Systematic Literature Review on Inductive Logic Prorgamming (ILP)

Francesco Cassano,
francesco.cassano2@studio.unibo.it,
University of Bologna, Cesena, Italy.

2022

Inductive Logic Programming or ILP is a research area that has emerged in last thirty years. The idea that gave the bases for ILP is merging machine learning and logic programming, in that way we can extract the best of both technologies and overcome many of the limitations of its forebears. ILP paradigm allows new methods for machine learning, and it allows to investigates the inductive construction of first-order clausal theories from a knowledge base and sets of positive and negative examples for a specific domain. That clausal will compose the logic program who can generalise positive example and it can exclude negative example.
The document realised is a Systematic Literature Review (SLR). SLR is a revision type that uses analytical and repeatable methods to collect and analyse data. In detail in the document will be analysed the theoretical basis of ILP, and the main approaches and technologies will be presented, described and compared.

Keywords: ILP, Inductive Logic Prorgamming, Machine Learning, Logic programming, First Order Logic, Systematic Literature Review

Contents

1 Introduction

Muggleton defines ILP as the intersection between Machine Learning and Logic Programming [MUG91b]. Both the technologies are aimed at human reasoning and in particular the ability to deduce, learn and elaborate that is the reasoning. The reasoning is classified in: deduction, induction and abduction. In Logic Programming we use deduction. This means that we use rules and predicates to model data, and an implicit truth is extracted from true premises, then recognised and shaped. By its nature, deduction is mainly used to mathematical proofs.
What we try to implement in ILP is inductive reasoning. In other words we try to develop tool and techniques to induce hypotheses from observations (positive examples) and synthesise new knowledge from experience. Through induction, in fact, it is possible to create hypotheses, logic programs made up from a set of rules. Premises and examples provide more or less strong evidences to support hypotheses in order to validate and use them. Nevertheless, we cannot be sure that the new knowledge produced from hypotheses is true, therefore there will be some degree of reliability [SM94].
Because of the probabilistic nature of induction, induction is often treated as abductive reasoning with the addition of a strong justification. Abductive reasoning is considered very powerful for this kind of systems, because it allows one to work with probabilities with much more freedom, effectively making abduction the appropriate reasoning for hypothesis formation.
This paper is a Systematic Literature Review on ILP. In section 2 we will present the methodology with witch the SLR is carried out, and in section 3 we will present the result of all the SLR stages and ILP will be explained in detail.

2 Introduction to SLR

Systematic Literature Review is a form of secondary study in which research synthesis is based on quantitative statistical methods. SLR examines all the primary studies with the aim of integrating and synthesising evidence related to a specific research question and analyse the topics from various points of view. In detail it can highlight advantages and disadvantages, and it can provide guidance on how to structure the research activity [Woh14]. SLR is usually used as the first step in new researches, moreover, illustrating the way in which SLR has been carried out allows the reader to appreciate the completeness and reliability of the data. The advantages of SLR are mainly:

- The well-defined methodology makes it less likely that the results of the literature are biased, although it does not protect against publication bias in the primary studies.
- They can provide information about the effects of some phenomenon across a wide range of settings and empirical methods. If studies yield consistent results, systematic reviews provide evidence that the phenomenon is robust and transferable. If the studies yield inconsistent results, sources of variation can be studied.
- In the case of quantitative studies, it is possible to combine data using meta-analytic techniques. This increases the likelihood of detecting real effects that individual smaller studies are unable to detect.

To write a SLR we have to follow a certain procedure that is divided into various phases. The first step is the most important. That is, **to set goals** of the SLR, in addiction we describe the topic and the research questions that will lead the review. Second step is the definition of the literature **search strategy**. In this step we will identify the keywords for the queries. The queries will be used to find papers that will be part of the start-set. Once that papers are found, we need to select only useful ones, so we define **selection criteria**. The selection criteria are a set of properties that a paper must possess and during the SLR allow the reviewer to choose only the articles deemed most important and reliable. After the first filtering we move on to the **data extraction and refinement** phase, where forms are created to extract data and standardise those to be compared. Then data is clustered so that the research questions can be discussed (**clustering and discussion**). In the end we **evaluate the result** to demonstrate the usefulness of the review. [BK07]

2.1 Research Questions

The first step of SLR is to declare the goals. In this document we will discuss the main method and technologies for *Inductive Logic Programming* application. We will also provide a theoretical base for understanding ILP.
The main objectives of this paper are to explain *what* ILP is and *how* and *why* ILP is used. For this reason research questions will be:

- What is ILP?
- For what ILP is used?
- What are main search methods?
- What are the fundamental features of logical language that make it crucial in ILP?
- What are main implementation?

The research questions will lead the review and will allow to research and analyse papers focusing on review objectives and will help on write SLR.

2.2 Search Strategy

Defining the search strategy allow us to perform a reliable and repeatable research.
These characteristics are important in order for the reader to understand the footprint that was given to the paper and how the data that influenced it were found, while also demonstrating the reliability of the sources. The search strategy chosen is named *snowballing*.
For the paper research, we use digital archive, where queries are made using keywords. Queries found the most useful papers for the review. Once the paper in start-set have been chosen, we analyse bibliography and the papers that cite the papers in start-set to find other useful paper. This is how backwards and forwards steps are done in snowballing.
To compose the start-set we used Google Scholar as an archive, which is a good alternative to ensure that there is no bias in favour of a specific author and publisher. Filters have also been applied to the search later which will be discussed in the next section, as highlighted in [Woh14] and [Woh16]. Queries that we have used to select papers for start set are:

Q1: $Inductive\ Logic\ Programming \vee ILP$

Q2: $(Inductive\ Logic\ Programming \vee ILP) \vee Machine\ Learning \wedge Logic\ Programming$

Q3: $Inductive\ Logic\ Programming \wedge (Algorithms \vee Utilities \vee Applications)$

After having established the goals of the document, search method and queries, we need to specify how to choose the most meaningful result. Title, keywords, abstract relevance and publication date were assessed. The papers are ordered by relevance and the chosen articles were written in last 10 years. Additional selection criteria were included in order to obtain only the most useful works for review. Only after these steps has been taken and the start set has been identified we can begin with the snowballing procedure, which consists in evaluating the usefulness of the research cited, or in which the chosen article is cited.

2.3 Selection Criteria

The huge number of paper make it necessary the introduction of constraint on the selection of the primary search to be used for the discussion of the research questions. This is why the selection criteria are like filter that we use to specialise the result of the paper search. The selection criteria that we use are:

- Title: Is the title relevant to the research? Does the title have any bearing on the topic pf the review?
- Publication venue: Was the article published by a reputable journal?
- Authors: Do we know that the authors have published relevant paper in the area studied before?
- Citations: Has the article been cited at least 30 times? (the number of citation is low in order to not exclude more recent paper)

If a paper that we are examining does not meet one or more of the constraints just defined, there is not a reason to exclude it, because it is a matter of inclusion criteria. That is why if a paper meet all the criteria it should compose the start set and move on to the next stage, while the paper that meet some constraints is considered as candidate for inclusion and will be examined and evaluated carefully [Woh14].
The exclusion criteria are now applied. To exclude a paper it is sufficient that it does not satisfy a single constraint. The exclusion criteria used are:

- Pertinence of the keywords: exclusion of research paper that do not mention inductive logic programming or machine learning, logic programming and probability or ILP and (specific technology in paper) in the keywords (where present).
- Pertinence of the abstract: exclusion of research papers which in abstracts do not highlight purposes similar to those of the document
- Work availability: exclusion of research papers that are not openly accessible or accessible trough university.
- Short paper: exclusion of research papers that are short versions of other papers.
- Repeated Papers: exclusion of research papers that have already been selected or excluded in other queries or snowballing step.

Selection and Exclusion criteria are applied in every stage of snowballing.

2.4 Data extraction

Once that the papers are chosen, we can begin the data extraction stage. This must be done taking in account the research questions stated at the beginning of the SLR. Documents are scrutinised so we can say with certainty whether a paper will be useful or not. Useful data is stored and organised, while not-useful paper are permanently excluded. [Woh14]

2.5 Clustering and Discussion

After data extraction phase, it will be necessary to cluster the data extracted from the papers. The clusters are made in such a way that each of them can answer one of the research questions. Once the information has been grouped, we can organise them to start writing a discussion that answers the research question to which the answer is related.

2.6 Quality Assessment

As the document is intended to be read by academics interested in ILP, in addition to revealing the research methodology, it was necessary to provide an assessment of the *quality* of the primary studies used. The guidelines must be chosen according to the needs of the study, but they must provide a detailed and objective analysis, in order to make any analysis of differences in results in other studies clear, assess the relevance of individual studies when discussing research questions, and guide the interpretation of results and recommendations for further research. The quality of a primary study is defined by a low number of biases, the validity of data and the generalization and application of observation in other studies [BK07]. For a reliable assessment, defined and objective criteria were used. The parameters chosen to obtain an objective overview of the articles are:

- Number of Citations: citations are evaluated, because we consider that the more they have, the more the overview offered will be elaborate and complete. Mention their own paper is rated less than normal, while mentioning paper already in start set is better rated because they are considered more relevant to our purpose.
- Paper Impression: it is first considered whether the article is a theoretical introduction or presents experimental or innovative features. In the first case the more the paper is cited the more the rate is good. In the second case we evaluate positively the presence of pseudo-code, libraries or API.
- Publisher Impression: the average annual number of citations of articles published in the last two years in a given journal was considered.
- Benefits in discussion: the number of questions to which the article was able to provide useful information for the drafting of the discussion was evaluated.

Results must be normalised due to to have a better evaluation on different papers.

3 Results

Here we will show the results of the procedures described in previous section (sec. 2)

3.1 Building Primary Research Set

Once research process is planned, it is not difficult to do so, but it is demanding enough. To carry it out effectively it will be divided into 4 phases:

1. Query-based papers research 2.2;
2. Inclusion and exclusion of articles based on the chosen criteria 2.3;
3. Application of the snowballing technique and use of inclusion and exclusion criteria;
4. Data extraction and refining 2.4;

3.1.1 Query-based papers research

After a little research, the field of research to which ILP belonged and the technologies to be involved in the study stood out. ILP is basically a set of algorithms used for the composition of logical rules and proved to be particularly useful in machine learning during the learning phase. This reasoning led to the composition of the queries in the 2.2 section. The result of this first phase is:

Query	**Google Scholar Results**	**Result Evaluated**
Query 1	14,400 (circa)	the first 100
Query 2	40,300 (circa)	the first 100
Query 3	28,500 (circa)	the first 150

Table 1: Results of search process

3.1.2 Inclusion and exclusion of articles

In this phase, each paper obtained from the queries will be analysed, using criteria defined in 2.3. These filters are in addition to those imposed on the search engine. The result is:

Query	**Examined Papers**	**Candidate Papers**	**Promoted Papers**
Query 1	100	17	6
Query 2	100	19	2
Query 3	150	12	4

Table 2: Results of application of inclusion/esclusion criteria

3.1.3 Snowballing

Once the candidate papers for the start set have been identified and evaluated through selection criteria, the remaining papers, which make up the actual start set, can undergo the snowballing process. The papers that will be submitted to the next phase of snowballing are:

[Dar20]: A. Darwish, **Reconciling between Symbolic and Connectionist AI**, Research Gate, 2020.

[Bib19]: M. Kastratia & M. Bibaa, **Statistical Relational Learning: A state of the art review**, Journal of Engineering Technology and Applied Sciences, 2019.

[Kli15]: J. Furnkranz & T. Kliegr, **A Brief Overview of Rule Learning**, Springer International Publishing Switzerland, 2015.

[SG15]: S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid & B. Zorn, **Inductive Programming Meets The Real World**, Communications Of The Acm, 2015.

[A.K15]: S. Tyagi & A.Kowcika, **Best Practices Generation for Storage Area Network: A Review**, Proceedings of the International Conference : "Computational Systems for Health & Sustainability", 2015.

[FR14]: F. Riguzzi, E. Bellodi & R. Zese, **A history of Probabilistic Inductive Logic Programming**, Frontiers in robotics and AI, 2014.

[Rus15]: S. Russell, **Unifying Logic and Probability**, Communications Of The Acm, 2015.

[JL15]: Johnson-Laird, S. S. Khemlani, & G. P. Goodwin, **Logic, probability, and human reasoning**, Elsevier Ltd., 2015.

[ML20b]: M. Law, A. Russo, E. Bertino, K. Broda & J. Lobo, **FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria**, The Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI-20), 2020.

[QZ14]: Q. Zeng, J. M. Patel & D. Page **QuickFOIL: Scalable Inductive Logic Programming**, Proceedings of the VLDB Endowment, Vol. 8, No. 3, 2014.

[DC12]: D. Corapi, A. Russo, & E. Lupu, **Inductive Logic Programming in Answer Set Programming**, ILP 2011, LNAI 7207, pp. 91–97, 2012.

[Mug17]: S. H. Muggleton, **Meta-Interpretive Learning: Achievements and Challenges**, RuleML+RR 2017, LNCS 10364, pp. 1–6, 2017.

Snowballing can now be started from the step backward. It starts by analysing the citation list of each article in the initial set, excluding documents that not meet any selection criteria. Subsequently, all documents that have already been examined and excluded previously are deleted from the list. Only remaining documents are considered candidate papers. To examine a candidate paper, first the abstract is read and then other parts of the paper that are considered important are read until a final decision can be made on whether to include or exclude the paper.
Regarding step forward, essentially the same procedure as for the step backward is performed, but the papers are chosen from the list of papers that mention those in the starter set. The new papers added to the set after snowballing are:

Documents in Start Set	**Documents Snowballing**	**Documents in Final Set**
12	26	38

Table 3: Results of application of snowballing

[Mug91a]: S. Muggleton, **Inductive Logic Programming**, New Generation Computing, vol. 8, OHMSHA, LTD, pp 295-318, 1991.

[Rea94]: S. Muggleton & L. De Readt, **INDUCTIVE LOGIC PROGRAMMING: THEORY AND METHODS**, LOGIC PROGRAMMING, vol 19, n 20, pp 629-679, Elsevier Science Inc., 1994.

[Ker08]: L. De Readt & K. Kersting, **Probabilistic Inductive Logic Programming**, Probabilistic ILP 2007, vol 4911, pp. 1–27, Springer-Verlag, 2008.

[Kit10]: E. Kitzelmann, **Inductive Programming: A Survey of Program Synthesis Techniques**, AAIP 2009, vol 5812, pp. 50–73, 2010.

[SM10]: S. Muggleton, J. Santos & A. Tamaddoni-Nezhad, **ProGolem: A System Based on Relative Minimal Generalisation**, ILP 2009, vol 5989, pp. 131–148, 2010.

[SM11]: S. Muggleton, L. De Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue & A. Srinivasan, **ILP turns 20**, Springerlink.com, 2011.

[Rig12]: E. Bellodi & F. Riguzzi, **Learning the Structure of Probabilistic Logic Programs**, ILP 2011, LNAI 7207, pp. 61–75, 2012.

[MD19]: M. Das, D. Singh Dhami, G. Kunapuli, K. Kersting & S. Natarajan, **Fast Relational Probabilistic Inference and Learning: Approximate Counting via Hypergraphs**, Association for the Advancement of Artificial Intelligence, 2019.

[Kin99]: A. Srinivasan & R. D. King, **Feature construction with Inductive Logic Programming: A Study of Quantitative Predictions of Biological Activity Aided by Structural Attributes**, Data Mining and Knowledge Discovery 3, pp 37–57, 1999.

[Sta94]: I. Stahl, **On the Utility of Predicate Invention in Inductive Logic Programming**, 1994.

[Zho17]: W. Dai & Z. Zhou, **Combining Logical Abduction and Statistical Induction: Discovering Written Primitives with Human Knowledge**, Association for the Advancement of Artificial Intelligence, 2017.

[NSO16]: N. Skovgaard-Olsen, H. Singmann & K. C. Klauer, **The relevance effect and conditionals**, Elsevier B.V., 2016.

[Ben18]: P. Schüller & M. Benz, **Best-effort inductive logic programming via fine-grained cost-based hypothesis generation**, Mach Learn, 2018.

[Mug19]: A. Cropper & S. Muggleton, **Learning efficient logic programs**, Mach Learn 108, pp. 1063–1083, 2019.

[AC22]: A. Cropper, S. Dumančić, R. Evans & S. Muggleton, **Inductive logic programming at 30**, Machine Learning (2022) 111, pp. 147–172, 2022.

[CH98]: C. H.Bryant, S. H. Muggleton, C. D. Page & M. J. E. Sternberg, **Combining Active Learning with Inductive Logic Programming to close the loop in Machine Learning**, 1998.

[Mug99]: S. Muggleton, **Inductive Logic Programming: Issues, results and the challenge of Learning Language in Logic**, Artificial Intelligence 114, pp. 283–296, 1999.

[Rae01]: K. Kersting & L. De Raedt, **Towards Combining Inductive Logic Programming with Bayesian Networks**, ILP 2001, LNAI 2157, pp. 118–131, 2001.

[Ker06]: K. Kersting, **An inductive logic programming approach to statistical relational learning**, AI Communications 19, pp. 389–390, 2006.

[Kin04]: R. D. King, **Applying Inductive Logic Programming to Predicting Gene Function**, AI Magazine Volume 25 n. 1, 2004.

[RDK96]: R. D. King, S. Muggleton, A. Srinivasani, & M. J. E. Sternberg, **Structure-activity relationships derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity by inductive logic programming**, Proc. Natl. Acad. Sci. USA Vol. 93, pp. 438-442, 1996.

[Mug95]: S. Muggleton, **Inverse Entailment and Progol**, New GenerationComputing, 13, pp. 245-286, 1995.

[OR03]: O. Ray, K. Broda & A. Russo, **Hybrid Abductive Inductive Learning: A Generalisation of Progol**, ILP 2003, LNAI 2835, pp. 311–328, 2003.

[CJ06]: J. R. Quinlan & R. M. Cameron-Jones, **FOIL: A Midterm Report**, ILP 2006, 2006.

[ML20a]: M. Law, A. Russo & K. Broda, **The ILASP System for Inductive Learning of Answer Set Programs**, ILASP Limited, 2020.

[AC20]: A. Cropper, R. Morel & S. Muggleton, **Learning higher-order logic programs**, Machine Learning (2020) 109, pp. 1289–1322, 2020.

3.1.4 Data Extraction and Refining

In the extraction phase, all selected documents are studied and thoroughly analysed. At this stage, the aim is to extract the knowledge needed to answer one or more research questions. Moreover, if at this stage a document does not add or confirm information relevant to the review, it can be eliminated from the set.
In extraction stage some documents were found to be superfluous and deleted.

Paper in Final Set	**Paper Deleted in Refining**	**Paper Used**
38	13	25

Table 4: Results of Refining

Papers were excluded because they were assessed as irrelevant for SLR purposes, in fact, they had overviews that were too general or too focused on certain topics or already or because the topic had already been dealt with in more detail in another paper. The excluded paper are: [FR14, Rus15, JL15, Kit10, Rig12, MD19, Kin99, NSO16, CH98, Rae01, Ker06, Kin04, RDK96].

3.2 Discussion

Once extracting and refining data is completed, the information that we have acquired from all papers in set will be collected, summarised and organised to answer research questions. The research question is the topic of discussion and its answer is the argumentation of that discussion.

3.3 What is ILP?

Paper: [Bib19, A.K15, ML20b, QZ14, DC12, Mug17, Mug91a, Rea94, Ker08, SM11, AC22, Mug99, ML20a]
Inductive logic programming ILP is a form of machine learning (ML) [Mug91a, Rea94]. As with other forms of ML, the goal is to induce a hypothesis that generalises training examples. However, whereas most forms of ML use vectors/tensors to represent data, ILP uses logic programs (sets of logical rules). Moreover, whereas most forms of ML learn functions, ILP learns relations [AC22].
The language that ILP use is first order logic. It is a formal language used for mechanically manage statements and reasoning that involve logical connectors, relationship and quantifier. Data represented by logic programs allow to ILP system to learn with complex relational information, such as constraints about causal networks, the axioms of the event calculus when learning to recognise events, and using a theory of light to understand images. Moreover, because hypotheses are symbolic, hypotheses can be added to background, and thus ILP systems naturally support lifelong and transfer learning ([Bib19, SM11]).
As pointed out by various authors in the various cited papers, due to the symbolic nature of logic programs, ILP can reason about hypotheses, which allows it to learn optimal programs, such as minimal time-complexity programs and secure access control policies and because of the expressivity of logic programs, ILP can learn complex relational theories, event calculus theory, Petri net and general algorithms. Regardless of the different ILP approach or algorithm that is analysed, there are certain characteristics that define the theoretical foundations of the technology, as explained by Muggleton in [Mug91a], [Rea94] and [Mug99] that are:

- The logic program that provides a **consistence background knowledge**, B, where the relationship $\exists e^+ \in E^+ | B \nvDash e^+$ is true.
- the set of **positive examples**, E^+, that is part of **training set**.
- the set of **negative examples**, E^-, that is part of **training set**.
- the set of program of **well formed hypothesis space**, $\mathcal{H}$. It defines the research space.
- consider H **hypothesis** o **target program**, so B, H is the logic program obtained by adding the clauses of H to B.
- syntactic distortion, **bias** of the language.

we can write the induction problem as:
Give a consistence examples set E and a consistence background knowledge, find a hypothesis H such as:

$B \cup H \vdash e$ (H cover e o $cover(e, H \cup B)$)
$\forall e^+ \in E^+, B, H \vDash e^+$ (H è **completeness**)
$\forall e^- \in E^-, B, H \nvDash e^-$ (H è **consistency**)

3.3.1 The Language

Now we introduce the basics of a logic language, using the papers [Bib19, ML20b, Rea94, Ker08, SM11, AC22, ML20a] as references:

- Constants: individual entities of the domain. They can also be considered as functions with arity 0. They are known a priori.
- Variables: entities of the domain not known a priori.
- Functions: uniquely identify an object of the domain through a relationship between other n objects. Functions in logic do not presuppose any concept of evaluation.
- Predicates: they identify a generic relationship between n entities of the domain. It is not certain that the identified relationship is true.

Through the definition of main component of a logic language we can define:

- Alphabet: set of symbols of constants, variables, functions and predicates.
- Term: constants, variables and n-ary functions are considered terms.
- Atoms: atoms, or atomic formulas, are considered to be the application of a predicate symbol with n terms: $p(t_1, ..., t_n)$
- Expression or formula: sequence of symbols belonging to the alphabet.
- Well Formed Formulas (wff): syntactically correct sentences of the language. They are obtained by combining atomic formulas, using connectives and quantifiers.
- Definite clauses: they are formulas of form

  $$A : -B_1, ..., B_m$$

  where A and B_i are atoms and variables are universally quantified. For example, the clause c

  $$c \equiv grandparent(X, Y) : -parent(X, Z), parent(Z, Y)$$

  and we can read X is grandparent of Y if X is parent of Z and Z is parent of Y. We can call $grandparent(X, Y)$ *head(c)* that is head of clause c, and $parent(X, Z)$, $parent(Z, Y)$ *body(c)* that is body of clause c.

Moreover, we can consider as *fact* the clauses with an empty *body*, and a set of clauses defines a logic program.
The function $Var(E)$ specify the set of variables in a atom or in a clause E. If $Var(E) = \emptyset$, regardless of what E is, it is defined *ground*.
A clause c is *range-restricted* if all the variables in head are also in the body of the clause, in formula $Var(head(c)) \subseteq Var(body(c))$.

A *Subsumption*$\theta = \{V_1/t_1, ..., V_n/t_n\}$ is an assignment of terms t_i to a variable V_i. Applying the θ-substitution to a term, atom o clause results in an instance of term, atom o clause $e\theta$ in which all the occurrences of variables V_i are simultaneously replace by the term t_i. So the definition becomes:
A clause c_1 θ *subsumes a clause* c_2 *if and only if there exists a substitution* θ *such that* $c_1\theta \subseteq c_2$. c_1 *is a generalisation of* c_2, *and* c_2 *is a specialisation of* c_1, *under* θ *subsumption.*
For example $grandparent(X, Y) : -parent(X, Z), parent(Z, Y)$ θ subsumes
$grandparent(X, ann) : -parent(X, Z), parent(Z, ann)$ con $\theta = \{Y = ann\}$.
The *Herbrand base* is another main component of logic programming. The Herbrand base of a logic program P is written as $\mathcal{B}(P)$, it is a set of *atomic ground* and it is composed with symbols of predicates, constants and functions in the alphabet of P. An example of Herbrand base is:
$\mathcal{B}$(grandparent) = { parent(ann, ann), parent(jeff, jeff), parent(paul, paul),
parent(ann, jeff), parent(jeff, ann), ...,
grandparent(ann, ann), grandparent(jeff, jeff), ...}
Herbrand interpretation of a logic program P is a subset of $\mathcal{B}(P)$. Herbrand interpretation I is a *model* $\mathcal{M}(P)$ if and only if all θ -substitutions such as $body(c)\theta \subseteq I$, and $head(c)\theta \in I$ are true. The Herbrand least model $LH(P)$, and it is what make up the semantic of a logic program P, the element of this set are all facts $f \in \mathcal{B}(P)$ such us P logically implies f , that is written as $P \vDash f$. All ground atoms in least Herbrand model are true.

3.3.2 The Semantic

Semantics is a topic explored in depth by Muggleton and De Raedt in [**?**], where they explain the different types of semantics and the consequences they have on language and hypotheses by means of a mathematical demonstration.
Depending on how the relationship between the logic elements used in inductive inference change, we describe two different semantics for ILP: the *normal semantics (monotonic s.)* and *non-monotonic semantics*. We also need to consider the presence of a syntactic bias, that aside from which semantic will be used, it will define the set of well-formed hypotheses and thus constitutes a parameter of any ILP task. Because the use of a syntactic bias is omni-present in ILP, presenting the conditions of semantics, we will not always write explicitly that we assume the hypotheses are well-formed with regard to this bias.

Normal semantic
Normal semantic is the general setting for ILP and allow examples, background theory, and hypotheses to be any well-formed logical formula. In normal semantic settings ILP problem can be write as:
Background knowledge B *and some examples* E *are given. The examples are composed by positive and negative example,* $E = E^+ \cup E^-$. *The goal is find the hypotheses such that the following condition are valid:*

Prior Satisfiability: $B \cup E^- \not\models \square$
Posterior Satisfiability: $B \cup H \cup E^- \not\models \square$
Prior Necessity: $B \not\models E^+$
Posterior Sufficiency: $B \cup H \models E^+$

Posterior Sufficiency correspond with completeness propriety, and Posterior Satisfiability correspond with consistency propriety. Both propriety are previously described. The semantic of logic program is based on the concepts of Herbrand Universe, Herbrand Base, Herbrand Interpretation. The three concepts are based on a vocabulary that contains all the constants, the functions and the symbols of predicates that are in the logic program. Herbrand's concepts are the best way to describe ILP settings. This definite setting is simpler than the general setting because a definite clause theory T has a unique minimal Herbrand model $\mathcal{M}^+(T)$, and any logical formula is either true or false in this least model. The new condition are:

Prior Satisfiability: $\forall e \in E^-$ *false in* $\mathcal{M}^+(B)$
Posterior Satisfiability: $\forall e \in E^-$ *false in* $\mathcal{M}^+(B \cup H)$
Prior Necessity: $\exists e \in E^+$ *false in* $\mathcal{M}^+(B)$
Posterior Sufficiency: $\forall e \in E^+$ *true in* $\mathcal{M}^+(B \cup H)$

The special case of the definite semantics, where the evidence is restricted to true and false ground facts (examples), will be called the example setting. Notice that the example setting is equivalent to the normal semantics, where B and H are definite clauses and E is a set of ground unit clauses. Example setting is the main setting of ILP. It is used by a large number of ILP systems.

Non-monotonic semantic
In non-monotonic semantic ILP setting, the background theory is a set of defined clauses, examples set is empty and the hypotheses are set of clauses expressed using the same alphabet as the background. Example set is empty due to the fact that positive examples are considered part of background theory, and negative example is derived implicitly, by making a kind of closed world assumption realised by taking the least Herbrand model. To define non-monotonic semantic we need of three requirement:

Validity: $\forall h \in H$ *true in* $\mathcal{M}^+(B)$
Completeness: *if general clause* g *is true in* $\mathcal{M}^+(B)$, *then* $H \models g$
Minimality: $\exists e \in E^+$ *false in* $\mathcal{M}^+(B)$

Validity assure that all clauses belonging to hypothesis are contained in B. Completeness assure that all information valid in B can be represented in hypothesis. This requirement should also be understood with regard to a given syntactic bias, which determines the set of well-formed hypotheses. The Minimality requirement aims at deriving non-redundant hypotheses.

The nonmonotonic setting hypothesises only properties that hold in the database. Therefore, the nonmonotonic semantics realises induction by deduction. The induction principle of the nonmonotonic setting states that the hypothesis H, which is, in a sense, deduced from the set of observed examples E and the background theory B holds for all possible sets of examples. This produces generalization beyond the observations. As a consequence, properties derived in the nonmonotonic setting are more conservative than those derived in the normal setting. The differences between the two settings are related to the closed world assumption. In ILP system may happen that, as in the case of non-monotone semantics, the negative example set E^- does not come explicitly defined. This is because through the assumption of a closed world it is possible to express $E^- = \mathcal{M}^-(B \cup E^+)$ and this means that E^- is a set consisting of all the ground clauses of $\mathcal{B}(B \cup E^+)$ not in the model $\mathcal{M}^+(B \cup E^+)$.

3.3.3 Relations of Generalities

Logic systems learn a theory by execute a research in the space of clauses order by relations of generalities. The relation of generality between two clauses can be defined as:
• *Formula A is more generic then Formula B if and only if* $A \vDash B$ *and* $B \nvDash A$.
From this definition we can deduce the definitions of generalization and specialisation, that are:
• *A specialisation operator maps a conjunction of clauses G onto a set of maximal specialisations of S. A maximal specialisation S of G is a specialisation of G such that G is not a specialisation of S, and there is no specialisation* S' *of G such that S is a specialisation of* S'.
•*A generalization operator maps a conjunction of clauses S onto a set of minimal generalisations of S. A minimal generalization G of S is a generalization of S such that S is not a generalization of G, and there is no generalization* G' *of S such that G is a generalization of* G'.
These relation are essential in ILP, for this reason it becomes really important to define what *subsumption* is that is fundamental in ILP systems. Subsumption is explained in the paragraph "The Language". Through subsumption, as demonstrated by Muggleton and De Raedt in [**?**], it is possible to express the logic proprieties of the language, such as:

> **Implication:** if $c_1\theta$ -subsumes c_2 then $c_1 \vDash c_2$. The opposite does not hold for self-recursive clauses, furthermore if $\exists c_1, c_2$ such that $c_1 \vDash c_2$ it does not a sufficient condition to assert that $c_1\theta$ -subsumes c_2.
>
> **Infinite Descending Chains:** There exist infinite descending chains, e.g.:
> $h(X_1, X_2) \leftarrow$
> $h(X_1, X_2) \leftarrow p(X_1, X_2)$
> $h(X_1, X_2) \leftarrow p(X_1, X_2), p(X_2, X_3)$
> ... This series is bounded from below by $h(X, X) \leftarrow p(X, X)$.
>
> **Infinite Ascending Chains:** There exist rather complicated infinite ascending chains.

Equivalence: There exist different clauses that are equivalent under θ-subsumption. When this append it is true both $c_1\theta$ -subsumes c_2 then $c_1 \vDash c_2$, and $c_2\theta$ -subsumes c_1 then $c_2 \vDash c_1$. ILP systems should generate at most one clause for each equivalence class.

Reduction: To overcome the equivalence problems, Plotkin has defined the clauses equivalence classes and he has shown that there is a unique representative of each clause, which he named the *reduced* clause. The reduced clause r of a clause c is a minimal subset of literals of c such that r is equivalent to c. An algorithm to reduce clauses follows from this. ILP systems can get around the problem of equivalent clauses when working with reduced clauses only.

Lattice: The set of reduced clauses form a lattice, this means that each pair of clauses is part of a lattice, so each pair of clauses has a least upper bound (lub) and a greatest lower bound (glb). This is one of the most important proprieties in ILP.

3.3.4 Learning

Papers [Bib19] and [Ker08] point out that an ILP system mainly uses three type of learning:
Learning from Entailment: this is the most popular ILP setting and it is addressed by a wide variety of well-known ILP systems such as FOIL and PROGOL. When learning from entailment, the examples are definite clauses and a hypothesis H covers an example e with respect to the background theory B if and only if $B \cup H \vDash e$, so each model of $B \cup H$ is also a model of e. In many well-known systems, the examples are required to be ground facts, a special form of clauses.
Learning from Interpretations: The learning from interpretations setting upgrades Boolean concept-learning in computational learning theory. When learning from interpretations, the examples are Herbrand interpretations, and a hypothesis H covers an example e with respect to the background theory B if and only if e is a model of $B \cup H$. Recall that Herbrand interpretations are sets of true ground facts and they completely describe a possible situation. The fundamental difference between learning from interpretations and learning from entailment is that interpretations contain much more information and even more complete. Indeed, when learning from entailment, an example may consist of a single fact, whereas when learning from interpretations, all facts that hold in the example are known. Therefore, learning from interpretations is typically easier and computationally more tractable than learning from entailment.
Learning from Proofs: Because learning by entailment and interpretations occupy extreme positions with respect to the information the examples carry, it is interesting to investigate intermediate positions. Shapiro's Model Inference System (MIS) fits perfectly into the framework of learning by entailment, where examples are made. However, to deal with missing information, Shapiro employs a clever strategy: MIS queries the users for missing information by asking them about the truth-value of facts. The answers to these queries allow MIS to reconstruct the trace or the proof of the positive examples.

Inspired by Shapiro's MIS we can define the learning from proofs setting. When learning from proofs, the examples are ground proof-trees and an example e is covered by a hypothesis H with respect to the background theory B if and only if e is a proof-tree for $H \cup B$.
At this point, there exist various possible forms of proof-trees. We will assume that the proof-tree is given in the form of a ground tree in which the nodes contain ground atoms. More formally, a tree t is a proof-tree for a logic program T if and only if t is a rooted tree where for every node $n \in t$ with $children(n)$ satisfies the property that there exists a substitution θ and a clause $c \in T$ such that $n = head(c)\theta$ and $children(n) = body(c)\theta$. Proof-trees contain, as interpretation, a lot of information. Indeed, they contain instances of the clauses that were used in the proofs. Therefore, it may be hard for the user to provide this type of examples. Even though this is generally true, there exist specific situations for which this is feasible.

3.3.5 Datalog

ILP systems have traditionally induced definite and normal logic programs, typically represented as Prolog programs. A recent development has been to use different hypothesis representations. One of new hypothesis representation is Datalog.
Datalog is a syntactical subset of Prolog that does not allow complex terms as arguments of predicates and imposes restrictions on the use of negation. It is a truly declarative language, whereas in Prolog reordering clauses can change the program. Moreover, Datalog query is guaranteed to terminate, though this guarantee is at the expense of not being a Turing-complete language, which Prolog is [AC22]. The general motivation for reducing the expressivity of the representation language from Prolog to Datalog is to allow the problem to be encoded as a satisfiability problem, particularly to leverage recent developments in SAT and SMT.

3.3.6 Higher Order

A higher order function is such if it accepts one or more functions as arguments and/or returns a function. Thus in predicate logic, the definition of higher order becomes a set of Horn higher order clauses in which the leading atoms have the same predicate symbol [AC20]. Higher order can prove fundamental for learning recursive first-order programs. Let us imagine learning a drop lasts programme, which removes the last element of each sublist in a list, e.g. $(alice, bob, carol) \rightarrow (alic, bo, caro)$ [AC22], normally an ILP system such as *Metagol* would obtain the program:

$f(A,B) :- empty(A), empty(B).$
$f(A,B) :- head(A,C), tail(A,D), head(B,E), tail(B,F), f1(C,E), f(D,F).$
$f1(A,B) :- reverse(A,C), tail(C,D), reverse(B,D).$

Although semantically correct, the program is verbose. This is where higher-order comes in, though initially it was only used to represent background knowledge it was later introduced following studies of inductive generalisation in higher-order logic using a restricted lambda calculus formula. Consequently, we will have higher-order rules where predicate

symbols can be used as terms. Continuing in our example, we will discuss $Metagol_{HO}$, a higher-order extension of *Metagol*, which can learn the following program:

$f(A,B) :- map(A,B,f1).$
$f1(A,B) :- reverse(A,C), tail(C,D), reverse(B,D).$

Even if ILP systems cannot define new higher-order predicate symbols, they can still provide for their use. In fact, to learn this higher-order programme, one must invent the predicate symbol f1, which is used twice in the programme: as a term in the literal $map(A,B,f1)$ and as a predicate symbol in the literal $f1(A,B)$. In [AC22] Croppere et al. emphasise that compared to the first-order programme, this higher-order programme is smaller because it uses $map/3$ (default in the background) to abstract from list manipulation and to avoid the need to learn an explicitly recursive programme (recursion is implicit in $map/3$). $Metagol_{HO}$ has been shown to reduce sample complexity and learning time and improve prediction accuracy.

3.3.7 Probabilistic ILP Semantics

As previously specified, ILP works through the use of relational data, however it is unable to handle uncertainties. Having uncertainties forces ILP to extend its semantics with probabilistic settings. This is how PILP (Probabilistic Inductive Logic Programming) was born. PILP combines ILP principles with statistical learning in the most natural way.
De Raedt and Kersting in [Ker08] and Kastrati and Biba in [Bib19] focus specifically on the theoretical foundations of PILP and analyse the main methodologies used in the probabilistic settings, highlighting the differences between ILP and PILP.
The PILP settings are based on those of the ILP but include the addition of definitions to manipulate the probability such as:

- $\mathcal{X}$ random variable with finite domain, $domain(\mathcal{X}) = \{x_1, x_2, ..., x_n\}$.
- $P(\mathcal{X})$ denote the probability distribution over the domain $\mathcal{X}$.
- $P(x_i)$ denote the probability that the random variable $\mathcal{X}$ takes the value x_i, where $x_i \in domain(\mathcal{X})$.
- The distribution $P(X_1, X_2, ..., X_n)$ over a set of random variables with $n > 1$ is called *joint probability distributions.*
- The *conditional probability* is defined as $P(X|Y) = \frac{P(X,Y)}{P(Y)}$ if $P(Y) > 0$ that using *law of Bayes* becomes $P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}$. If X and Y are conditionally independent variables, the law of Bayes is no longer valid. It therefore becomes possible to give a third variable Z if and only if the formula $P(X,Y|Z) = P(X|Z)P(Y|Z)$ is valid. This formula is really important in PILP.
- Clauses are annotated with probabilistic information such as conditional probabilities. (First real difference with ILP).

- The covers relation becomes probabilistic. Probabilistic covers relation becomes $cover(e, H \cup B) = P(e|H, B)$. (Second real difference with ILP).
- The covers relation and the law of Bayes allow us to redefine ILP problem. Therefore, the learning problem with Probabilistic ILP becomes: *Given a probabilistic-logical language $\mathcal{L}_{\mathcal{H}}$ and a set E of examples over some language $\mathcal{L}_E$, find the hypothesis H^* in $\mathcal{L}_H$ that maximises $P(e|H^*, B)$.* Under the usual identically and independently distributed variables assumption, this results in the maximisation of

$$P(E|H^*, B) = \prod_{e \in E} P(e|H^*, B) = \prod_{e \in E} covers(e, H^*, B)$$

Similar to the ILP learning problem, the language $\mathcal{L}_E$, that represent the examples together with the probabilistic covers relation, determines different learning setting. In ILP, this lead to learning from interpretations, from proofs, and from entailment, so it should be no surprise that this very same distinction also applies to probabilistic knowledge representation formalism. Indeed, Bayesian networks essentially define a probability distribution over interpretations or possible worlds, and stochastic grammars define a distribution over proofs, derivations or traces. Therefore we can define three settings for probabilistic ILP Learning.

Probabilistic Learning from Interpretations
In order to integrate probabilities in the learning from interpretations setting, we need to find a way to assign probabilities to interpretations covered by an annotated logic program. In the past decade, this issue has received a lot of attention and various different approaches have been developed. The most popular propositional frameworks are Bayesian network and Markov networks. Later on, these propositional frameworks have been extended to the relational case such probabilistic-logic programs , probabilistic relational models , relational Bayesian networks, and Bayesian logic programs.
A **Bayesian network** is an augmented, directed acyclic graph, where each node corresponds to a random variable X_i and each edge indicates a direct influence among the random variables. It represents the joint probability distribution $P(X_1, ..., X_n)$. The influence is quantified with a conditional probability distribution $cpd(X_i)$ associated to each node X_i. It is defined in terms of the parents of the node X, which we denote by $Pa(X_i)$, and specifies $cpd(X_i) = P(X_i|Pa(X_i))$. The Bayesian network thus has two components: a qualitative one, i.e. the directed acyclic graph, and a quantitative one, i.e. the conditional probability distributions. Together they specify the joint probability distribution.
Bayesian Logic Programs are a language used to integrate definite logic programs with BNs that results probability distributions over first-order interpretations. The main use of BLPs is to create an one-to-one mapping between ground atoms and random variables, and between the immediate consequence operator and the dependency relation. In this way, BLPs combine the advantages of both definite clause logic and BNs. ILP learning from interpretation can be combined with qualitative component, quantitative component or both.

Relational Markov networks are undirected graphical models that extend the framework of MNs to relational domains. Undirected models (MNs) address two limitations of the directed models. First, MNs are undirected models so there are free of the problem of cycles. Second, undirected models are more appropriate for discriminative training, where it is optimised the conditional likelihood of the labels for given features, that generally helps to improve classification accuracy. It is showed how these models can be trained to be more efficient, and also they illustrated how to use approximate probabilistic inference over the learned model for collective classification and link prediction. A relational Markov network $M = (\mathcal{C}, \phi)$ is defined by a set of clique templates C and corresponding potentials $\phi = \{\phi_C\}_{C \in \mathcal{C}}$ in order to define a conditional distribution as shown:

$$P(I.y|I.x, I.r) = \frac{1}{Z(I.x, I.r)} \prod_{C \in \mathcal{C}} \prod_{c \in C(I)} \phi_c(I.x_c, I.y_c)$$

where $Z(I.x, I.r)$ represents the normalising partition function

$$Z(I.x, I.r) = \sum_{I.y} \prod_{C \in \mathcal{C}} \prod_{c \in C(I)} \phi_c(I.x_c, I.y_c)$$

Probabilistic Proofs
To define probabilities on proofs, ICL, PRISMs, and stochastic logic programs attach probabilities to facts (respectively clauses) and treat them as stochastic choices within resolution. Probabilistic learning from proofs can be illustrated using stochastic logic programs. Stochastic logic programs are inspired on stochastic context free grammars. The analogy between context free grammars and logic programs is that:

- Grammar rules correspond to definite clauses;
- Sentences (or strings) to atoms;
- Productions to derivations.

Furthermore, in stochastic context-free grammars, the rules are annotated with probability labels in such a way that the sum of the probabilities associated to the rules defining a non-terminal is 1.0. For this reason it was introduce a couple clause-probability where each clause c has an associated probability label p_c such that the sum of the probabilities associated to the rules defining any predicate is 1.0.
This framework allows ones to assign probabilities to proofs for a given predicate q given a stochastic logic program $H \cup B$ in the following manner. Let D_q denote the set of all possible ground proofs for atoms over the predicate q. For simplicity reasons, it will be assumed that there is a finite number of such proofs and that all proofs are finite. Now associate to each proof $tq \in Dq$ the probability

$$v_t = \prod_c p_c^{n_{c,t}}$$

where the product ranges over all clauses c and $n_{c,t}$ denotes the number of times clause c has been used in the proof t_q. For stochastic context free grammars, the values v_t

correspond to the probabilities of the production. However, the difference between context free grammars and logic programs is that in grammars two rules of the form $n \rightarrow q, n_1, ..., n_m$ and $q \rightarrow q_1, ..., q_k$ that became $n \rightarrow q_1, ..., q_k, n_1, ..., n_m$ whereas resolution may fail due to unification. Therefore, the probability of a proof tree t in Dq, i.e., a successful derivation is

$$P(t|H, B) = \frac{v_t}{\sum_{s \in D_q} v_s}$$

The probability of a ground atom a is then defined as the sum of all the probabilities of all the proofs for that ground atom.

$$P(a|H, B) = \sum_{s \in D_q} v_s$$

where s is a proof for a.
Motivated by $P(t|H, B)$ formula, we can learn them from proofs. This makes structure learning for stochastic logic programs relatively easy, because proofs carry a lot information about the structure of the underlying stochastic logic program. Furthermore, the learning setting can be considered as an extension of the work on learning stochastic grammars from proof-banks. It should therefore also be applicable to learning unification based grammars. On the other hand, we can use $P(t|H, B)$ formula as covers relation and, hence, employ the learning from entailment setting. Here, the examples are ground atoms entailed by the target stochastic logic program. Learning stochastic logic programs from atoms only is much harder than learning them from proofs because atoms carry much less information than proofs.

Probabilistic Learning from Entailment
In order to integrate probabilities in the entailment setting, an ILP system needs to find a way to assign probabilities to clauses that are entailed by an annotated logic program. Since most ILP systems working under entailment employ ground facts for a single predicate as examples, and the authors are unaware of any existing probabilistic ILP formalism that implement a probabilistic covers relation for definite clauses as examples in general, we will restrict our attention to assign probabilities to facts for a single predicate. It remains an open question as how to formulate more general frameworks for working with entailment.
More formally, let us annotate a logic program H consisting of a set of clauses of the form$p \leftarrow bi$, where p is an atom of the form $p(V_1, ..., V_n)$ with the V_i different variables, and the b_i are different bodies of clauses. Furthermore, we associate to each clause in H the probability values $P(b_i|p\theta)$; they constitute the conditional probability distribution that for a random substitution θ for which $p\theta$ is ground and true , the query $b_i\theta$ succeeds in the knowledge base B. Furthermore, we assume the prior probability of p is given as $P(p\theta)$, it denotes the probability that for a random substitution θ, $p\theta$ is true. This can then be used to define the covers relation $P(p\theta|H, B)$ as follows:

$$P(p\theta|H, B) = P(p\theta|b_1\theta, ..., b_k\theta) = \frac{P(b_1\theta, ..., b_k\theta|p\theta)P(p\theta)}{P(b_1\theta, ..., b_k\theta)}$$

And with Bayes assumption became

$$P(p\theta|H,B) = \frac{\prod_i P(b_i\theta|p\theta)P(p\theta)}{P(b_1\theta,...,b_k\theta)}$$

3.3.8 Declarative Bias

This overview on ILP allows to define the ILP problem in a more formal way.
Given a set of positive and negative examples E^+ and E^- over some language L_E , a background theory B, in the form of a set of definite clauses, a hypothesis language L_H , which specifies the clauses that are allowed in hypotheses, and a covers relation $covers(e, H, B)$, which basically returns the classification of an example e with respect to H and B, find a hypothesis H in $\mathcal{H}$ that covers (with respect to the background theory B) all positive examples in E^+ (completeness) and none of the negative examples in E^- (consistency). [Rea94]
In this definition we note the addition of language, this is because the language or rather, the definition of linguistic biases are one of the fundamental parts in the construction of an ILP system. One of the reasons for this procedure is the "too much" background knowledge, in fact, redundancy of clauses and equivalent clauses can lead to an exponential growth of background knowledge causing serious inefficiency during the research or an erroneous learning of the hypotheses.
For instance, Metagol uses meta-rules to restrict the syntax of hypotheses and thus the hypothesis space. If a user can provide suitable meta-rules, then Metagol is extremely efficient. However, if a user cannot provide suitable meta-rules (which is often the case), then Metagol is almost useless. This brittleness is applied to ILP systems that employ mode declarations. So if theoretically it is possible to impose any type of bias, it is also true that weak restrictions lead to very poor performance. For good performance, users of mode-based systems often need to manually analyse a given learning task to tweak the mode declarations, often through a process of trial and error.Even for ILP experts, determining a suitable language bias is often a frustrating and time-consuming process. Moreover, if a user makes a small mistake with a mode declaration, such as giving the wrong argument type, then the ILP system is unlikely to find a good solution. By contrast, there are some neural net architectures that can be applied successfully to a large range of diverse problems without requiring any domain-specific tuning. This is probably why ILP is not widely adopted.
Current ILP systems distinguish two kinds of declarative bias: syntactic bias (sometimes also called language bias) and semantic bias.
Syntactic bias acts on the form of the clauses in a hypothesis, while the semantic bias places restrictions on the behaviour of the induced hypotheses.

Syntactic Bias
Formally speaking, a syntactic bias defines the set of well-formed hypotheses $\mathcal{H}$. The set of well-formed hypotheses $\mathcal{H}$ is usually defined from a language bias $\mathcal{L}$, which is the set of syntactically acceptable clauses. Whereas, previously, most ILP systems employed an

implicit built-in syntactic bias, there is a growing interest in general formalism to specify syntactic bias. The advantage of such general formalism is that language bias can be decoupled from particular ILP implementations. Hence, it becomes a true portable parameter of the system, which facilitates the comparison of different systems.
There exist four main approach for syntactic bias: the inductive logic programming language of Bergadano, the antecedent description grammars of Cohen, and their variants. The fourth is parametric languages. All of these approaches are parametric, so it means that a number of parameters determine the syntax of the clause language in the assumptions hypotheses space. The benefit of parametric approaches is that they are easy to implement and modify by changing the language accordingly. n fact, it becomes necessary to modify the linguistic bias in the case in which no solutions are produced, so just modify the parameters to make the language become much more expressive.
In the parametric approach, various parameters have been employed; many of them are rather straightforward and include criteria such as restrictions on the maximum number of variables in a clause, the maximum number of literals in a clause, the predicates allowed in the hypotheses, etc.

Semantic Bias
Although modes and types are usually employed to optimise the efficiency of Prolog compilers they are also relevant to bias the set of acceptable hypotheses in inductive logic programming. Indeed, since Shapiro's MIS, it has become quite standard in inductive logic programming to provide the learner with type and mode declarations. Another very popular method is the use of meta-rules.
The mode declarations provides the learner with the instructions to use a clause. A mode declaration is expressed as:

$$mode(recall, predicate(m_1, ..., m_n))$$

this means that a *predicate* will succeed *recall* times in a single clause, while second argument specifies how to use the *predicate* and the arguments it needs. You can specify whether arguments are input, output or ground by using symbols $+, -, \#$. Different ILP systems use mode declarations in slightly different ways.
Modes are useful for inductive logic programming for two reasons. First, if we are in a single-predicate learning context, and the background predicates and their modes are correctly specified, the learner can guarantee termination by assuring that the queries it generates are mode-conform. Second, the learner can optimise its search when answering queries. Indeed, given the declaration for the predicate, the learner does not need to backtrack after having found a first solution to a query matching the declaration. Given type declarations of the predicate to be learned, the learner need only consider the type-conform subset of its hypothesis space. This can drastically reduce the computation needed.
The other type of semantic bias is meta-rule. The purpose of meta-rules in ILP is to define the admissible structure of the induced predicates, aiming to reduce the hypothesis space. Meta-rules are higher-order clauses and, in addition to the ordinary Prolog

variables, they introduce the concept of second-order variable expressed as:

$$X(A, C) : -Y(A, B), Z(B, C)$$

X, Y, Z are the second-order variables, while letters A, B, C are ordinary (first-order) variables. A second-order variable can be bounded to a predicate. Meta-rules approaches try to find the best substitution for the second-order variables. The major difference between meta-rules and other forms of bias in ILP is that meta-rules themselves are logical statements. Anyway, despite their widespread use, there are few searches and algorithms to determinate the best meta-rules to apply to a learning task.

3.4 What is ILP used for?

Papers: [Bib19, SG15, SM11, AC22, Mug99]
Since the 1970s, basic research in IFP and ILP resulted in the development of fundamental algorithms tackling the problem of inducing programs from input/output examples [SG15]. However, these approaches have remained in the context of artificial intelligence research and have not successfully triggered transfer to technologies that can be applied in a wider context. That depends from restriction that we must guarantee. But with the development of new approaches and techniques, the potential is gained to overcome some of the restrictions of previous systems: learning from very few positive examples becomes possible when users and systems share background knowledge that can be represented in a declarative way, which, combined with name inference, is likely to be more easily understandable [SG15, SM11]. Using algorithmic techniques developed in ILP program can produce learning that results in more powerful ILP algorithms; the adoption of techniques based on domain-specific languages has allowed the realisation of ready-to-use technologies in mass products.
The areas where ILP is most used are [SM11, AC22]:

- **New Scientific Discovery:** maybe the most prominent application of ILP. It generally allows generation of scientific theories starting from observations, generation of discriminating experiments and theory test. ILP is recognised as one of the best methodology to identify and predict ligands and infer missing pathways in protein signalling networks. It is also used in programs that allow the creation of new drugs and in ecology and weather forecast.

- **Program analysis:** thanks to the great expressiveness of logical languages in representing complexity, ILP systems are also very successful in software design applications. ILP systems have proven effective in learning SQL queries, programming language semantics and code searching.

- **Robotics:** even in robotics application what make ILP so attractive is ease modelling of complex realities to provide domain knowledge and the versatility in implementing strategies. For instance, The Robot Engineer uses ILP to design tools for robots and even complete robots, which are tested in simulations and real-world environments. Metagol learns robot strategies considering their resource efficiency.

- **Vision:** Background Knowledge is also valuable in Computer Vision. Recently a research work has shown that Logical Vision can outperform state-of-the-art statistical machine learning, in particular image recognition tasks, by employing ILP thanks to general Newtonian physics background knowledge concerning reflection of light.

- **Data curation and transformation:** Another successful application of ILP is in data curation and transformation, which is largely used due to the fact that ILP can learn from executable programs. Is also used to acquire background knowledge in expert systems. Data transformation is widely used due to the success it has

achieved in synthesising programs for end-user problems, such as string manipulation in Excel. Other really useful transformation tasks include the extraction of data form structured and semi-structured data, such as XML file or Medical Records and the extraction of relations from ecological papers.

- **Learning from trajectories:** the trajectories are intended as interpretation transitions, in other words this type of learning on which it is based automatically builds a model of the dynamics of a system from the observation of its state transitions. Its name is LFIT (learning from interpretation transitions). LFIT has been applied to learn biological models, like Boolean Networks, under several semantics: memory-less deterministic systems and their multi-valued extensions.

ILP difficulties

ILP has many advantages in many applications, but is not as widely used as ML. One problem with the ILP is the lack of well-designed tools. Although over 100 ILP systems have been built, fewer than a handful of systems can be used in a meaningful way by ILP researchers, not to mention that many of them are difficult to use even for experts [AC22]. According to Muggleton et al. in [SM11], this is due to the difficulty of the technology and the lack of familiarity with logic programmes among non- ILP specialists. The fact that many data analysts are not able to come up with an adequate logic formalism to compose a background knowledge base suitable for an ILP system, makes this option discarded a priori, whereas, as Muggleton et al. in [SM11] also state, for at least another 10 years, we will be in a scientific momentum in which we will be able to exploit not only the new and more solid mathematical foundations, but also new insights into the potential of logical representations for machine learning. Unfortunately, driven by industry, researcher had preferred ML technologies. A frustrating problem with ILP systems is that they use many different language biases or even different syntax for the same biases [AC22]. Finding an optimal configuration is quite difficult and even if a suitable one is found, it will not be flexible enough to be reused. But the major open question in ILP is how best to handle noisy and ambiguous data or how to manage a huge BK. This can degrade performance or create ambiguity, from which ILP can suffer. Several solutions have been tried out, but no study has proven the effectiveness of the solution.

Given its diametrically opposed nature, ML is the suitable solution to overcame ILP limitation. Used in a pair the two approaches are complementary and many people have advocated some sort of unification of the two. There is much activity in this area, and much work still to do to produce a truly convincing unification of these very different paradigms [AC22].

3.5 What are main search methods?

Papers: [Kli15, QZ14, Mug91a, Zho17, AC22, AC20]
The main ILP problem is to efficiency search for valid hypothesis in a large hypothesis space. The first ILP search approach were *top-down* and *bottom-up* search, it depends on whether they use generalization or specialisation operators [AC22]. These algorithms rely on notion of generality, usually identified through *θ-subsumption*, in witch one program is more specific or more general than another. A third more recent approach is *meta-level* ILP.

3.5.1 Top-Down approaches.

Top-down approaches start with a general hypothesis and then they specialise them. These systems often employ a separate-and-conquer rule-learning strategy [QZ14]. The advantages in this approach is that they can learn recursive programs (although not all do)[AC22]. A disadvantage is that they can be prohibitively inefficient because they can generate many hypotheses that do not cover the examples.
An example of this approach is **Bottom-Clause propositionalisation**. With propositionalisation we indicate the process of explicitly converting a relational data-set into a propositional data-set. The input data can be examples represented by structured terms, several predicates in first-order logic, or several tables in a relational database. The output is a representation of the attribute value in a single table, where each example corresponds to a row. It represents a new logic-oriented propositionalisation technique which consists in generating bottom clauses, for each first order example and using the set of all the literals in the body of the bottom clauses as possible characteristics.

3.5.2 Bottom-up approaches.

Bottom-up approaches build most-specific clauses from the training examples, and searches the hypothesis space by using generalization. This approach often use the lattice to find the least-general generalisation. By their very nature, bottom-up approaches can be seen as being data- or example-driven [AC22]. An advantage of these approaches is that they are typically fast. The problems with bottom-up approaches is essentially three: they typically use unnecessarily long hypotheses with many clauses, it is difficult for them to learn recursive hypotheses and multiple predicates simultaneously and they do not easily support predicate invention.
A peculiar technique used in this approach is **relative-least-general generalization** o **RLGG**, recommended by Plotkin [Mug91a]. As forecast by the bottom up research, we start from a more specific clause of the background and we must arrive at a much more general one that once added to the background, the general clause will be able to demonstrate more clauses through θ-subsumption. It uses both positive and negative examples. The positives are used to search for the clauses in the hypothesis space and the negatives to reduce the size of the hypothesis space to be searched and to introduce constraints, so as to exclude useless literals for the purpose of the search. Formally we can define it as [Mug91a]:

Let P be a logic program and C and D be two Horn clauses. The relative least general generalisation of C and D, $rlgg(C, D)$ is the least general clause within the θ-subsumption lattice for which $P \wedge rlgg(C, D) \vdash C \wedge D$ where $rlgg(C, D)$ is used only once in the derivation of C and D.

In order to find the RLGG has been introduced a new operator: V-operator. Referring

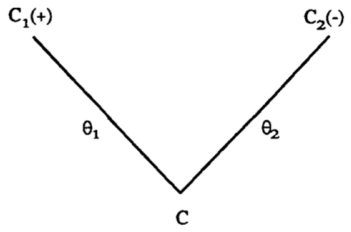

Figure 1: V-operator in single resolution.

to a lattice, resolution derives the clause at the base of the 'V' given the two clauses on the arms. In 1, the literal resolved on is positive (+) in C1 and negative (-) in C2. Note that within any logic program containing C1 and C2, C is redundant. Other clauses within the background theory may also be made redundant by the addition of C1 or C2 to the growing theory. It is the ability to discard redundant clauses that allows these operators to simplify and compact theories.

An approach valued like RLGG is **Inverse-Entailment**. Inverse-Entailment is a relationship of generality in inductive logic programming. More in detail, when learning from an implication using a basic theory, the hypothesis that is learned must cover an example related to the basic theory. The goal is to induce a rule that, together with basic knowledge, entail to example. The inverse implication is based on the observation that if the example is implied by hypotheses and background knowledge, it is equivalent to state that the same denied example is implied by the same knowledge base and by the denial of the rule. The main problem that characterises the use of IE is the presence of multiple declaration templates. In this case, the technique is inapplicable to the program and it needs a relaxation of the conditions for further extensions.

3.5.3 Both Top-down and Bottom-up.

Muggleton, who inspired many other ILP approaches combines both top-down and bottom-up approaches. To generalise an example, Progol uses mode declarations to build the bottom clause, the logically most-specific clause that explains the example. The bottom clause bounds the search from below (the bottom clause) and above (the empty set) [AC22]. Progol then uses an A* algorithm to generalise the bottom clause in a top-down (general-to-specific) manner and uses the other examples to guide the search.

3.5.4 Meta-level

Top-down and bottom-up approaches refine and revise a single hypothesis. A third approach has recently emerged called meta-level ILP. Most approaches encode the ILP problem as a meta-level logic program, i.e. a program that reasons about programs. Meta-level approaches then often delegate the search for a hypothesis to an off-the-shelf solver after which the meta-level solution is translated back to a standard solution for the ILP task. In other words, most meta-level approaches formulate the learning problem as a declarative search problem [AC22, AC20]. The main advantage of meta-level approaches is that they can more easily learn recursive programs and optimal programs and, unlike classic ILP systems, meta-level approaches use diverse techniques and technologies and they are not restricted to the use of Prolog. Most meta-level approaches encode the ILP learning task as a single static meta-level program. A major issue with this approach is that the meta-level program can be very large so these approaches can struggle to scale to problems with non-trivial domains and programs with large clauses.

3.5.5 Answer Set Programming (ASP)

Another approach to the representation of hypotheses is ASP. The Answer Set Programming (ASP) is an approach to nonmonotonic logic programs that builds on the stable model semantics. ASP provides a natural solution to computational problems that are more adequately solved by SAT-based techniques (constraint solving) rather than resolution. Besides the challenges posed by non-monotonicity, ILP systems are not declarative enough. Users are sometimes required to experiment with the ordering of the clauses, this being relevant not only to efficiency but even to termination or correctness. Compared to Datalog and Prolog, ASP supports additional language constructs, such as disjunction in the head of a clause, choice rules, and hard and weak constraints. A key difference between ASP and Prolog is semantics. A definite logic program has only one model. By contrast, an ASP program can have one, many, or even no stable models (answer sets) [AC22]. Due to its non-monotonicity, ASP is particularly useful for expressing common-sense reasoning. An example of ASP program is [AC22]:

$$
\begin{array}{l}
0\{in(V0,V1)\}1 :- edge(V0,V1). \\
reach(V0) :- in(1,V0). \\
reach(V1) :- reach(V0), in(V0,V1). \\
:- not\ reach(V0), node(V0). \\
:- V1 != V2, in(V0,V1), in(V0,V2)
\end{array}
$$

This program illustrates useful language features of ASP. The first rule is a choice rule, which means that an atom can be true. In this example, the rule indicates that there can be an edge between the vertex V1 and V0. The last two rules are hard constraints, which essentially impose integrity constraints. The first hard constraint states that it is impossible to have a node that is not reachable. The second hard constraint states that it is impossible to have a vertex with two in edges from distinct nodes.
The learning process starts by generating all the possible skeleton rules. Given the skele-

ton rules and a set of abductions associated to them, the burden of the search for the final hypotheses is shifted onto the ASP solver. The procedure provide for first phase that derives from the given mode declarations the skeleton rules with an additional abducted clauses in the body that identifies the rule and contains the constants that appear in it. The generation of skeleton rules is the core part of the algorithm and it is driven by mode declarations. A direct generation from mode declarations is not possible since the definition introduces redundancies in the hypotheses space, due to the use of ordered rules. Redundancies like $p(X) \leftarrow q(X), r(X)$ and $p(X) \leftarrow r(X), q(X)$ have the effect of producing multiple answers for the same inductive solution with no benefit on the search. To overcome this problem, it is possible to define a subset $\mathcal{R}^O$, whose elements satisfy a given total order [DC12].
ASP distinguish two main learning approach: brave learners and cautious learners. Brave learners aim to learn a program such that at least one answer set covers the examples while cautious learners aim to find a program which covers the examples in all answer sets. ILASP is notable because it supports both brave and cautious learning, which are both needed to learn some ASP programs.

3.6 What are the fundamental features of logical language that make it crucial in ILP?

Papers: [Rea94, Sta94, AC22, Mug95]
Two of the main issue of ILP are learning recursive programs and setting Background Theory. For this issue we have two solution: *recursion* in ILP and *predicate invention*.

3.6.1 Recursion

The power of recursion is that an infinite number of computations can be described by a finite recursive program [AC22]. To illustrate the importance of recursion, we can think at program:

```
f(A, B) :- tail(A, C), empty(C), head(A, B) .
f(A, B) :- tail(A, C), f(C, B) .
```

In this example, f is the target predicate that we want to learn. We also provide auxiliary information as background knowledge, that is represented as a logic program where *empty(A)* holds when the *list A* is empty, *head(A,B)* holds when B is the head of the list A and *tail(A,B)*, which holds when B is the tail of the list A. Each line of the program is a rule. The first one is a rule for termination, while the second one is a recursive rule that allows you to add an undefined number of items in the list [AC22].
Without recursion, an ILP system would need to learn a separate clause to find the last element for each list of length n, such as this program for when $n = 3$:

```
f(A, B) :- tail(A, C), empty(C), head(A, B) .
f(A, B) :- tail(A, C), tail(C, D), empty(D), head(C, B).
f(A, B) :- tail(A, C), tail(C, D), tail(D, E), empty(E), head(D, B).
```

This program does not generalise to lists of arbitrary lengths. Moreover, most ILP systems would need examples of lists of each length to learn such a program. In the second program each rule can produce the string by itself, but the length is chosen and limited. But in the first program, because of the symbolic representation and the recursive nature, it generalises to lists of arbitrary length and which contain arbitrary elements. In general, without recursion, it can be difficult for an ILP system to generalise from small numbers of examples. A common limitation with existing approaches is that they rely on bottom clause construction. In this approach, an ILP system creates the most specific clause that entails the example. That is done for each example in example set. Then system tries to generalise the clause to entail other examples. However, this sequential covering approach requires examples of both the base and inductive cases. Interest in recursion has revived with the introduction of meta-interpretive learning. The key idea of MIL is to use meta-rules or templates to restrict hypotheses space by restricting the form of inducible program. A meta-rule is an higher-order clause, this means that the goal of a MIL system is to find a suitable substitution for higher-order variables. For this reason many meta-level ILP system can learn recursive program in the same way as MIL.
With recursion, ILP systems can now generalise from small numbers of examples, often a single example. Moreover, the ability to learn recursive programs has opened up ILP

to new application areas, including learning string transformations programs, answer set grammars and general algorithms.

3.6.2 Predicate Invention

A key characteristic of ILP is the use of background theories, which contains facts and rules in the form of a logic program [AC22]. It's a great advantage across a wide range of domains, because of the possibility to represent complex relationships in complex domains, such as theory of light to understand images, or higher-order operations to solve programming puzzle. So choosing appropriate background is crucial for good learning performance. ILP has traditionally relied on hand-crafted background, often designed by domain experts. This approach is limited because obtaining suitable background can be difficult and expensive. Indeed, the over-reliance on hand-crafted background is a common criticism of ILP.
Rather than expecting a user to provide all the necessary background, the goal of predicate invention (PI) is for an ILP system to automatically invent new auxiliary predicate symbols [Sta94]. This idea is similar to when humans create new functions when manually writing programs, to reduce code duplication or to improve readability. Whilst PI has attracted interest since the beginnings of ILP not many ILP systems implement it. That is because it is a challenge, for the system, to decide when and how to invent a new symbol, this is difficult because it is not clear how many arguments an invented predicate should have, how the arguments should be ordered, etc.
The theoretical characterisation of predicate invention is based on the definition of the program P and the set of all predicate symbols found in the heads of clauses of P, $\mathcal{P}(P)$ called the vocabulary of definition of P. ILP has the following main three vocabularies of definition [Rea94]:

Examples (Observations) Vocabulary: $\mathcal{O} = \mathcal{P}(E^+ \cup E^-)$
Theoretical vocabulary: $\mathcal{T} = \mathcal{P}(B) - \mathcal{O}$
Invented vocabulary: $\mathcal{I} = \mathcal{P}(H) - (\mathcal{T} \cup \mathcal{O})$

Several discussions on the subject have led to the definition of additional conditions for PI to work.

Necessary invention: $\mathcal{I} \neq \emptyset$ for each H which provides sufficiency and consistency.

In other words, predicate invention is only necessary when there does not exist a finite axiomatisation of the predicates in $\mathcal{O}$ containing only predicate symbols from $(\mathcal{T} \cup \mathcal{O})$. Although it is possible to introduce new predicates regardless of whether they are necessary, it is clear that any one of a potentially infinite set of new predicates could be introduced. It seems reasonable that when it is necessary to extend the vocabulary, this should be done in as conservative a manner as possible. To do so requires a notion of ordering over invented predicates.
There are several approaches to implement Predicate Invention.

A classical approach to PI is to predefine invented symbols through mode declarations, which call **placeholders**. However, this placeholder approach is limited because it requires to specify a bias where a user manually specify the arity and argument types of a symbol, which rather defeats the point or requires generating all possible invented predicates, which is computationally expensive.
An implementation that allows automatic PI is MIL. In automatic PI, user does not need to predefine invented symbols, this means that we can avoid issue of older ILP systems by using **meta-rules** to define the hypothesis space and in turn reduce the complexity of inventing a new predicate symbol. To induce longer clauses, such as to drop first three elements from a list, Metagol uses the same meta-rule but invents a predicate symbol to chain their application. A side-effect of this metarule-driven approach is that problems are forced to be decomposed into reusable solutions. PI has been shown to help reduce the size of target programs, which in turn reduces sample complexity and improves predictive accuracy. Several new ILP systems support PI using a metarule-guided approach. There are metarule-driven PI approaches that perform PI during the learning task. A recent trend is to perform PI as a **pre/post-processing** step to improve knowledge representation. It is usually used in unsupervised learning. There are three main approaches to pre/post-processing of background and they are quite different from each other.
The first is CUR^{2}LED. It performs PI by clustering constants and relations in the provided background, turning each identified cluster into a new background predicate. The purpose of CUR^{2}LED is not to use a single similarity measure, but rather a set of various similarities . This choice is motivated by the fact that different similarities are useful for different tasks, but in the unsupervised setting the task itself is not known in advance.
ALPs perform PI using an auto-encoding principle: they learn an encoding logic program that maps the provided data to a new, compressed latent representation, defined in terms of the invented predicates, and a decoding logic program that can reconstruct the provided data from its latent representation. This approach shows improved performance on supervised tasks, even though the PI step is task-agnostic.
Knorf pushes the idea of ALPs even further. Knorf compresses a program by removing redundancies in it. The refactored program is smaller in size and contains less redundancy in clauses, both of which lead to improved performance. The authors experimentally demonstrate that refactoring improves learning performance in lifelong learning and that Knorf substantially reduces the size of the BK program, reducing the number of literals in a program by 50% or more [AC22].
An approach to acquiring BK is to learn it in a lifelong learning setting. The general idea is to reuse knowledge gained from solving one problem to help solve a different problem. Meta rules are useful in this context, as they allow ILP systems to adapt to tasks. An example of meta rules-driven ILP system that allow that is Metagol$_{DF}$, an extension of Metagol. Given a set of tasks, it uses Metagol to try to learn a solution for each task using at most one clause. If Metagol finds a solution for a task, it adds the solution to the B and removes the task from the set. This approach allows Predicate Invention but prefers to reuse solutions from previous solved tasks. This process repeats until Metagol$_{DF}$ solves all the tasks or reaches a maximum program size. The method is

to start from easier tasks and reuse solutions to solve the most difficult tasks. Multi-task approach performs substantially better than a single- task approach because learned programs are frequently reused and leads to a hierarchy of induced programs. Metagol$_{DF}$ saves all learned programs (including invented predicates) to the background, which can be problematic because too much irrelevant background is detrimental to learning performance. To address this problem, some ILP systems introduce the idea of forgetting. In this kind of approach continually grows and shrinks its hypothesis space by adding and removing learned programs to and from its background. It was demonstrated that forgetting can reduce both the size of the hypothesis space and the sample complexity of an ILP learner when learning from many tasks.

3.7 What are main Implementation?

Papers: [QZ14, Rea94, CJ06, Mug91a, Rea94, SM10, Mug95, OR03, Mug17, Mug19, AC20, DC12, Ben18, ML20a, Dar20, A.K15]
Countless ILP implementation exist, for this reason we will only deal with the main ones. Some of the most important ILP implementation are *FOIL, Golem, Progol, Metagol and ILASP*.

3.7.1 FOIL

FOIL is a supervised learning implementation that learn function-free Horn clause (which are a subset of first order logic clause) definitions of a relation in term of itself of other relations. To make that, a top-down approach is used. FOIL systems use Prolog semantics and in addiction they can employ constants in definitions they produce.
FOIL's input consists of information about the relations, one of which (the target relation) is to be defined by a Horn clause program. For each relation it is given a set of tuples of constants that belong to the relation. For the target relation it might also be given tuples that are known not to belong to the relation; alternatively, the closed world assumption may be invoked to state that no tuples, other than those specified, belong to the target relation. Tuples known to belong to target relation are positive examples and they are marked with the symbol $\oplus$ and those not in the relation are negative examples and they are marked with $\ominus$ [CJ06].
FOIL starts with the head of the clause and specialises it by adding literals to the body of the clause, until no negative examples are covered by the clause or when heuristics indicate that the clause is too complex. As new variables are introduced by the added literals, the size of the tuples in the training set increases so that each tuple represents a possible binding for all variables that appear in the partially-developed clause.
If the target relation R has k arguments, the process of finding one clause for the definition of R can be summarised as follows [QZ14]:

Algorithm 1 FOIL general algorithm

Pos ← *positive examples*
Neg ← *negative examples*
while *Pos not empty* **do** ▷ Learn a new rule
  $NewRule \leftarrow$ *most general rule possible*
  $NegExamplesCovered \leftarrow Neg$
  while $NegExamplesCovered$ *not empty* **do** ▷ Add a literal to specialise $NewRule$
    $Candidate\_Literal \leftarrow generate_candidate()$
    $Best\_literal \leftarrow argmax_{L \in Candidate\_Literal} FOIL\_Gain(L, NewRule)$ ▷ Add $Best\_literal$ to $NewRule$ preconditions
    $NegExamplesCovered \leftarrow$ *subset of* $NegExamplesCovered$ *that satisfies* $NewRule$ *preconditions*
  end while
  $Learned\_Rules \leftarrow Learned\_Rules + NewRule$
  $Pos \leftarrow Pos-$ {member of Pos covered by $NewRule$}
end while
return $Learned\_Rules$

The outer loop adds new rules to the output until no more positive examples are covered, while the inner loop searches for the next best rule by incremental specialisation. Although FOIL incorporates a simple backup mechanism, the clause-building process is essentially a greedy search; once a literal is added to a clause, alternative literals are usually not investigated.
So, the key question is how to determine literals and appropriate literals to append to the developing clause. Some clauses in reasonable definitions will inevitably contain literals with zero gain. We suppose to have a property D, and the literal $D(X, Y)$ that defines the value Y for object X. If it maps value from X to Y, the gain will always be zero. But if we suppose that, for any value of X, supplied several possible values for Y, then the literal might even have negative gain. Instead, if X is a previously defined variable and Y is a new variable, there is an important difference because it may exclude some tuples or may cause the number of tuples in the training set to grow. This is the key insight underlying certain literals, an idea inspired by GOLEM's certain terms the value of each new variable is forced or determined by the values of existing variables. To choose the best literal FOIL use an information gain formula, i.e.:

$$FOIL\_GAIN(L, R) = t\left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0}\right)$$

where:

- L is the candidate literal to add to rule R.
- p_0 is the number of positive binding of R.
- n_0 is the number of negative binding of R.

- p_1 is the number of positive binding of $R + L$.
- n_1 is the number of negative binding of $R + L$.
- t is the number of positive number of R also covered by $R + L$.
- $-\log_2 \frac{p_0}{p_0+n_0}$ is the optimal number of bits to indicate a class of positive binding covered by R.

As we can see in gain formula, FOIL notes determinate literals found while searching for gainful literals. The maximum possible gain is given by a literal that excludes all $\ominus$ tuples and no $\oplus$ tuples [CJ06]. Unless a literal is found whose gain is close to ($\geq 80\%$ of) the maximum possible gain, FOIL adds all determinate literals to the clause and tries again. This may seem rather extravagant, since it is unlikely that all these literals will be useful. However, FOIL incorporates clause-refining mechanisms that remove unnecessary literals as each clause is completed, so there is no ultimate penalty for this all-in approach.
The method that find literals could be done indefinitely, for this reason, FOIL borrows another idea from GOLEM. Variables in left-hand side of the clause have depth 0, while a variable that first occurs in some literal has depth one greater than the greatest depth of any previously-occurring variable in that literal [Rea94]. By placing an upper limit on the depth of any variable introduced by a determinate literal, we rule out indefinite runaway. This limit does reduce the class of learnable programs. However, a sufficient stopping criterion requires that the hypothesis covers all the positive examples and none of the negative examples, the necessary stopping criterion requires that the building clause has no negative tuple ($T = \emptyset$) [QZ14].
FOIL systems, like other first-order systems, are able to learn recursive theories. Recursive theories in first-order systems are expressive and hence powerful. It increase expressiveness, but more care must be taken to avoid nonsensical recursion. For this reason several conditions are imposed to add literals on right-hand side, moreover a literal on the right-hand side must be judged to be less than the head of the clause in some ordering of literals. The earlier versions of FOIL used a method that discover ordering that guaranteed that a single clause could not lead to a recursive loop by calling itself. The order discovery was removed in following releases, which relied on the user specifying the constants of each type in an appropriate order. Order discovery mechanisms have been reinstated in the most recent versions and the method of ordering recursive literals has been generalised so that the guarantee now applies to sets of clauses for a single relation, not just to a single clause. Ergo FOIL implements a method to compare variable of same type to ensure ordering of literals for the purpose of not to create recursive loop.

3.7.2 GOLEM

GOLEM is an ILP implementation that uses bottom-up search to find hypotheses in the hypotheses space. In particular GOLEM makes use of Relative Least General Generali-

sation (RLGG) which are based on subsumption relative to a bottom clause. GOLEM is one of most classic ILP algorithm to use RLGG approach. Among all the clauses assembled with RLGG, GOLEM chooses those that cover the greatest number of positive examples and does not cover the negative examples. On the next step GOLEM generalises the best RLGG. This process continues until increasing the set of cover positive examples from the constructed RLGG stop. As a final step GOLEM reduces constructed RLGG by dropping irrelevant literals. Both the BK and examples consist only ground facts. GOLEM uses Horn clauses, this means that clauses have a literal in the head and n literals in the body, furthermore the length of ij-determinate RLGG clauses were shown to be bounded by a polynomial function for given values of i and j [SM10]. Without this constraint Plotkin showed that the length of RLGG clauses grows exponentially in the number of positive examples covered.
In this example the algorithm shown uses the cover set approach to construct a theory consisting of more than one clause [SM10]:

Algorithm 2 GOLEM general cover set algorithm

Input: Example E, mode declarations M, background knowledge B
$T = \{\}$
$S = E^+$
while S *not empty* **do** ▷ Generalisation process
  $e = example\ in\ S$
  $C_e = Construct\ of\ bottom\ clause\ from\ e, B\ and\ M$
  $C = ComputeRLGG(C_e, E)$
  $C' = Negative\_Base\_Reduction(C, E^-, B)$
  $T = T \cup C'$
  $S' = example\ S\ covered\ by\ C'$
  $S = S - S'$
end while
return T ▷ Set of learned rules

To construct C_e the body of the example is filled with all the rules present in the background knowledge and then the terms of the body of each example clause that do not contain the arguments of the head of the other clause of the pair are filtered. The RLGG of the pair of rules are calculated by the method *computeRLGG* and the generalisation is applied to each equal argument to obtain a final general rule. Generally the procedure for calculating a RLGG provides that initially the *lgg* of the head is calculated by combining the two heads of the pair of clauses, then the *lgg* of the body is calculated by combining each term of the body of the first example clause with each term of the body of the second and finally the general clause is constructed from the calculated *lggs*. Methods of reduction are necessary, because all the RLGG clauses that prove at least one negative example are removed. Finally, the coverage of generalisations is calculated and the algorithm starts a new iteration with the remaining uncovered examples.
The output is induced hypothesis in generalised Horn form.
One of the best procedure for calculating a RLGG is the *Best ARMG Algorithm*. Gen-

erally, the use of ARMG in GOLEM is to repeatedly extend ARMGs using positive examples and keep the best ARMGs at each iteration to be extended in the next iteration until the ARMGs' score no longer increases [SM10].

Algorithm 3 Best ARMG algorithm

```
Input: Positive Example E+, Bottomclause C_e, Sample size K, Beam width N
best_armgs = {C_e}
do
    best_score = highest score from best_armgs
    Ex = K random positive examples from E+
    new_armgs = {}
    for C in best_armgs do
        for e' in Ex do
            C' = C
            while there is a blocking atom b_i with reference to e' in C' body do
                C' = C' without b_i
                C' = C' without b_i not head-connected to C'
            end while
            if score(C') > best_score then
                new_armgs = new_armgs ∪ C'
            end if
        end for
    end for
    if new_armgs ≠ {} then
        best_armgs = N highest scoring clauses from new_armgs
    end if
while new_armgs = {}
return T                                          ▷ Set of learned rules
```

Regarding the *Negative_Base_Reduction* its purpose is to generalise a clause by keeping only the literals that prevent negative examples from being proved. The algorithm is simple and work as follow. Given a clause $h \leftarrow b_1, ..., b_n$, find the first literal, b_i such that the clause $h \leftarrow b_1, ..., b_i$ covers no negative examples. Prune all literals after b_i and move b_i and all its supporting literals to the front, yielding a clause $h \lrcorner S_i, b_i, T_i$, where S_i is a set of supporting literals needed to introduce the input variables of b_i and T_i is $b_1, ..., b_{i1}$ with S_i removed. Then reduce this new clause in the same manner and iterate until the clause length remains the same within a cycle [SM10]. Both algorithms are indispensable for the calculation of RLGG.
RLGG does not allow the learning of complex hypotheses, such us recursive rules, and it limits induction to simple rules, having only terms in the body with arguments present in the head [Mug91a, SM10, Mug95]. Furthermore, the construction of the *lgg* lattice can grow exponentially as the basic knowledge provided in input increases, significantly slowing down induction without leading to the expected result.

3.7.3 PROGOL

The Progol implementation is a system that performs learning through both Bottom-Up and Top-Down research methods. It uses the inverse entailment (IE) technique for the constructing hypotheses [Mug95]. The peculiarity of Progol consists in the application of the Top-Down method for the construction of general clauses, called most specific clauses, equal to the number of mode declarations of the input head, i.e. rules having the literal of the example as head and for body only the literals of basic knowledge which contain among their arguments the arguments of the head. Secondly, it applies the Bottom-Up method for the construction of clauses with maximum compression, i.e. rules deriving from the previous ones to which the body literals that do not conform to the hypothesis to be induced are removed and, therefore, making substitutions from the general to the particular for prove the induced facts. Therefore, given Horn theories B, E^+ and E, Progol aims to return an augmented theory $B' = B \cup \{h_1, h_2, ...h_n\}$ that entails E^+ and is consistent with E. Each hypotheses h_i tries to cover the largest number of remaining positive examples, not including negative ones and containing the fewest number of literal [OR03]. The use of this technique implies the definition of mode declarations to prevent the explosion of the clauses, for this reason Progol takes a set of M of *mode of declaration* that specifies a language bias with which hypothesised clauses must be compatible. Mode-declarations consist of head-declarations and body-declarations that impose syntactic constraints on the head and body atoms of hypothesised clauses. If p and t are predicates and X is a variable, then the head-declaration $modeh[p(+t)]$ states that the atom $p(X)$ may appear in the head of a hypothesis clause, and the body-declaration $modeb[p(+t)]$ states that the atom $p(X)$ may appear in the body providing X appears also in the head.
Similar to the Golem system, Progol is one of the first algorithms developed in inductive programming but, unlike the previous one, it allows the learning of more complex clauses not composed only of literals in the body containing only the arguments of the head literal; for example $head(X, Y) : -bodyTerms(X, Y, Z)$, where $bodyTerms(X, Y, Z)$ represents the set of conjunctions between the literals of the body having as arguments not only the X and Y of the head, but also a hypothetical Z calculated by induction.
Just like GOLEM, Progol only learns a rule at a time. For this reason all the positive examples used to induce a hypothesis must necessarily have the same functor. Furthermore, this algorithm exploits the definition of input arguments for *modeh* and *modeb*, i.e. arguments that allow you to directly select the correct literals in order not to build inconsistent rules. To produce a rule, Progol selects a seed example e from among those remaining in E^+, then it constructs a *maximally compressed hypothesis* h that together with the current B covers at least e. Finally, it adds h to B and removes all covered examples from E^+. The step of hypothesis formation is performed in two stages. A finite Horn subset of *most specific clauses* is constructed by START-SET and BOTTOM-SET, and is generalised by SEARCH. The head atom is computed by START-SET by reasoning with contrapositives, then the body atoms are computed by BOTTOM-SET using an interpreter, and finally the most compressed generalisation is determined by SEARCH using a general-to-specific search through the $\theta-$subsumption lattice.

Algorithm 4 Progol general algorithm

Input: Example E, mode declarations *modeh* and *modeb*, background knowledge B
$S = E^+$
$T = \{\}$ ▷ clauses induced
while S is not empty **do**
$e = example\ in\ S$
$mostSpecificClause = construction(e, modeh, modeb, B)$
$C = search(mostSpecificClause, S, E^-, B, modeh, modeb)$ ▷ Searching for clause with max compression
$T = T \cup C$
$S' = example\ S\ covered\ by\ C'$
$S = S - S'$
end while
return T ▷ Set of learned rules

Therefore, for each positive example e not yet covered, Progol uses START-SET and BOTTOM-SET in the *construction* method to construct a rule that at least covers e [OR03]. The *modeh* and *modeb* mode declaration are fundamental in this process, since in the subset of the more specific clauses, as many most specific clauses are created as there are *modeh* with the same functor of the example. As for the body it is built through the conjunction of the basic knowledge literals that replace each *modeb*, as long as they have the same functor as the *modeb* and at least one argument belonging to the input arguments found so far [Mug95]. Another parameter to guide the construction of body is *recall*. Recall is a parameter that is part of *modeb* and indicates the number of times the functor is repeatable in the rule.

Algorithm 5 Construction algorithm

Input: Mode declarations *modeh* and *modeb*, background knowledge B, example e
Depth bound h, clause length i
$k = 0$
$InTerms = \{\}$
$mostSpecificClause = <>$
$\overline{e}$ *be the clause normal form logic program* $\overline{a}$
if no *modeh* **then return** $\square$
end if
m first declaration of modeh such that $a(m)$ *is less general of a with* $\theta-$*substitution*
$a_h = copy(a(m))$
for (v/t) *in* θ_h **do**
  if v *corresponds to a #type in modeh* **then**
    replace v *by* t *in* a_h
  else
    replace v *in* a_h *by* v_k *where* $k = hash(t)$
    if v *corresponds to a* $+type$ **then**
      $InTerms = InTerms \cup v$
    end if
  end if
end for
$mostSpecificClause = mostSpecificClause \cup a_h$
while $k \neq i$ **do**
  for m *in modeb* **do**
    $V(m) = \{v_1, .., v_n\}$ ▷ *be the variables of* $+type$ *in* $a(m)$
    $T(m) = T_1 \times ... \times T_n$ ▷ *be a set of n-tuples of terms such that each* T_i *corresponds to the set of all terms of the type associated with* v_i *in* m
    for $< t_1, .., t_n >$ *in* $T(m)$ **do**
      $a_b = copy(a(m))$
      $\theta = \{v_1/t_1, .., v_n/t_n\}$
      if $deduction(h, T(m), V(m), a_b\theta)$ *not fail* **then** ▷ If Prolog with depth-bound h succeeds on goal $a_b\theta$ with the set of answer substitutions Θ_b
        for θ_b *in* Θ_b **do**
          for v/t *in* θ_b **do**
            if v *correspond to a #type* **then**
              replace v *by* t *in* a_b
            else
              replace v *in* a_h *by* v_k *where* $k = hash(t)$
              if v *corresponds to a* $-type$ **then**
                $InTerms = InTerms \cup v$
              end if
            end if
          end for
        end for
        $mostSpecificClause = mostSpecificClause \cup a_b$
      end if
    end for
  end for
end while
return $mostSpecificClause$

This algorithm is explained in detail by Muggleton in [Mug95].
Through *search* method, Progol searches for the clause with maximum compression, that is the clause whose size of the body corresponds to the sum of the recall of each *modeb* used in the clause itself. The search for the clause is done by trying to add each literal of the body of each most specific clause and finally selecting the one with the highest score. The score is calculated by a simple formula:

$$score = p - (n + c + h)$$

where

- p is the positive examples covered;
- n is the negative examples covered;
- c is current body length;
- h is number of missing literals at maximum compression.

When Progol encounters, during the search, rules that do not respect the constraints of the declaration methods, it attributes a score value of -20 to those rules. This evaluation allows you to discard in advance most of the rules that do not fit expectation. To implement the *search* method, the algorithm A^* is usually used.
Although, like GOLEM, Progol cannot create recursive clauses, the rules it produces are very powerful and complex, because literals present in the body having not only head arguments. Furthermore, its performances are comparable to those of FOIL although Progol does not have the ability to invent new literals and has much more stringent constraints. FOIL also uses extensional background knowledge rather than the intensional background knowledge of Progol and GOLEM.
Even though, both GOLEM and Progol use a background knowledge composed by unique facts, which has generalisation limits, for GOLEM is impossible to calculate *lgg* from different clauses, while Progol can define two different *modeb* which are not a priori demonstrable in the background knowledge.
Muggleton, the creator of inversion entailment has shown that the notion of inverting entailment is of a more fundamental nature than that of inverting proof, since it is based on the model-theory which underlies proof. This approach has led to the development of a new state-of-the-art ILP system called Progol.

3.7.4 METAGOL

Metagol is an ILP system based on Meta-Interpretive Learning (MIL). MIL is a recent ILP technique aimed at supporting learning of recursive definitions and purely symbolic systems created. It also introduces a new feature, that is, when learning predicate definition, it automatically introduces sub-definition, allowing for decomposition into a hierarchy of reusable parts. This is possible thanks to *predicate invention*, indeed, if we suppose to machine learn a set of kinship relationship, we provide ancestor relationship

examples and the background contains only father and mother facts, then the system must not only be able to learn ancestor as a recursive definition but also simultaneously invent parent relationship to learn these definitions. Although the topic of Predicate Invention was investigated in early ILP research it is still seen as hard and under-explored.
Most forms of program induction are biased towards learning simple programs, typically those with minimal textual complexity, such as the number of clauses or the number of literals. This is possible through the use of biases that ignore the efficiency of the hypothesised programs. An example of this are Golem and Progol. Both can learn very useful algorithms but at the same time they are limited in many aspects. Moreover, ILP systems such FOIL have no predicate invention and limited recursion learning, therefore cannot learn recursive grammars from example sequences. By contrast, Metagol does not define the clause length bias, but meta-rules. Meta-rule can be seen as a simpler linguistic bias and like for mode declaration in previous ILP systems, meta-rules are required as input parameters.
A meta-rule is very similar to an ordinary clause, but a meta-rule has some second-order (higher-order) variables in its head and/or body which means that not only the arguments of terms can be substituted but also functors. As meta-rules are higher-order variables, there is no logic engine that natively supports the representation of a meta-rule as first-class abstractions, therefore a new level of abstraction is introduced. A meta-rule is defined as follows [Mug19]:

$$metarule(name, Subs, (Atom :- Body))$$

where

- *name* is the name of meta-rule;
- *Subs* denote the higher-order variables in meta-rule;
- *Atom* denote the literals in the head of the rule;
- *Body* denote the literals in the body of the rule;

Keeping this definition in mind we can define main meta-rules:

Name	**Meta-rule**
Ident	$P(A,B) \leftarrow Q(A,B)$
Precon	$P(A,B) \leftarrow Q(A), R(A,B)$
Curry	$P(A,B) \leftarrow Q(A,B,F)$
Chain	$P(A,B) \leftarrow Q(A,C), R(C,B)$
Tailrec	$P(A,B) \leftarrow Q(A,C), P(C,B)$

Table 5: Table of the main meta-rules [Mug17, Mug19]. The letters P, Q, R, and F denote higher-order variables. The letters A, B, and C denote first-order variables.

The Metagol induction algorithm is basically a Prolog meta-interpreter. Metagol tries to prove each atom in turn. It first attempts to deductively prove an atom using compiled background knowledge by delegating the proof to Prolog (call (Atom)), where the compiled background knowledge contains standard Prolog definitions. Metagol uses prim statements to allow a user to specify which predicates are part of the compiled background knowledge. The prim statements are of the form prim (P / A), where P is a symbol of the predicate and A is the associated arity. If this deductive step fails, Metagol tries to unify the atom with the head of a metarule ($metarule(Name, Subs, (Atom :- Body))$) and to bind the higher-order variables existentially quantified in a meta-rule to the symbols in the signature of the predicate. Metagol saves the resulting substitutions and tries to test the body of the meta-rule. After trying all the atoms, a Prolog program is formed by projecting the substitutions onto their corresponding meta-rules. Metagol checks the consistency of the learned program with negative examples. If the program is inconsistent, Metagol goes back to explore different branches of the SLD tree [Mug19]. Metagol uses iterative drill down to ensure that the first consistent hypothesis returned has the minimum number of clauses. The search begins at depth 1. At depth d the search returns a coherent hypothesis with at most d clauses if any exist; otherwise it continues at depth d + 1. At each depth d, Metagol introduces d - 1 new predicate symbols. The following pseudo-code show how work the meta-interpreter.

Algorithm 6 Metagol general algorithm

Input: Example E, Metarules M, background knowledge B, depth D
$rules = \{\}$ ▷ rules induced
$prog = \{\}$
for *metarule* in M **do**
  $prog = learn(metarule, E^+, E^-, D, B)$
  $prog = applyBiasShift(prog, B)$
  if $verifyConsistency(prog, B, E^-)$ **then**
    $rules = rules \cup prog$ ▷ else backtrack to next metarule
  end if
end for
return *rules*

For each meta-rule in meta-rules set the meta-interpreter first tries to prove all the positive examples with the current meta-rule through *learn* method. Before inducing a new rule it tries to deduce positive examples with background knowledge or already induced rules. When interpreter discover that a new rule needs to be induced, it perform a *partial substitution* that is used to replace a higher order variable in head with the predicate of the example, but arguments (first order variable) remain unchanged. Subsequently it tries to prove the rule by replacing all higher-order variables in the body with literals from the background knowledge and induced rules. If no rule is found, Metagol proceeds with the invention of predicates. In this scenario abstraction allows to use higher order variables in curry metarule to recursively solve the problem, decomposing it into sub-problems that are solved one at the time in the same way. By successively

interleaving these two steps, Metagol supports the invention of conditions and functions to an arbitrary depth.
Through the use of the *applyBiasShift* method, any redundancies arising from the predicate invention can be eliminated. Finally, the last check on the consistency of the induced rules is performed in the *verifyConsistency* method. Consistent rules are added to background otherwise the interpreter directly backtrack to another metarule and explores a new branch in SDL-tree.
In contrast to previous algorithms, the induction technique exploited by Metagol is carried out with a completely different logic, a true evolution in rule learning. The first difference is the lack of constraints due to the generality problem and the lack of true bias. In fact, the biases for describing rule composition are replaced by meta-rules that are more versatile and powerful. Meta-rules also make it possible to induce recursive rules and to invent predicates out of necessity, not to mention the possibility of having more complex reasoning. However, while a step ahead of FOIL, Golem and Progol, it is still limited to the induction of rules composed of terms with a maximum arity of 2 and a body size of no more than 3. Finally, choosing a proven set of meta-rules for induction, while easier than defining linguistic bias, remains a problem to be solved.

3.7.5 ILASP

The goal of Inductive Logic Programming (ILP) is to learn a program that explains a set of examples in the context of some preexisting background knowledge. The theoretical framework underlying the ILASP system differs from conventional approaches for Inductive Logic Programming (ILP), which are mainly focused on learning Prolog programs. The paradigm that ILASP uses is ASP. Due to the declarative nature of ASP, the learning process in ILASP aims learning the logical specification of a problem, rather than the procedure for solving it; moreover, programs learned by ILASP can include other types of rules that are not available in Prolog, such as choice rules and hard and weak constraints [Ben18]. Learning these additional rules has opened up new applications that were previously outside the scope of ILP systems; for example, learning weak constraints allows ILASP to learn a user's preferences from examples of his or her preferred solutions. It has been shown that ILASP's learning framework generalises existing frameworks and systems for learning ASP programs, such as the *brave learning framework*, adopted by almost all previous systems, and the less common *cautious learning framework*. The two approaches are quite different from each other, which is why they are used to solve different problem. Law et al. in [ML20a] showed that some ASP programs cannot be learned with either a courageous or a prudent approach and that a combination of courageous and prudent reasoning is required to learn ASP programs in general. ILASP's learning framework allows this combination and is capable of learning the entire class of ASP programs.
ILASP consists of three main components: background knowledge B, mode bias M and examples E. Regarding background knowledge, there are no major differences from other ILP systems. Even the definition of the language bias, or mode bias, are quite similar to those of Golem and Progol. The mode bias M is used to express the ASP programs that

can be learned; for example, it specifies which predicates may be used in the head/body of learned rules, and how they may be used together. From M, it is possible to construct a finite set of rules S_M called the *rule space*, that contains every rule that is compatible with M. The power set of S_M , $\mathcal{P}(S_M)$, is called the *program space*, and contains the set of all ASP programs that can be learned [ML20a]. Instead, the examples E differ from other ILP systems. While they adhere to the core principles of ILP systems such as the goal and coverage relationship they maintain a different structure given the different operation of the ASP paradigm underlying ILASP.
Many ILP systems learn from (positive and negative) examples of atoms which are true or false. This is because many ILP systems are targeted at learning Prolog programs, where the main output of a program is a query of a single atom. In ASP, the main output of a program is a set of answer sets. For this reason, ILASP learns from positive and negative examples of partial interpretations, which are a set of responses that the program should learn or avoid. These examples are sufficient to learn any ASP program consisting of normal rules, choice rules and hard constraints. Otherwise, weak constraints cannot be learned using only positive and negative examples, because they can only specify what should (or should not) be a set of responses. Weak constraints have no effect on what is or is not an answer set, but only create an order of preferences over answer sets. For this reason, ILASP allows a second type of example called an *ordering example*, the semantic property of which is a preference ordering over a pair of answer sets of $B \cup H$ [ML20a]. Learning weak constraints corresponds to a form of "preference learning". Like positive and negative examples, ordering examples come in two forms: brave orderings, which express that at least one pair of answer sets that satisfy the semantic properties of a pair of positive examples are ordered in a particular way, and cautious orderings, which express that every such pair of answer sets should be ordered in that way.
ILASP, due to its nature, can also require *Noisy Examples* [Ben18]. All the assumptions made assume that all examples are labelled correctly and thus that all examples are covered by the learned program. In real-world applications, of course, this is often not the case; examples may be noisy (i.e., mislabelled) and thus finding a program that covers all examples may not be possible, or even desirable (overfitting on examples). In ILASP, each example can be assigned a penalty, which is a cost for not covering that example. The search for an optimal learned program now looks for a program that minimises $|H| + cost$, where $|H|$ is the length of the program and *cost* is the sum of the penalties of all examples not covered by the learned program. The $|H| + cost$ function is an example of a scoring function. Most systems, including ILASP, provide a built-in scoring function that cannot be modified, but is possible to define methods that allow the user to define his or her own scoring function, allowing a customised (domain-specific) interpretation of optimality.
The algorithm behind ILASP consists of 3 main phases: *pre-processing*, that map input learning task into an ASP program; *learning*, where the ASP program is solved by an ASP solver; finally the answer set, output of the solver, is *post-processed* to extract the the learned program [ML20a]. ILASP is currently on his third version, ILASP 3. Before this, there are three other versions, ILASP 1, ILASP 2, ILASP 2i. Each version is an

update of the previous one. ILASP 3 uses a novel method of translating an example into a set of coverage constraints over the solution space. The intuition of these coverage constraints is that they give a list of conditions which are satisfied by exactly those programs that cover the examples. What make ILASP 3 better than previous ILASP is the addition of the *Example Translator*, which has two steps: firstly, it translates the relevant examples into a set of constraints on the solution space; and secondly, in the implication check step, it checks which other examples would be guaranteed to not be covered if the coverage constraints were not satisfied. The coverage constraints give an approximation of the coverage and score of every program in the program space. This approximation of a program's score is always guaranteed to be less than or equal to the program's real score, as it will only overestimate the program's coverage. As the approximation never underestimates the coverage of H, it suffices to only search for a relevant example within the set of examples that the approximation says H should cover. To facilitate this, in addition to returning H, the program search also returns the set of examples, $Uncov$, which are known not to be covered by H (according to the coverage constraints). The relevant example search is then within E and $Uncov$, rather than the full example set E. ILASP3 has several other optional features, designed to boost performance on certain types of task.

Algorithm 7 ILASP general algorithm

Input: Example E, Background knowledge B, language bias M
$Uncov = E$ ▷ rules induced
$H = \{\}$
while H is not empty AND $Uncov$ is empty **do**
$re = searchRelevantExample(H, Uncov, constraints, M)$ ▷ search for relevant example using Single-shot ASP Solver
$contraints = translateExamples(B, M, re)$ ▷ Translate example
$Uncov = checkImplaication(B, Uncov, translatedRe, M, constraints)$ ▷ It checks which other examples would be guaranteed to not be covered if the coverage constraints were not satisfied
$H, Uncov = searchProgram(H, Uncov, constraints, M)$ ▷ Program search
end while
return H

In the algorithm [DC12, ML20a], it can be seen that the inputs, as the first hit, are used to search for relevant examples, a practice introduced in ILASP2i. The first method *searchRelevantExample* searches for the examples that allow to identify a particular class of example. In this way ILASP constructs a subset of the examples called the relevant examples, which is often significantly smaller than the full set of examples, but nonetheless still forces ILASP to learn the correct program. The first time the relevant examples are searched in E, while in following rounds they are searched in $Uncov$. The relevant examples are used to extract constraints through the *translateExamples* method, and the result is checked by the *checkImplaication* method. These are components added by ILASP3. The first method translates the relevant examples into a set of

constraints on the solution space and the second method checks which other examples would be guaranteed as uncovered if the coverage constraints were not satisfied Finally, there is a single-shot ASP to solve the problem and find the new program. Unlike all other versions of ILASP, the third one can induce a valid program in one shot because of the constraints, whereas all other versions must use a multi-shot ASP solver. In addition, this method updates the values contained in H and $Uncov$. All this operation are repeated until a valid program is found, i.e. when $Uncov$ empties while H contains the program.
The ILASP systems try to minimise user search bias by receiving only syntactic ones and deriving constraints from the relevant examples [ML20a]. Moreover the ASP approach is based in a preliminary construction of so called skeleton rules that serve as base for final hypotheses. The number of skeleton rules grows exponentially with the number of allowed conditions, but for many significant applications, compressed rules are preferred. The way ILASP works is quite different from all other ILP systems seen so far, but it guarantees an optimal and working result even in real-world contexts where the possibility of noise (errors in the data) is high [ML20a]. Despite the potential of ILASP, there are also weaknesses, in fact ILASP suffers from the high number of examples that decrease efficiency and increase computational cost.

3.7.6 ILP vs ML

As already explained, ILP is a branch of ML, so while ML has interests, applications and algorithms in different fields, ILP is more focused on data mining. ILP allows examples to be mapped and new data to be searched, predicted or induced. Another ML technology widely used for this purpose are decision trees, support vector machine (SVM) and neural network. These branch of ML are quite different from each other. Indeed, while Inductive Logic Programming is the automated learning of rules from examples and background knowledge, decision tree, SVM and neural network are based on statistical inference. In addition, these algorithms are more tolerant to errors and data with missing attributes and are able to learn even from raw data and unlike ILP are not limited by the use of a restricted formalism for knowledge representation and are not forced to define biases that are difficult to formulate. On the other hand, ILP offers several other advantages including an explicit symbolic structure that is human-friendly and thus can be read, understood, and verified; efficient use of data that allows rules to be derived even with a few examples; and support of continuous and transferable learning by allowing learned knowledge to be stored in background knowledge.

3.8 Quality Assessment

As described in section 2.6, we evaluate the optimality of items for the purposes of our research. The parameters evaluated are the number of citations made, the type of paper and any other associated material, the number of times the paper was cited by another paper, and how many research questions the paper answered. The scores are distributed according to the following criterion:

- Number of citation: 0.25 points are awarded for every 15 citations made. If the citation is for a start-set article, an additional 0.1 points are awarded for each citation.
- Paper type: this SLR is just a theoretical base for ILP, for this reason 0.5 points are awarded for theoretical papers, 0.3 for experimental and innovative papers. Extra 0.2 points are awarded if a pseudo-code, a library or an API is present. This is because these resources make paper more valuable.
- Paper impact: if the paper is cited less than 10 times 0.05 points are awarded, 0.2 points are awarded if the citations are between 10 and 20, 0.5 points are awarded if the citations are between 20 and 50, 0.75 points if the citations are less than 100, 2 points if the citations are over 1000, otherwise 1 point is awarded. In addiction 0.1 points are awarded for each citation in start-set.
- Paper influence in SLR: the paper receive 0.25 points for each question it it helped answer.

Paper ID	Number of citations		Paper type		Paper impact		Questions answered					Total
	Citations	Citations from Start-Set	Type of the paper	Extra-resources presence	n of mention	n times cited (in start-set)	Q1	Q2	Q3	Q4	Q5	
[Dar20]	0.4	0	0.5	0	0.05	0	0	0	0	0	0.25	1.2
[Bib19]	1	0	0.5	0	0.05	0	0.25	0.25	0	0	0	2.05
[Kli15]	1	0	0.3	0.2	0.75	0	0	0	0.25	0	0	2.5
[SG15]	0.6	0	0.5	0	1	0	0.25	0	0	0	0	2.35
[A.K15]	0.2	0	0.5	0	0.05	0	0	0.25	0	0	0.25	1.25
[ML20b]	0.4	0	0.3	0.2	0.5	0	0	0.25	0	0	0	1.65
[QZ14]	0.6	0	0.3	0.2	1	0	0	0.25	0.25	0	0.25	2.85
[DC12]	0.2	0	0.5	0.2	1	0	0	0.25	0	0	0.25	2.4
[Mug17]	0.4	0	0.5	0.2	0.05	0	0	0.25	0	0	0.25	1.65
[Mug91a]	0.6	0	0.5	0	2	0.2	0	0.25	0.25	0	0.25	4.05
[Rea94]	2	0	0.5	0	2	0.3	0	0.25	0	0.25	0.25	5.55
[Ker08]	1	0	0.5	0.2	1	0.1	0	0.25	0	0	0	3.05
[SM10]	0.4	0	0.3	0.4	0.75	0.1	0	0	0	0	0.25	2.2
[SM11]	1.6	0	0.5	0	1	0.2	0.25	0.25	0	0	0	3.8
[Sta94]	0.4	0	0.5	0	0.25	0.1	0	0	0	0.25	0	1.5
[Zho17]	0.6	0.1	0.3	0.2	0.25	0	0	0	0.25	0	0	1.7
[Ben18]	0.8	0.1	0.3	0.2	0.5	0.1	0	0	0	0	0.25	2.25
[Mug19]	0.8	0.2	0.3	0.4	0.5	0	0	0	0	0	0.25	2.45
[AC22]	2	0.1	0.5	0	0.25	0	0.25	0.25	0.25	0.25	0	3.85
[Mug99]	0.6	0	0.5	0	1	0	0	0.25	0	0	0	2.35
[Mug95]	0.8	0	0.5	0	2	0.3	0	0	0	0.25	0.25	4.1
[OR03]	0.4	0	0.3	0.2	0.75	0.1	0	0	0	0	0.25	2
[CJ06]	0.4	0	0.5	0	1	0	0	0	0	0	0.25	2.15
[ML20a]	0.6	0	0.5	0.2	0.5	0.1	0	0.25	0	0	0.25	2.4
[AC20]	0.6	0.1	0.3	0.2	0.5	0	0	0	0.25	0	0.25	2.2

References

[AC20] R. Morel & S. Muggleton A. Cropper. Learning higher-order logic programs. *Machine Learning*, 109:1289–1322, 2020.

[AC22] R. Evans & S. Muggleton A. Cropper, S. Dumančić. Inductive logic programming at 30. *Machine Learning*, 111:147–172, 2022.

[A.K15] S. Tyagi & A.Kowcika. Best practices generation for storage area network: A review. *Proceedings of the International Conference : Computational Systems for Health & Sustainability*, 2015.

[Ben18] P. Schüller & M. Benz. Best-effort inductive logic programming via fine-grained cost-based hypothesis generation. *Mach Learn 96*, 2018.

[Bib19] M. Kastratia & M. Bibaa. Statistical relational learning: A state of the art review. *Journal of Engineering Technology and Applied Sciences*, 2019.

[BK07] Stuart M. Charters Barbara Kitchenham. Guidelines for performing systematic literature reviews in software engineering. 2007.

[CH98] C. D. Page & M. J. E. Sternberg C. H.Bryant, S. H. Muggleton. Combining active learning with inductive logic programming to close the loop in machine learning. 1998.

[CJ06] J. R. Quinlan & R. M. Cameron-Jones. Foil: A midterm report. *ILP 2006*, 2006.

[Dar20] A. Darwish. Reconciling between symbolic and connectionist ai. *Research Gate*, 2020.

[DC12] & E. Lupu D. Corapi, A. Russo. Inductive logic programming in answer set programming. *ILP 2011*, LNAI 7207:91–97, 2012.

[FR14] E. Bellodi & R. Zese F. Riguzzi. A history of probabilistic inductive logic programming. 2014.

[JL15] & G. P. Goodwin Johnson-Laird, S. S. Khemlani. Logic, probability, and human reasoning. 2015.

[Ker06] K. Kersting. An inductive logic programming approach to statistical relational learning. *AI Communications 19*, page 389–390, 2006.

[Ker08] L. De Readt & K. Kersting. Probabilistic inductive logic programming. *Probabilistic ILP 2007*, 4911:1–27, 2008.

[Kin99] A. Srinivasan & R. D. King. Feature construction with inductive logic programming: A study of quantitative predictions of biological activity aided by structural attributes. *Data Mining and Knowledge Discovery 3*, page 37–57, 1999.

[Kin04] R. D. King. Applying inductive logic programming to predicting gene function. *AI Magazine Volume*, 25(1), 2004.

[Kit10] E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. *AAIP 2009*, 5812:50–73, 2010.

[Kli15] J. Furnkranz & T. Kliegr. A brief overview of rule learning. *Springer International Publishing Switzerland*, 2015.

[MD19] G. Kunapuli K. Kersting & S. Natarajan M. Das, D. Singh Dhami. Fast relational probabilistic inference and learning: Approximate counting via hypergraphs. *Association for the Advancement of Artificial Intelligence*, 2019.

[ML20a] A. Russo & K. Broda M. Law. The ilasp system for inductive learning of answer set programs. *ILASP Limited*, 2020.

[ML20b] E. Bertino K. Broda & J. Lobo M. Law, A. Russo. Fastlas: Scalable inductive logic programming incorporating domain-specific optimisation criteria. 2020.

[Mug91a] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8:295–318, 1991.

[MUG91b] Stephen MUGGLETON. Inductive logic programming. *New Generation Computing, OHMSHA*, 8:295–318, 1991.

[Mug95] S. Muggleton. Inverse entailment and progol. *New GenerationComputing*, (13):245–286, 1995.

[Mug99] S. Muggleton. Inductive logic programming: Issues, results and the challenge of learning language in logic. *Artificial Intelligence*, 114:283–296, 1999.

[Mug17] S. H. Muggleton. Meta-interpretive learning: Achievements and challenges. *RuleML+RR 2017*, LNCS 10364:1–6, 2017.

[Mug19] A. Cropper & S. Muggleton. Learning efficient logic programs. *Mach Learn 108*, page 1063–1083, 2019.

[NSO16] H. Singmann & K. C. Klauer N. Skovgaard-Olsen. The relevance effect and conditionals. 2016.

[OR03] K. Broda & A. Russo O. Ray. Hybrid abductive inductive learning: A generalisation of progol. *ILP 2003*, 2835:311–328, 2003.

[QZ14] J. M. Patel & D. Page Q. Zeng. Quickfoil: Scalable inductive logic programming. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[Rae01] K. Kersting & L. De Raedt. Towards combining inductive logic programming with bayesian networks. *ILP 2001*, 2157:118–131, 2001.

[RDK96] A. Srinivasani & M. J. E. Sternberg R. D. King, S. Muggleton. Structure-activity relationships derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity by inductive logic programming. *Proc. Natl. Acad. Sci.*, 93:438–442, 1996.

[Rea94] S. Muggleton & L. De Readt. Inductive logic programming: Theory and methods. *LOGIC PROGRAMMING*, 19(20):629–679, 1994.

[Rig12] E. Bellodi & F. Riguzzi. Learning the structure of probabilistic logic programs. *ILP 2011*, 7207:61–75, 2012.

[Rus15] S. Russell. Unifying logic and probability. *Communications Of The Acm*, 2015.

[SG15] E. Kitzelmann S. H. Muggleton U. Schmid & B. Zorn S. Gulwani, J. Hernández-Orallo. Inductive programming meets the real world. *Communications Of The Acm*, 2015.

[SM94] Luc DE RAEDT Stephen MUGGLETON. Inductive logic programming: Theory and methods. *LOGIC PROGRAMMING*, 19, 20:629–679, 1994.

[SM10] J. Santos & A. Tamaddoni-Nezhad S. Muggleton. Progolem: A system based on relative minimal generalisation. *ILP 2009*, 5989:131–148, 2010.

[SM11] D. Poole-I. Bratko P. Flach K. Inoue & A. Srinivasan S. Muggleton, L. De Raedt. Ilp turns 20. 2011.

[Sta94] I. Stahl. On the utility of predicate invention in inductive logic programming. 1994.

[Woh14] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. 2014.

[Woh16] Claes Wohlin. Second-generation systematic literature studies using snowballing. 2016.

[Zho17] W. Dai & Z. Zhou. Combining logical abduction and statistical induction: Discovering written primitives with human knowledge. *Association for the Advancement of Artificial Intelligence*, 2017.