parent(Z_0,Z_1)$.	**Check for region similarity**
H_2^i	$ancestor(X,Y) \leftarrow parent(X,Y)$	
H_3	$ancestor(X,Y) \leftarrow parent(X,Z_0) \wedge$	$ancestor(Z_0,Y)$

Table 3.1: Complete execution of the ICL with biases and connectionist selection of candidates

This chapter presented our approach to deal with language biases in order to guide the hypothesis generation. The expressive power of our notation do not mix the logical language of hypothesis representation with the settings to be followed by based on invention and bridging concepts. In the next chapter we shall describe the experimental results we found in order to figure out the feasibility of a connectionist method to be used in a self-organized inductive learning mechanism.

Chapter 4

Towards a Self-Organized Inductive Learning Mechanism

This chapter is meant to present the directions to follow in order to meet the targets of this work. For this, we briefly describe some partial results achieved so far, presented in [12]. The main components of inductive clausal learning are presented, as well as its limitations and we pinpoint where it shall benefit from self-organizing map of concepts to produce a self-organized inductive learning. We end the chapter by presenting a schedule of activities for the next six months.

4.1 SOM Experiments

Throughout this chapter we shall write constants with vector notation to represent their encoding when needed. For instance, the constant "Jake" will be written $\overrightarrow{jake}$.

Before proceeding any further, a change was made to the original data: the constant `andy` was removed from the BK, since it only has one binding (`mother(mary,andy)`) and that might make give way to confusing results in the future if, for example, we tried to create a hypothesis for the "sibling" relation. Once the necessary alteration was done, we started the experiments, the basis for which consists of the following:

1 Create a table where each line is a β representing where a binding of an object in the predicate space;

		space	predicate	instance	att position
Jake	father(jake,bill)	3	1	1	1
Bill	father(jake,bill)	3	1	1	2
Jake	father(jake,john)	3	1	2	1
John	father(jake,john)	3	1	2	2
...					

2 Create a table where each line is a β representing a binding of an object in the constant space;

	space	constant	instance	att position
Jake	1	1	1	1
Bill	1	2	1	2
Jake	1	1	2	1
John	1	3	1	2
...				

Table 4.1: Constant table

3 Normalize each line of both tables so that each β according to vector normalization, turn each embed in its unit vector form given by the formula.

for any vector $\vec{v}$, $\hat{v} = \frac{\vec{v}}{\|\vec{v}\|}$

		space	predicate	instance	att position
Jake	father(jake,bill)	0.866025	0.288675	0.288675	0.288675
Bill	father(jake,bill)	0.774597	0.258199	0.258199	0.516398
Jake	father(jake,john)	0.774597	0.258199	0.516398	0.258199
John	father(jake,john)	0.707107	0.235702	0.471405	0.471405
...					

Table 4.2: Normalized predicate table

	space	constant	instance	att position
Jake	0.5	0.5	0.5	0.5
Bill	0.316228	0.632456	0.316228	0.632456
Jake	0.377964	0.377964	0.755929	0.377964
John	0.258199	0.774597	0.258199	0.516398
...				

Table 4.3: Normalized constant table

4 Subtract each β in the predicate table from its corresponding β in the constant table. In this way, we obtain the actual vectors between the space of constants and the space of predicates;

	space	constant	instance	att position
Jake	-0.366025	0.211325	0.211325	0.211325
Bill	-0.458369	0.374257	0.058029	0.116058
Jake	-0.396633	0.119765	0.239531	0.119765
John	-0.448908	0.538895	-0.213206	0.044993
...				

Table 4.4: Subtraction table

5 Use vectors from the normalized predicate table to calculate a vector orthogonal to the predicate hyperplane. From that table it was picked three vectors at random, say $\vec{v_1}, \vec{v_2}, \vec{v_3}$, presented in Table 4.5;

$\vec{v_1}$	0.15359	0.051197	-0.090427	0.051197
$\vec{v_2}$	0.39347	0.038707	-0.25368	0.038707
$\vec{v_3}$	0.387299	0.387299	-0.645497	0.1291

Table 4.5: Vectors used to calculate vector orthogonal to hyperplane

$$P_s = \begin{bmatrix} 0.15359 & 0.051197 & -0.090427 & 0.051197 \\ 0.39347 & 0.038707 & -0.25368 & 0.038707 \\ 0.387299 & 0.387299 & -0.645497 & 0.1291 \end{bmatrix}$$

The hyperplane passing across these vectors can be found by the standard method found in linear algebra, which find the unit vector orthogonal to such a hyperplane. This is done by reducing matrix P_s to its Reduced Row Echelon Form and solving the following linear equations:

$$\begin{bmatrix} 1 & 0 & 0 & 0.3601797955 \\ 0 & 1 & 0 & 0.8715918482 \\ 0 & 0 & 1 & 0.5390635833 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \vec{0}$$

$0.3601797955x_4 = -x_1$

$0.8715918482x_4 = -x_2$

$0.5390635833x_4 = -x_3$

Assuming $x_4 = -1$, then:
$x_1 = 0.3601797955$;
$x_2 = 0.8715918482$;
$x_3 = 0.5390635833$.

Therefore, $\vec{n} = (0.3601797955, 0.8715918482, 0.5390635833, -1)$

6 Calculate the cross-product between each vector from the table obtained in step 4 without the “space” column, and the last three coordinates of the orthogonal vector obtained in step 5. In this way, we obtain the projection of the vectors between constants and predicates on the predicate space. The resulting table has one line for each binding of an object in the base, ignoring the "space" column;

		concept	instance	att position
Jake	father(jake,bill)	-0.3252426117	0.3955141473	-0.07027153558
Bill	father(jake,bill)	-0.1205916414	0.4754122067	0.1511707161
Jake	father(jake,john)	-0.3040919501	0.2241511977	-0.1442123169
John	father(jake,john)	0.1889519122	0.578110532	0.4763272813
...				

Table 4.6: Cross-products table

7 Create a table where each line is constructed by mapping all reference vectors of an object to a single vector. This operation is done by simply appending all lines that are bindings of the same object.

The table obtained in step 7, which can be seen in Appendix A, would be used as input in a SOM. Although this SOM was originally meant to be implemented and incorporated into aNeMuS, we later came to the conclusion that it would be more advantageous to make our experiments separated from what was already implemented. This is due to the fact that the current aNeMuS structure is relatively stable, and we believed that it would not be worth modifying until we were sure of the results of the experiments.

Experiments were made with python 3, using Jupyter and MiniSom, a minimalistic implementation of SOM. The maps generated were 9x9, and were trained for one thousand epochs.

Our first experiment consisted simply of the steps listed above, with no further additions. The resulting SOM can be seen in Figure 4.1.

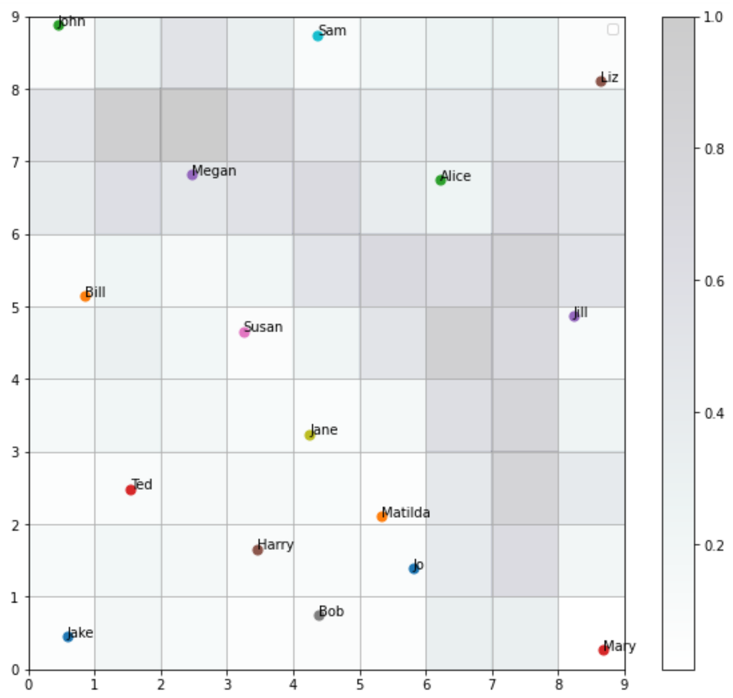


Figure 4.1: Map generated in first experiment, following the listed steps with no further additions.

It is notable in the map of Figure 4.1 that all the individuals that were neither fathers nor mothers appear in the four leftmost columns of the map. All mothers appear in the next three columns, and fathers can be seen in the next five. However, the node representing `jake` appears to be closer to "mother" constants than to "father" ones. This is possibly due to the fact that, as `jake` is the first father in the base, it does not have bindings where it appears as the second argument of a predicate (which is what happens to other fathers in the base, since we have information about their parents). Additionally, in general, the last couple of generations appear in the first few lines of the map and earlier generations appear lower.

As we want to create clusters of objects induced by the vector representations of their occurrences in the concepts where they appear, i.e. conceptual clusters, we believed that the "space" column should be incorporated somehow into the vectors of the resulting input table. As a first attempt to achieve this, we attempted to find a formula for linear transformation, to reduce the dimensions of our vector in a way that incorporated the value of the "space" column into all following columns. However, no satisfactory formula was reached, and we decided to try a different approach.

This different approach consisted of calculating the mean of the cells corresponding to "space" for each object from the table obtained in step 4. Then, we made two additional variations of the previously obtained input: one by multiplying each "concept" cell by its corresponding "mean of spaces", and another by multiplying each "attribute" cell by the same corresponding "mean of spaces".

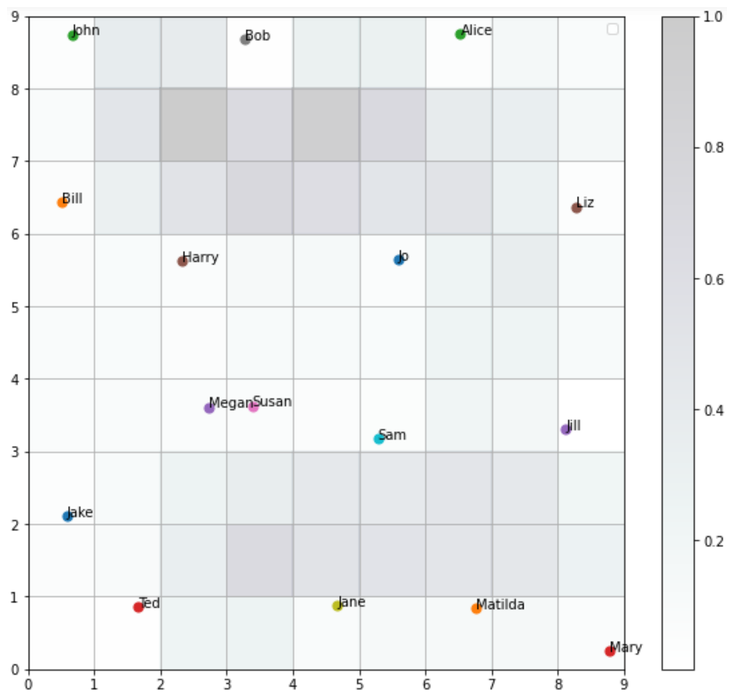


Figure 4.2: Map generated in second experiment, by multiplying each "concept" cell by its corresponding "mean of spaces".

In the map seen in Figure 4.2, most fathers appear in the seven uppermost lines, while all mothers are in the next three, and those who are neither appear in the last five lines. Once again the node that represents `jake` appears to be closer to instances of mothers and children than to other fathers, and so the approach taken for this map does not seem correct. However, it is still interesting to note that all fathers appear in the same or almost the same column as the mother of their children (and as their children themselves, when these are not also fathers). Older generations appear mostly to the right, while the newer ones are to the left

The configuration of the map in Figure 4.3 is as follows: instances of father are on the upper half of the map, while mothers are on the lower right quarter, and instances of constants that are neither appear on the lower left quarter. This is the first map where `jake` appears along with the other fathers, and so, inserting the "space" information into the "attribute" column seems a promising approach. We note also that all mothers appear lower and to the right of the fathers of their children, who all seem to appear below their fathers and to the left of their mothers.

As using the mean of the cells in the "space" column did not yield linearly separable clusters, we tried another strategy: as our BK only has predicates of the form $father(a_1, a_2)$ and

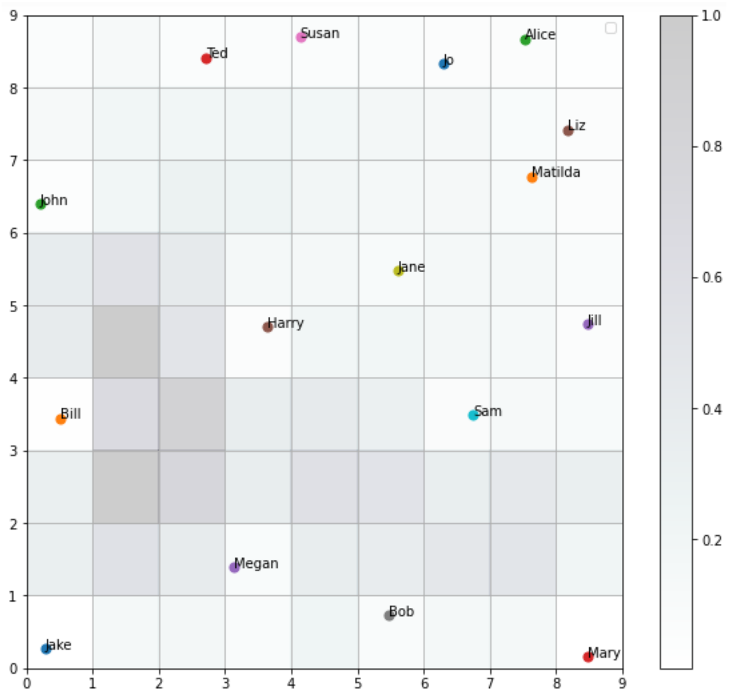


Figure 4.3: Map generated in second experiment, by multiplying each "attribute" cell by its corresponding "mean of spaces".

$mother(b_1, b_2)$, we decided to calculate the deviation between all cells in the "space" column that corresponded to "father" relations, and multiply each "attribute" cell corresponding to these relations in the cross-products table by this value. We then did the same calculation for the cells in the "space" column that corresponded to "mother" relations, and multiplied each corresponding "attribute" cell in the cross-products table by this new value.

The division in the map of Figure 4.4 is clearer than the others, as regions are linearly separable. Fathers to the left on the map, mothers are on the right, and children (who are neither fathers nor mothers) are in the middle.

Although last attempt resulted in linearly separable regions for each concept, these regions still have to be identified manually. We attempted to make this division automatic by applying the k-means clustering technique on a dataset composed of the vectors with the final weights of the nodes that represent each individual in the original base.

The first of the three resulting clusters consisted of all constants that were neither fathers nor mothers, as well as "Ted", "Jo" and "Mary". The second cluster included all fathers except for "Ted", and the third cluster included only "Jo" and "Mary". This result is shown in Figure 4.5

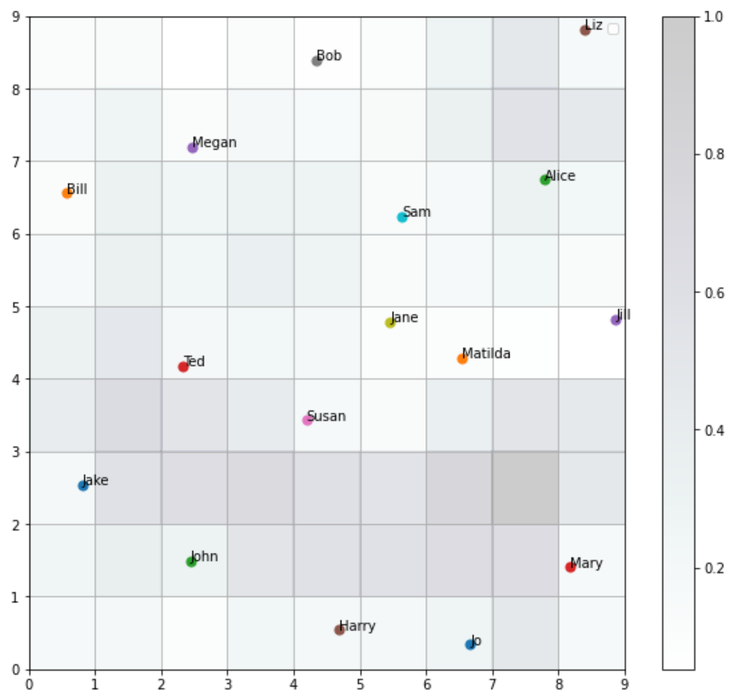


Figure 4.4: Map generated in the third experiment, where the deviation of all "space" cells of each predicate were calculated, and each "attribute" cell was multiplied by the deviation value corresponding to its predicate

	names	0
0	Jake	1
1	Bill	1
2	John	1
3	Ted	0
4	Megan	0
5	Harry	1
6	Susan	0
7	Bob	0
8	Jane	0
9	Sam	0
10	Jo	2
11	Matilda	0
12	Alice	0
13	Mary	2
14	Jill	0
15	Liz	0

Figure 4.5: Clusters generated by applying k-means on vectors of final weights

4.2 K-Means Experiment

The SOM experiment did not generate the expected clusters. In order to verify if this result was too unusual, we decided to make another experiment based on the k-means clustering technique. The database for this experiment consisted of four vectors, one for each attribute of the concepts in the original base ($father$ and $mother$). The values in these vectors for any given constant were "1" if that constant appeared in the base at least once as that attribute in that concept, and "0" if not. These vectors f_1 (first attribute of $father$), f_2 (second attribute of $father$), m_1 (first attribute of $mother$) and m_2 (second attribute of $mother$) are split in Tables 4.7 and 4.8. They were created manually in this experiment, but it would be easy to generate them automatically using aNeMuS.

	Jake	Bill	John	Ted	Megan	Harry	Susan	Bob	Jane
f_1	1	1	1	1	0	1	0	0	0
f_2	0	1	1	1	1	1	1	1	1
m_1	0	0	0	0	0	0	0	0	0
m_2	0	1	1	1	1	1	1	1	1

Table 4.7: The vectors f_1, f_2, m_1 and m_2 were used in this experiment

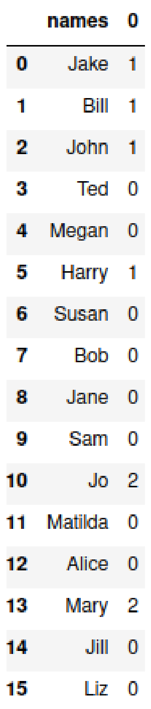
	Sam	Jo	Matilda	Alice	Mary	Jill	Liz
f_1	0	0	0	0	0	0	0
f_2	1	1	0	0	0	0	0
m_1	0	0	1	1	1	1	1
m_2	1	1	0	0	0	0	0

Table 4.8: The vectors f_1, f_2, m_1 and m_2 were used in this experiment

The resulting three clusters correctly separate the concepts, as shown in figure 4.6. Therefore, this approach is preferable to using SOM.

4.3 Discussion

The experiments described in the previous section were made with the purpose of finding a definition for the concept of neighbourhoods. The expected outcome of these experiments could help us to find organization patterns in the BK to guide the search for good hypotheses. In [48], where the use of SOM for maps of semantic concepts was first proposed, Kohonen states that "any realistic semantic brain maps would need a much more complicated, probably hierarchical model". This served as inspiration for the basis of the aNeMuS symbolic



	names	0
0	Jake	1
1	Bill	1
2	John	1
3	Ted	0
4	Megan	0
5	Harry	1
6	Susan	0
7	Bob	0
8	Jane	0
9	Sam	0
10	Jo	2
11	Matilda	0
12	Alice	0
13	Mary	2
14	Jill	0
15	Liz	0

Figure 4.6: Clusters generated by applying k-means on vectors f_1, f_2, m_1 and m_2

framework, with its distinct spaces for each concept, linked in a hierarchical way to one another.

The original idea was to integrate the aNeMuS framework with a self-organizing structure in order to provide it with "higher-level meanings" (term used by Kohonen in [48]). These are formally described by means of logical language, because logical expressions have sound semantic interpretation. In our methodological procedure, we projected the vectors between constants and predicates on the predicate hyperplane to observe how both spaces would influence each other, in this case, constants on predicates. As such, the result should resemble clusters of similar relations over constants.

Figure 4.7 shows that the last experiment yielded a map with linearly separable clusters, one for each concept: fathers, mothers, and constants that were neither. Knowing which objects are in which regions of the map could reduce the hypotheses search space and "prune" the search. Considering the relation `ancestor(X,Y)`, if `X` is in the region of those that are neither fathers nor mothers, then the relation is false for any value of `Y`. This could also be discovered by simply examining the bindings of each constant, but as this could require walking multiple paths in a graph, it would eventually become an NP-hard problem given a BK big enough. We claim that there must be a way to associate neighbourhoods of concepts that can yield observable patterns of recursion.

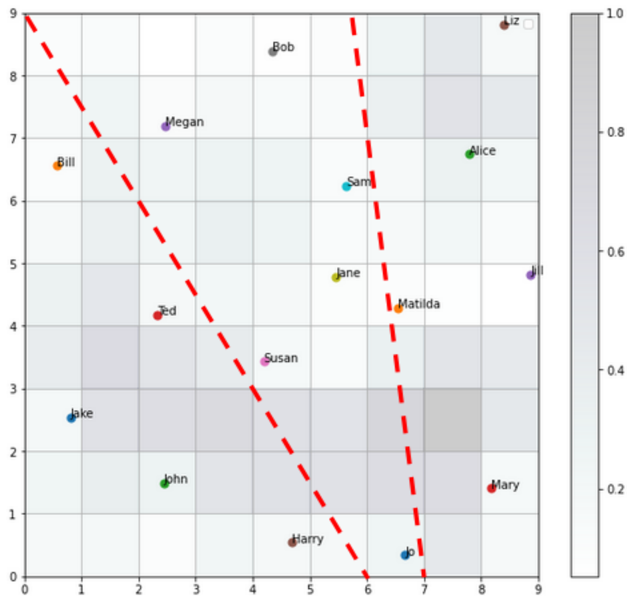


Figure 4.7: Map generated in the third experiment. The red dotted lines indicate the regions of concepts. The leftmost region indicates fathers, while rightmost one indicates mothers and the middle region includes only elements which are neither.

On the other hand, it is very likely that pre-processing all this information may be time-consuming. However, as we did not expect the outcome presented in previous sections, more analysis is necessary to figure out more effective experiments. One possible avenue to explore would be using a Graph Attention-like component "plugged" into the aNeMuS structure so that we could use a structured self-attention to pinpoint where such patterns could emerge.

Chapter 5

Conclusion and Future Work

In this work, we aimed to find a suitable method to generate neighbourhood patterns to be used for inductive learning and reasoning to reduce the search space of hypotheses. Besides, we defined a language bias to guide the process of generating hypotheses. The latter objective was achieved as presented in Sections 3.5.1 and 3.5.2. These were incorporated into the Inductive Clause Learning mechanism of aNeMuS. The results of the former objective can be seen in the presentation of the experiments in Chapter 4.

Nevertheless, there are aspects that we have not been able to take into account, but that would be beneficial to consider for future research. The first aspect would be to put into question the use of SOM (which would be part of a revision of the methodology). Though our reasons for using this specific method have been explained, we believe that research may benefit from exploring other methods as well, such as KNN.

Additionally, we have not yet been able to incorporate the weights generated by aNeMuS into the induction. An interesting direction we wish to take is to use them as bias for the projection of vectors between constants and predicates onto the predicate space. For this, we would consider the statistical distribution of objects according to their occurrences within the structure of the language, similar to what CILP ([22], [23], [24]) did successfully for connectionist inductive propositional logic programming. They compute the influence of literals in a logic program by calculating their statistics within the structure of a logic program, and using them to yield the initial weights and biases of the neural network obtained from it. Just with that, they achieved faster convergence in comparison to other approaches. Perhaps the path to take, in our case, may be getting the statistics of the objects (logical terms) and other logical compounds across the spaces of the aNeMuS structure.

Furthermore, when projecting vectors onto the predicate hyperplane, we used three arbitrary vectors from this hyperplane to find a vector orthogonal to it (normal vector). Using three other vectors would have led to a different normal vector being found, and possibly, to a different configuration on the maps generated. We do not believe that this would change the topology of the maps very much, but wish to account for this possibility by verifying if there exists a normal distribution of hyperplanes, and using this distribution to find one most significative hyperplane that could be used reliably.

However, the achieved results do not suggest that our hypothesis could be proved. Perhaps our methodology should be revised, our data representation changed, or perhaps it is not feasible to find useful neighbourhoods of concepts at all. One very interesting suggestion that came to us was to use the vector representation of BERT ([49]) for each constant and concept, mostly used for natural language processing. As such, we might translate our logical representation into natural language expressions, and then use BERT. As a result, our experiment would have a strong inclination towards natural language inference[50].

Appendix A

Table of Embedding SOM Training

constant	concept	instance	attribute	concept	instance	attribute	concept	instance	attribute	concept	instance	attribute
Jake	-0.3252	0.3955	-0.0072	-0.3041	0.2242	-0.0147	-0.6775	-0.3254	-0.0538	-0.5119	-0.3779	-0.0456
Bill	-0.1206	0.4754	0.0154	-0.0116	0.4896	0.0259	-0.0402	0.3813	0.0174	-0.3439	-0.0682	-0.0489
John	0.1890	0.5781	0.0486	0.2676	0.6877	0.0617	0.1625	0.5927	0.0471	-0.4373	-0.0163	-0.0567
Ted	0.4073	0.6525	0.0722	0.4320	0.7970	0.0823	0.3049	0.7214	0.0668	-0.0351	0.2101	0.0120
Megan	0.5539	0.7103	0.0884	0.3561	0.5048	0.0846	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Harry	0.6526	0.7555	0.0997	0.6027	0.8652	0.1013	0.4841	0.8247	0.0885	0.9761	0.2653	0.0651
Susan	0.7208	0.7910	0.1077	0.5681	0.6339	0.1216	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Bob	0.7694	0.8189	0.1136	0.6341	0.6796	0.1335	0.1602	0.1730	0.0152	0.0000	0.0000	0.0000
Jane	0.8051	0.8412	0.1180	0.6842	0.7165	0.1428	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Sam	0.8326	0.8592	0.1214	0.7232	0.7467	0.1501	0.0514	0.1108	0.0073	0.0000	0.0000	0.0000
Jo	0.8532	0.8739	0.1240	0.7542	0.7716	0.1560	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Matilda	0.2704	0.3202	0.0593	0.3911	0.3741	0.0788	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Alice	0.6206	0.4589	0.1145	0.6368	0.5243	0.1217	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Mary	0.7779	0.5944	0.1451	0.7461	0.6403	0.1446	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Jill	0.8479	0.6893	0.1614	0.7998	0.7191	0.1576	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Liz	0.8831	0.7537	0.1709	0.8302	0.7734	0.1657	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table A.1: Final table of embeddings to be used for SOM

Bibliography

[1] L. Barreto and E. Mota, "Self-organized inductive reasoning with NeMuS," in *IJCAI - Neural-Symbolic Learning and Reasoning Workshop* (D. Doran, A. d'Avila Garcez, and F. Lecue, eds.), vol. 14, NeSy Association, August 2019.

[2] K. K. Teru, E. Denis, and W. L. Hamilton, "Inductive relation prediction by subgraph reasoning," in *ICML*, 2020.

[3] J. R. Quinlan, "Learning logical definitions from relations," *Machine Learning*, vol. 5, pp. 239–266, 1990.

[4] R. S. Michalski, "A theory and methodology of inductive learning," in *Machine Learning: An Artificial Intelligence Approach* (R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, eds.), pp. 83–134, Berlin, Heidelberg: Springer Berlin Heidelberg, 1983.

[5] A. Cropper and S. Dumančić, "Inductive logic programming at 30: a new introduction," 2020.

[6] J. Waskan, "Connectionism." [Online; accessed 10-July-2021].

[7] M. Dougherty, "A review of neural networks applied to transport," *Transportation Research Part C: Emerging Technologies*, vol. 3, no. 4, pp. 247–260, 1995.

[8] J. McMCarthy, "Epistemological challenges for connectionism," *Behavioral and Brain Sciences*, vol. 11, no. 1, pp. 11–44, 1988.

[9] R. Evans and E. Grefenstette, "Learning explanatory rules from noisy data," 2018.

[10] E. d. S. Mota and Y. B. Diniz, "Shared Multi-Space Representation for Neural-Symbolic Reasoning," in *NeSy 2016* (T. R. Besold, L. Lamb, L. Serafini, and W. Tabor, eds.), vol. 1768, CEUR Workshop Proceedings, July 2016.

[11] E. d. S. Mota, J. Howe, and A. d'Avila Garcez, "Inductive learning in shared neural multi-spaces," in *NeSy 2017* (T. R. Besold, A. d'Avila Garcez, and I. Noble, eds.), vol. 2003, CEUR Workshop Proceedings, July 2017.

[12] E. Mota, J. M. Howe, A. Schramm, and A. d'Avila Garcez, "Efficient predicate invention using shared "NeMuS"," in *IJCAI - Neural-Symbolic Learning and Reasoning Workshop* (D. Doran, A. d'Avila Garcez, and F. Lecue, eds.), vol. 14, NeSy Association, August 2019.

[13] A. Cropper, S. Dumančić, R. Evans, and S. H. Muggleton, "Inductive logic programming at 30," *Machine Learning*, vol. 111, no. 1, pp. 147–172, 2022.

[14] K. Inoue, A. Doncescu, and H. Nabeshima, “Completing causal networks by meta-level abduction,” *Machine learning*, vol. 91, no. 2, pp. 239–277, 2013.

[15] S. Muggleton, “Inductive logic programming,” in *New generation computing*, vol. 8, pp. 295–318, Ohmsha, Ltd., 1991.

[16] S. MUGGLETON and W. BUNTINE, “Machine invention of first-order predicates by inverting resolution,” in *Machine Learning Proceedings 1988* (J. Laird, ed.), pp. 339–352, San Francisco (CA): Morgan Kaufmann, 1988.

[17] S. Muggleton, L. De Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue, and A. Srinivasan, “Ilp turns 20,” *Machine Learning*, vol. 86, pp. 3–23, Jan 2012.

[18] A. Cropper, S. Dumančić, and S. H. Muggleton, “Turning 30: New ideas in inductive logic programming,” 2020.

[19] A. Cropper and S. H. Muggleton, “Metagol system.” https://github.com/metagol/metagol, 2016. "Online; accessed 02-April-2021".

[20] A. Cropper and S. Tourret, “Logical reduction of metarules,” *Machine Learning*, vol. 109, pp. 1323–1369, Jul 2020.

[21] A. d’Avila Garcez, T. R. Besold, L. de Raedt, P. Földiak, P. Hitzler, T. Icard, K.-U. Kühnberger, L. C. Lamb, R. Miikkulainen, and D. L. Silver, “Neural-symbolic learning and reasoning: Contributions and challenges,” in *AAAI Spring Symposium on Knowledge Representation and Reasoning: Integrating Symbolic and Neural Approaches*, pp. 18–21, AAAI Press, March 2015.

[22] A. d’Avila Garcez and G. Zaverucha, “The connectionst inductive learning and logic programming system,” *Applied Intelligence*, vol. 11, pp. 59–77, July 1999.

[23] A. S. d’Avila Garcez, K. Broda, and D. Gabbay, *Neural-Symbolic Learning Systems: Foundations and Applications, Perspectives in Neural Computing*. Springer-Verlag, 2002.

[24] M. V. M. França, A. S. D’Avila Garcez, and G. Zaverucha, “Relational knowledge extraction from neural networks,” in *Proceedings of the 2015th International Conference on Cognitive Computation: Integrating Neural and Symbolic Approaches - Volume 1583*, COCO’15, (Aachen, DEU), p. 146–154, CEUR-WS.org, 2015.

[25] L. C. Lamb, T. R. Besolf, and A. d’Avila Garcez, “Human-like neural symbolic computing,” Dagstuhl Reports 5, Creative Commons, May 2017.

[26] R. Evans and E. Grefenstette, “Learning explanatory rules from noisy data,” *CoRR*, vol. abs/1711.04574, 2017.

[27] W.-Z. Dai, Q. Xu, Y. Yu, and Z.-H. Zhou, “Bridging machine learning and logical reasoning by abductive reasoning,” in *Advances in Neural Information Processing Systems*, vol. 32, pp. 2815–2826, 2019.

[28] T. Rocktäschel and S. Riedel, "Learning knowledge base inference with neural theorem provers," in *Proceedings of the 5th Workshop on Automated Knowledge Base Construction*, (San Diego, CA), pp. 45–50, Association for Computational Linguistics, June 2016.

[29] G. Šourek, V. Aschenbrenner, F. Železny, and O. Kuželka, "Lifted relational neural networks," in *Proceedings of the NIPS Workshop on Cognitive Computation*, COCO'15, (Aachen, DEU), pp. 52–60, CEUR-WS.org, 2015.

[30] G. Šourek, Manandhar, Suresh, Schockaert, Steven, and O. Kuželka, "Learning predictive categories using lifted relational neural networks," in *Inductive Logic Programming* (J. Cussens and A. Russo, eds.), (Cham), pp. 108–119, Springer International Publishing, 2017.

[31] Gustav Šourek, Vojtech Aschenbrenner, Filip Železny, Steven Schockaert, Ondřej Kuželka, "Lifted relational neural networks: Efficient learning of latent relational structures," *Journal of Artificial Intelligence Research*, vol. 62, pp. 69–100, May 2018.

[32] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[33] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *ArXiv*, vol. abs/1812.04202, 2018.

[34] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, "Graph convolutional networks: a comprehensive review," *Computational Social Networks*, vol. 6, p. 11, Nov 2019.

[35] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," NIPS'17, (Red Hook, NY, USA), p. 1025–1035, Curran Associates Inc., 2017.

[36] A. C. A. R. P. L. Y. B. P. Velickovic, G. Cucurull, "Graph attention networks," Proceedings of the 7th International Conference on Learning Representations, 2018.

[37] J. X. H. M. I. K. J. Zhang, X. Shi and D.-Y. Yeung, "Gaan: Gated attention networks for learning on large and spatiotemporal graphs," Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, 2018.

[38] T. M. J. Chen and C. Xiao, "Fastgcn: fast learning with graph convolutional networks via importance sampling," Proceedings of the 7th International Conference on Learning Representations, 2018.

[39] P. C. J. Ma and W. Zhu, "Depthlgp: Learning embeddings of out-of-sample nodes in dynamic networks," Proceedings of the 32nd AAAI Conference on Artificial Intelligence, 2018.

[40] Z. Zhang, "A note on spectral clustering and svd of graph data," 09 2018.

[41] N. Vieira, *Máquinas de Inferência para Sistemas Baseados em Conhecimento*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, 1987. PhD Thesis.

[42] A. Robinson, “A machine-oriented logic based on the resolution principle,” *Journal of the ACM*, vol. 12, no. 1, pp. 23–42, 1965.

[43] J. W. Lloyd, *Foundations of Logic Programming, Second, Extended Edition*. Springer-Verlag, 1993.

[44] E. Komendantskaya, “Unification neural networks: unification by error-correction learning,” *Logic Journal of the IGPL*, vol. 19, pp. 821–847, May 2010.

[45] A. G. Konheim, *Hashing in Computer Science: Fifty Years of Slicing and Dicing*. John Wiley & Sons, 2010.

[46] P. Idestam-Almquist, “Generalization under Implication by Recursive Anti-unification,” in *International Conference on Machine Learning*, pp. 151–158, Morgan-Kaufmann, 1993.

[47] A. C. M. Schramm, E. d. S. Mota, J. Howe, and A. d’Avila Garcez, “Category-based inductive learning in shared nemus,” in *NeSy 2017* (T. R. Besold, A. d’Avila Garcez, and I. Noble, eds.), vol. 2003, CEUR Workshop Proceedings, July 2017.

[48] T. Kohonen, “Self-organizing semantic maps,” *Biological Cybernetics, Springer-Verlag*, vol. 61, no. 1, pp. 241–254, 1989.

[49] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, (Minneapolis, Minnesota), pp. 4171–4186, Association for Computational Linguistics, June 2019.

[50] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, “A large annotated corpus for learning natural language inference,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, (Lisbon, Portugal), pp. 632–642, Association for Computational Linguistics, Sept. 2015.