Conducting Inductive Logic Programming Directly in Database Management Systems

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Akshay Reddy Koppula

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Richard Maclin

July 2015

Acknowledgements

I would like to thank my thesis advisor, Dr. Richard Maclin for sharing his immense knowledge and motivating me. His patience in answering all the questions I had and his guidance helped me in all the time of research and writing of this thesis.

I would like to thank my committee members, Dr. Haiyang Wang and Dr. Steven Trogdon for their support. I would like to thank Dr. Peter Willemsen for improving my coding skills. I would like to thank the computer department science faculty, Dr. Ted Pedersen, and Dr. Hudson Turner for teaching me the important concepts of computer science that will help me in the future. I would like to thank Lori Lucia and Clare Ford for providing help and support in times of need.

I would like to thank my fellow graduate student, Ravikanth Repaka, for sharing his knowledge, skills, and experience throughout the thesis. I would like to thank my family and friends for their continuous support and encouragement.

Dedication

I dedicate this thesis to my parents, Vishnudev Reddy and Sunitha Koppula, and my brother, Dheeraj Reddy whose words of encouragement and push for tenacity ring in my ears.

Abstract

Inductive logic programming (ILP) is a research area formed at the intersection of machine learning and logic programming. Given a set of background knowledge as well as positive and negative examples of a concept, an ILP system attempts to learn rules that cover all the positive examples and none of the negative examples by using the background knowledge. Over the years, ILP is being used extensively in medical applications. Existing ILP systems are implemented in Prolog, using first-order logic. But, Prolog does not integrate well with database systems, where a lot of the data of interest is stored. Prolog is also not often used in business applications.

This thesis presents a novel approach of storing the facts (background knowledge, examples) required for ILP in databases and using Java for easy access and retrieval of the stored knowledge. Since most of the ILP machine learning data sets can be stored easily in databases, this approach provides an easier to use technique. Facts are stored in the form of tables in database and rules are stored as database views by using a database join on the multiple arguments in a fact. A Sequential covering algorithm that uses the best-first search approach to learn rules for ILP problems is implemented in this thesis. The results obtained on two real-world test data sets by using this approach are compared with traditional systems. The accuracy of the system presented in this thesis is on par with the accuracy of the traditional systems. These results are very assuring and the system provides an easy-to-use approach for the ILP users.

Contents

List of Tables

List of Figures

1 Introduction

Inductive Logic Programming (ILP) [13] is a subfield of machine learning, that is widely used to learn interesting patterns. It has quickly developed into a very active research field. Many real-world datasets that have been applied to ILP have benefited from the ILP systems. The ability of these systems to accommodate background knowledge is a key. Most of the ILP systems are implemented in Prolog, using first-order logic. However, these systems are restricted in the sense that they generally cannot use datasets that are stored in relational databases.

Many real world datasets will be in a database format, which makes it hard to use them in the traditional ILP systems, which are usually implemented in Prolog. To overcome this obstacle, this thesis proposes a concept of using database as backend and a programming language such as Java to implement an ILP system. This provides an easy-to-use technique for ILP users as databases are traditionally used for many different programs and a user does not feel out of home while working on them. The use of the background knowledge enables the user to develop a suitable problem representation and to introduce problem specific constraints into the learning process. The ability to store this background knowledge in relational databases makes it easier to develop and solve a problem.

In this thesis, a relational database is used to store the facts, positive examples and, negative examples that are required for an ILP system to generate a hypothesis. This thesis presents a unique way of storing facts in the database as tables, rules as database views. The predicates in the rules can have relations with one another. These relations are addressed using database joins. So, rules are stored as views by joining the common arguments in

predicates.

In order to learn new rules, the system presented in this thesis uses a sequential covering algorithm. A best-first search mechanism is employed in order to limit the search process by focusing on the 'best' possible solutions first. This algorithm uses the data stored in the database and also creates different tables that store the positive as well as negative examples. All the data required, such as background knowledge, positive examples, and negative examples are stored in the database and this makes it easy to score a rule that has been generated. The system is based on communications between a database and Java to learn new rules.

This paper talks in detail about the background of ILP including definitions, techniques, and implementations of ILP. These topics are covered in Chapter 2. Chapter 3 talks in detail about the implementation of the system with a few examples. Chapter 4 talks about the results obtained when this system is tested on two real-world datasets. Apart from discussing about the results obtained by this system, it also compares those results with the results obtained from traditional systems. Chapter 5 talks about the conclusion and future work of this thesis.

2 Background

This chapter discusses in detail the basic notions necessary to understanding ILP. It starts with the definition of ILP and explains the concept of ILP with different examples. Different characteristics and techniques of ILP will be discussed. Further, different ILP systems and comparison between those systems will be shown.

2.1 Inductive Logic Programming

Inductive Logic Programming (ILP) [15] is a combination of machine learning and logic programming. Definition of ILP and the terminology used in ILP are explained clearly in Example 2.1.

Example 2.1

Consider the case of credit approval. A credit agency has to consider many different options in order to approve a loan or a line of credit for a person. For example, the current income of a person, age, male, female, loan amount are some of the options that a credit agency might consider while approving a loan. These options are called as *predicates* in ILP. These predicates with arguments are called as *facts*. Facts state the relationship that holds between arguments [23].

current_income(p1, 10000)

The example given above is a fact, stating that the current income of a person *p1* is

$10000 per month. In this example, current_income, *p1* are called as *atoms*. The number 10000 is called a *constant*. In general, a *constant* is either an atom or a number.
Example of some other facts to be considered for credit approval are:

male(p1)

age(p1,30)

loan_amount(p1, 100000)

female(p2)

age(p2, 25)

loan_amount(p2,50000)

The combination of all the facts is called the *background knowledge* [24]. The credit agency might have a set of previous instances where the borrower has paid back the loan with interest. These instances are called as *positive examples*, as the bank is happy with the loan. For example, if a person *p1* has paid back the loan successfully then this person is listed as a positive example.

The credit agency may also have instances where the borrower did not pay back the loan. These instances are called as *negative examples*. For example, if a person *p2* did not pay back the loan, then this person is listed as a negative example.

Consider a scenario with the following Background Knowledge, Positive examples and, Negative example.
Background Knowledge (B):

male(p1) age(p1,30), loan_amount(p1,100000), current_income(p1,10000),

male(p2), age(p2,23), loan_amount(p2,100000), current_income(p2,2000),

male(p3), age(p3,29), loan_amount(p3,100000), current_income(p3,5000),

male(p4), age(p4,28), loan_amount(p4,100000), current_income(p4,8000).

Positive Examples(E^+):

credit_approve(p1),

credit_approve(p4).

Negative Examples(E^-):

credit_approve(p2),

credit_approve(p3).

ILP is an automated system, which when given a set of training examples and background knowledge, generates a hypothesis that covers all the positive examples, but none or few of the negative examples. The training examples, background knowledge and the hypothesis are all represented in the logic form with training examples and background knowledge mostly being logical facts.

For a logical representation, let us assume that there is a new predicate as follows:

greater_than(X,Y).

This fact states that *X* is greater than *Y*. Here, *X* and *Y* are called as *variables*. A variable [23] is used to refer to an unspecified individual. So, any constant or atom can take

the place of X or Y.

Using B, E^+,E^- , the following hypothesis (H) can be generated as follows:

$$credit\_approve(X) : -male(X), age(X, A), greater\_than(A, 25), \\ loan\_amount(X, 100000), current\_income(X, B), \\ greater\_than(B, 5000) \tag{2.1}$$

Equation 2.1 is called as *Rule*. Rules [9] allow to make conclusions about the world. For Example, the Rule in Equation 2.1 will either evaluate to true or false based on different values of X.

H consists of only one rule in this case. A rule consists of a head and a body which is identified as follows in Equation 2.1:

credit_approve(X) is called the *head* of the rule (predicate to the left of ':-'),
Everything that is towards the right of the symbol ':-' is called the *body* of the rule.

A *clause* is defined as a fact or rule.
Equation 2.1 can be read as follows:

> *Approve credit for X if X is a male and the age of X is greater than 25 and the loan amount requested by X is 100000 and the current income of X is greater than 5000, where X can be any person.*

The predicates with new variables such as X in Equation 2.1 are called as *literals*. This rule in first-order logic is a conjunction of all the literals which means that ',' in rule 2.1

stands for ‘AND’. So, the rule stands true only if all the literals are true.

Equation 2.1 covers both the positive examples which is explained as follows:
Break the rule into literals and replace X with p1, then the literals will be as follows:

male(X) - male(p1) - True

age(X,A), greater_than(A,25) - age(p1, 30) - True

loan_amount(X,100000) - loan_amount(p1,100000) - True

current_income(X,B), greater_than(B,5000) - current_income(p1,10000) - True

All the above listed literals are converted into facts when X is substituted with *p1*. Since, all the literals when substituted with values become facts, the rule stands true in case of *p1*. Similar is the case when X is substituted with *p4*. This process of matching the items with variables in order to determine if a clause is met by a set of facts is called *Unification*.

The process of total mapping from variables to terms is called *Substitution* [15].
Equation 2.1 does not cover any negative examples which is explained as follows by replacing *X* with *p3*:

male(X) - male(p3) - True

age(X,A), greater_than(A,25) - age(p3, 29) - True

loan_amount(X,100000) - loan_amount(p3,100000) - True

current_income(X,B), greater_than(B,5000) - current_income(p3,5000) - False.

The last literal does not satisfy when X is matched with *p3* and since the rule is a conjunction of literals, it does not hold true.

The generated Hypothesis is called *Complete* if all the positive examples are covered and is called *Consistent* if no negative examples are covered. H is both complete and consistent [7].

ILP is described as a system that learns rules automatically. There are many different mechanisms for an ILP system to learn rules from the examples and background knowledge. A background on the different types of systems that are present is described in the further sections.

2.1.1 Recursion in ILP

The facts provided for an ILP setting might contain a pattern that repeats itself again and again. This pattern can be captured by the recursion [4]. Recursion is better explained with the following example.

Example 2.2

Ancestor is a good recursion example. *Ancestor(X,Y)* means that a person *X* is ancestor of *Y*. Consider the following background knowledge and, positive examples.

Background Knowledge (B):

Parent(Jack,Diana)

Parent(Diana, Bob)

Parent(Diana, Kate)

Parent(Bob, Linda)

Positive Examples(E^+):

Ancestor(Jack, Diana)

Ancestor(Jack,Bob)

Ancestor(Jack,Linda)

Ancestor(Jack, Kate)

Ancestor(Diana, Bob)

Ancestor(Diana, Linda)

Ancestor(Diana, Kate)

Ancestor(Bob, Linda)

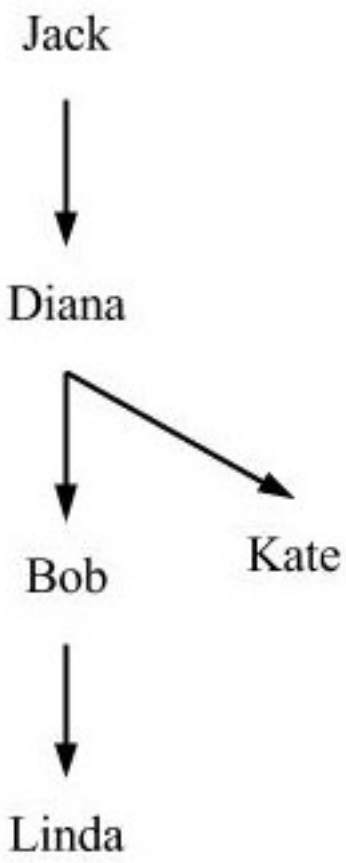

Figure 2.1: Illustration of Parent relation for Ancestor rule

Figure 2.1 gives the parent relation in the form of a network. In this figure, relations are depicted by arrows. is a parent of *Diana* and hence the arrow points from *Jack* towards

Diana. In this case, *Jack* is an ancestor of *Diana*, *Bob*, *Kate*, *Linda*. *Bob* is a parent as well as ancestor of *Linda*. So, every relation along with the sub-relation of that relation is an ancestor relation.

This kind of relations can only be represented using recursion. Then hypothesis for Ancestor can be represented as follows:

$$Ancestor(X, Y) : -Parent(X, Z), Ancestor(Z, Y) \tag{2.2}$$

$$Ancestor(X, Y) : -Parent(X, Y) \tag{2.3}$$

The recursion hypothesis will have a base case rule. A base case rule is necessary to terminates the recursion. Equation 3.2 is the base case rule. Let us check the completeness of this recursion rule:

Ancestor(Jack, Linda)?

Using the substitution method, here X is replaced by Jack and Y is replaced by *Linda*. Check if the base case rule satisfies this relation. The base case rule does not satisfy this relation as there is no fact such as *Parent(Jack, Linda)*.

Ancestor(Jack, Linda) :- Parent(Jack, Z), Ancestor(Z, Linda)

There is only one possibility for *Jack* to be a Parent. Therefore, Z should be *Diana*.

Ancestor(Jack, Linda) :- Parent(Jack, Diana), Ancestor(Diana, Linda)

Again, the recursive rule *Ancestor(Diana, Linda)* is called. In this case, the X in Equation 2.2 is substituted by Diana and Y in 2.2 is substituted by *Linda*.

Ancestor(Diana, Linda) :- Parent(Diana, Z), Ancestor(Z, Linda)

There are two possibilities for Z. Z can be replaced with either *Bob* or *Kate*. First, let us replace Z with *Bob*.

Ancestor(Diana, Linda) :- Parent(Diana, Bob), Ancestor(Bob, Linda)

The recursive rule calls itself and here, X is replaced by *Bob* and Y is *Linda*.

Ancestor(Bob, Linda) :- Parent(Bob, Linda)

The base case rule is reached and recursion is stopped. There is a fact *Parent(Bob, Linda)* in the background knowledge which makes the statement true. This rule returns true and hence, the all the recursions return true. Therefore, *Ancestor(Jack, Linda)* returns true.

2.2 Characteristics of Inductive Logic Programming

ILP systems are known to have four different characteristics [3] which are listed as follows:

- Incremental/Non-Incremental
- Interactive/Non-Interactive
- Single/Multiple Predicate Learning
- Theory Revision.

These four characteristics are explained as follows:

Incremental/ Non-Incremental: In Non-Incremental ILP, all the examples are given as input before the start of the learning process. In Incremental ILP, examples are given as input one by one during the learning process.

Interactive/Non-Interactive: In Interactive ILP systems, the learner keeps asking the user questions that help in determining whether the intended ILP system is being achieved or not. If the learner does not ask any question then it is Non-Interactive. Clearly, Interactive ILP systems employ incremental model. Most of the ILP systems are Non-Interactive.

Single/Multiple Predicate Learning: ILP systems can learn either a single predicate or multiple predicates.

Theory Revision: If an ILP system accepts an existing hypothesis as input for learning then it called Theory Revision. This is basically an incremental multiple predicate learner. These systems are called Theory Revisors.

ILP systems usually combine more than one of the above dimensions to develop ILP systems. There are two basic ILP systems. First, non-incremental and non-interactive learners that learn single predicates from start which are called *Empirical ILP* systems. Second, incremental and interactive theory revisors that learn multiple predicates which are called *Interactive ILP* systems.

2.3 Hypothesis Space

ILP deals with learning rules. Learning rules requires a system to search the hypothesis space [7]. Searching the hypothesis space means searching different possible combinations of literals and finally reaching a point that satisfies the required criterion (cover all positive examples but no negative examples). An ILP system can be differentiated from

other systems by the way it searches the hypothesis space. This section talks about the θ-subsumption [16] lattice that gives a structure of the hypothesis space.

θ is a substitution mechanism that substitutes variables with terms. Substitution is defined in Example 2.1. C1θ-subsumes C2 if there exists a substitution θ such that C1$\theta \subseteq$C2 [18]. This concept is called θ-subsumption. Example 2.3 explains the concept of subsumption in detail.

Example 2.3

Consider the predicates in Example 2.1. Let C1 be the following clause,

C1: credit_approve(X) :- Male(X).

Apply as substitution θ- X/p1 to every literal in C1 returns the following Clause,

$$C1 : credit\_approve(p1) : -Male(p1) \tag{2.4}$$

Let C2 be the following clause,

C2: credit_approve(X) :- Male(X), age(X, 25).

Now, it is possible to say that C1θ-subsumes C2 under the empty substitution set θ - □. Here C1θ subsumes *C2* because *C1*θ is a proper subset of *C2*. All the literals present in *C2* are present in *C1* and since θ is an empty substitution, *C1*θ-subsumes C2.

C1θ-subsumes *C2* if the substitution is as follows:

θ - X/p1

$$C2 : credit\_approve(p1) : -Male(p1), age(p1, 25) \tag{2.5}$$

Here C1θ subsumes *C2* because *C1*θ is a proper subset of *C2*. It can be observed that Equation 2.4 is a subset of Equation 2.5.

θ-subsumption is a concept that talks about generality of a clause. A clause *C1* is at least as general as *C2* if *C1*θ subsumes *C2*. The concept of generality is better explained in section 2.4.

2.4 ILP Techniques

Hypothesis can be derived either from a bottom-up or top-down search of the hypothesis space [7]. Top-down search techniques start the search from generalization and then go to specific examples to check if the hypothesis formed is complete and consistent.

Example 2.4 Consider the following Background knowledge, Positive examples and Negative examples with respect to finding the relation *Son* using the relations Parent and Male. The predicates are declared as follows:

Son(X,Y) - X is a son of Y

Parent(X,Y) - X is a parent of Y

Male(X) - X is a male

Background Knowledge (B):

Parent(George, Bob),

Parent(Bob,Lisa),

Parent(Alice, Jack),

Parent(George, Alice),

Male(Bob),

Male(Jack),

Male(George).

Positive Examples(E^+):

Son(Jack, Alice),

Son(Bob, George)

Negative Examples(E^-):

Son(Lisa, Bob),

Son(Alice, George)

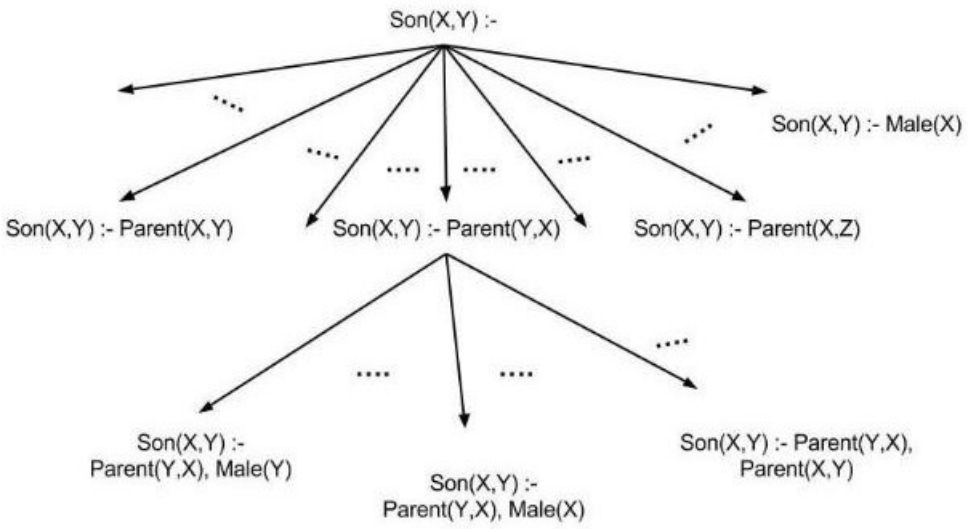

Figure 2.2: A Tree explaining the top-down approach of ILP.

Figure 2.2 displays the tree approach that the top-down technique uses. Initially, the head of the rule is added to an empty rule. This rule is then extended by adding all the possible literals in the body of the rule. The best rule among all of these is taken and is extended to next level to add further literals. This process of adding literals continues until a hypothesis is generated that covers no negative examples.

Using B, E^+,E^- , the following hypothesis (H) can be generated as follows:

Son(X,Y) :- Parent(Y,X), Male(X)

The top-down techniques are also called as specialization techniques. These techniques start from most general clause and keeps adding literals to this clause until the hypothesis covers no negative examples. Specialization techniques use θ-subsumption to make a general clause to a less general clause. With the help of θ-subsumption, more literals can be added to the clause while confirming that the clause is becoming more specific.

Bottom-up search techniques work in the opposite way of top-down search techniques. They start from the training examples and then generalize the examples to result in a hypothesis. These techniques are called generalization techniques. There are two generalization techniques which are explained in section 2.4.1

2.4.1 Generalization Techniques

Generalization techniques follow the bottom-up search mechanisms to generate hypothesis [6]. These techniques start from the most specific clause that covers a given example and then generalizes the clause until it cannot be further generalized.

Least general generalization (lgg) [18] is a technique that assumes that generalization of a set of clauses which are true is also true. If two clauses c1 and c2 are true, then most likely lgg(c1,c2) is also true. Following example gives an insight into lgg.

Example 2.5

Consider the facts provided in Example 2.4

Let, clause *c1 = Son(Jack,Alice) Parent (Alice,Jack), Male(Jack)* and

clause *c2 = Son(Bob,George) Parent(George,Bob), Male(Bob)*

The lgg of *c1* and *c2* will be :

lgg(c1,c2) = Son(X,Y) Parent(Y,X), Male(X)

where X = *lgg(Jack,Bob)* and Y = *lgg(Alice, George)*

Here a new variable is introduced in common places. For example, the first argument in *Son*, the second argument in *Parent*, the sole argument in *Male* are shared by *Jack* and *Bob* in clauses *c1* and *c2* respectively. Therefore, a new variable, say *X*, can be introduced in their place. Similarly, a new variable *Y* is also introduced in the place of *Alice* and *George*. Since, clause *c1* is true and *c2* is also true lgg assumes that *lgg(c1,c2)* is also true.

Relative Least General Generalization

Relative least general generalization (rlgg) is an extension of lgg. Rlgg of two clauses *c1* and *c2* is the *lgg(c1,c2)* with respect to background knowledge B [6]. Given K, as a conjunction of facts in the background knowledge, rlgg of two positive Examples *P1*, *P2* will be as follows:

$$rlgg(P1, P2) \leftarrow lgg((P1 \leftarrow K), (P2 \leftarrow K)) \tag{2.6}$$

Example 2.6

Consider the facts, background knowledge, provided in Example 2.1.

Given the positive examples *Son(Jack, Alice)* as *p1*, *Son(Bob, George)* as *p2* along with the

background knowledge, the rlgg of *p1* and *p2* is :

$$rlgg(p1, p2) \leftarrow lgg((p1 \leftarrow K), (p2 \leftarrow K)) \tag{2.7}$$

where K is

Parent(George, Bob) $\wedge$ Parent(Bob,Lisa) $\wedge$ Parent(Alice, Jack) $\wedge$ Parent(George, Alice) $\wedge$ Male(Bob) Male(Jack) $\wedge$ Male(George).

For easy computation purposes, let Parent be represented as P, Male as M, Son as S, George as g, Bob as b, Alice as A, Jack as J, Lisa as L.

So, Now K will be P(g,b) $\wedge$ P(b,l) $\wedge$ P(a,j) $\wedge$ P(g,a) $\wedge$ M(b) $\wedge$ M(J) $\wedge$ M(g). Rlgg can be computed as follows:

$$\begin{aligned} S(V_{b,j}, V_{g,a}) \leftarrow\ & P(g,b) \wedge P(b,l) \wedge P(a,j) \wedge P(g,a) \wedge M(b) \wedge M(J) \wedge M(g) \wedge P(V_g, V_{b,a}) \\ & \wedge P(V_{g,b}, V_{b,l}) \wedge P(V_{g,b}, V_{a,l}) \wedge P(V_{g,a}, V_{a,j}) \wedge P(V_{g,a}, V_{b,j}) \wedge P(V_{b,a}, V_{l,j}) \\ & \wedge P(V_b, V_l) \wedge P(V)a, V_j) \wedge M(V_{g,b}) \wedge M(V_{b,j}) \wedge M(V_{g,j}) \end{aligned} \tag{2.8}$$

Where $V_{x,y}$ stands for rlgg(x,y) for each x and y in background knowledge. After eliminating the unnecessary literals, the final clause will be

$$S(V_{b,j}, V_{g,a}) \leftarrow P(V_{g,a}, V_{b,j}), M(V_{b,j}) \tag{2.9}$$

where ',' represents conjunction.

Equation 2.9 can be rewritten as S(X,Y) $\leftarrow$ P(Y,X), M(X).

This is the relative least general generalization technique that uses bottom-up approach.

Inverse Resolution

Inverse Resolution is another generalization technique [14] where the resolution works backwards. This idea of inverse resolution was first introduced in [19]. With inverse resolution, rules can be learned from examples, background knowledge. It is used in the ILP system 'FOIL'.

In these rules, the lower case letters are atoms and upper case letters are the combination of atoms. The absorption and identification steps invert a single resolution step which is shown as a *V* operator. Intra and Inter constructions introduce a new predicate which is shown as a *W* operator [15].

The connected resolution can be depicted as a *V* operator which represents a single resolution inference. The resolution is able to resolve the base clause with the help of the clause at other arm and the clause at the base of the operator *V*. In other words, the absorption operator constructs C2 given C1 and C3. The identification operator constructs C1 given C2 and C3. These two operations together produce operator *V*. The *V* operator is depicted in Figure 2.3 [15].

The *W* operator is a combination of two *V* operators. Clause C2 is the common clause for both the *V* operations. The Clauses C1, C2, C3 are constructed given B1 and B2. Since, it is already known that a *W* operator introduces a new predicate symbol, it can be observed that the predicate symbol 'q' cannot be found in B1 and B2.The *W* is shown in Figure 2.4 [15].

This tree shown in Figure 2.5 a bottom-up approach where *B1* and *E* combine to produce *C1*. In this case, *Bob* is substituted with *X*. Later *B2* and *C1* are combined to produce *C2*. In this case, *George* is substituted with *Y* to produce a generalized clause. This is the inverse resolution technique that uses the four rules of absorption, identification, Intra constructions, and Inter constructions to produce a clause.

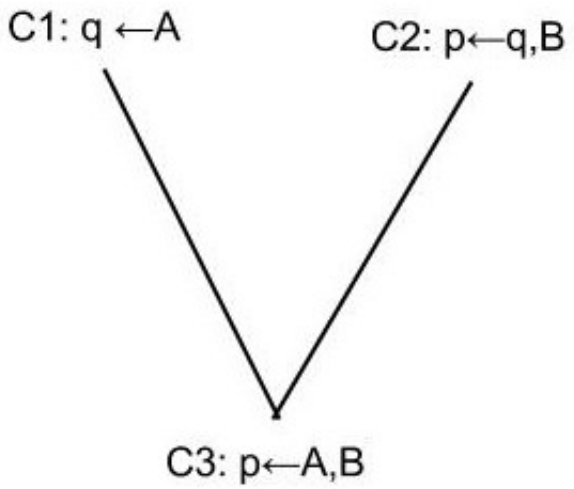

Figure 2.3: V Operator

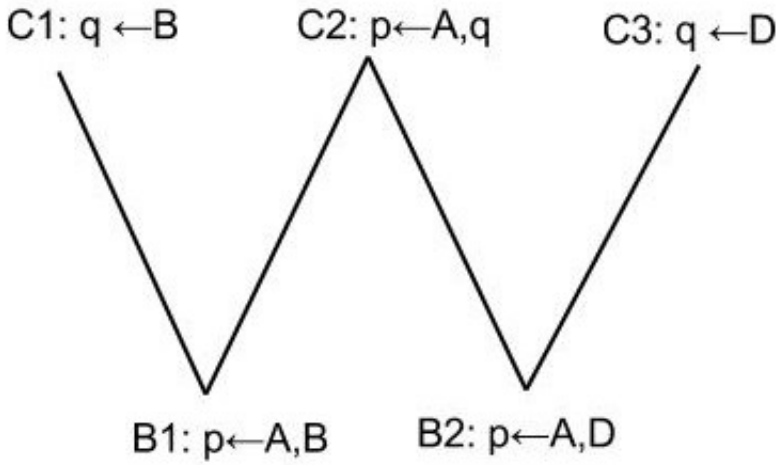

Figure 2.4: W operator

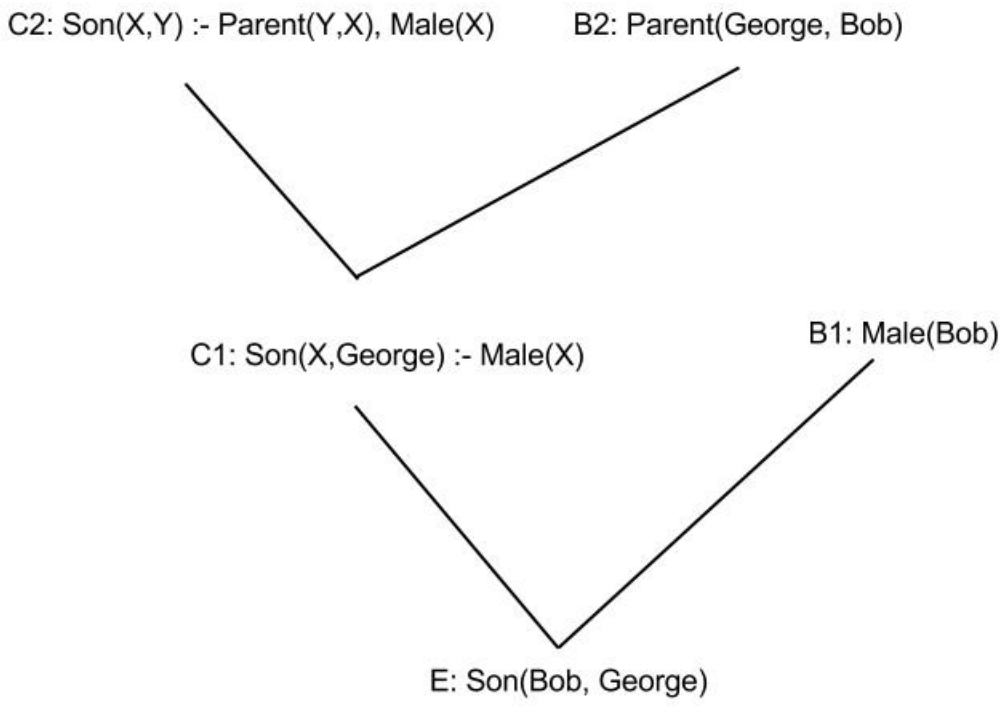

Figure 2.5: Resolution of a rule

2.5 ILP Systems

ILP systems are known to learn new rules. Learning is a process of classifying given set of attributes into a mutually exclusive and exhaustive classes. It is a task of predicting the class of an unseen object based the given attributes. There are three different types of learning [10].

First is supervised learning, where the new rules are learned using the knowledge deducted from the given training data. Here, the algorithm will learn the target provided for each input and adjusts itself in such a way that it is more likely to provide the closest possible target for an unclassified input.

Second is unsupervised learning, where the learner is not provided with any training

data. The algorithm has to work in ways such that it is able to represent the given input in the form of clusters or categories.

Third is reinforcement learning, where the learner receives feedback on learned rules. It is similar to supervised learning except that the reinforcement learning will know the situation where the approach is inappropriate.

2.5.1 Sequential Covering Algorithm

Sequential covering algorithm is a rule-based algorithm that follows the process of building a hypothesis theory that covers all the positive examples with as few negative examples as possible [11]. It follows the learn-one rule approach. In learn-one rule, a rule is learnt with high accuracy with any possible coverage. The Sequential covering algorithm will delete the positive examples covered by this rule and will repeat the process of learn-one rule again. The algorithm for Sequential covering [20] will be as follows:

Algorithm 1 Sequential Covering Algorithm

1: **procedure** Sequential_Covering($Target\_rule, Attributes, Examples, Threshold$)
2: $Learned\_rules \leftarrow Empty\_Set$
3: $Rule \leftarrow Learn\_one\_rule(Target\_rule, Attributes, Examples)$
4: **while** $Performance(Rule, Examples) > Threshold$ **do**
5: $Learned\_rules \leftarrow Learned\_rules + Rule$
6: $Examples \leftarrow Examples - examples\_correctly\_classified\_by\_the\_Rule$
7: $Rule \leftarrow Learn\_one\_rule(Target\_rule, Attributes, Examples)$
8: $Learned\_rules \leftarrow sorted\_Learned\_rules$
9: **return** $Learned\_rules$

Algorithm 1 gives the algorithm of Sequential Covering. It takes four arguments as input. Target rule is the rule to be generated, Attributes are the facts that will be used to generate a rule, Examples are positive and, negative examples, and Threshold is the Performance upto which the process has to continue. Initially, the learned are empty. Sequential Covering will call *Learn One Rule* method which will return the best possible rule. After a

Algorithm 2 Learn One Rule Algorithm

```
1: procedure Learn_one_rule(Target_rule, Attributes, Examples, Threshold)
2:     Pos ← Positive_Examples
3:     Neg ← Negative_Examples
4:     if Pos then
5:         NewRule ← Most_General_Rule
6:         NewRuleNeg ← Neg
7:         while NewRuleNeg do
8:             Literals ← Generate_Literals()
9:             Best_literal ← Best_Performance_literal()
10:            NewRule ← NewRule + Best_literal
11:            NewRuleNeg ← Negative_NewRule
12:    return NewRule
```

Algorithm 3 Foil Algorithm

```
1: procedure Foil_Learn_Rules(Target_Predicate, Predicates, Examples)
2:     Pos ← Positive_Examples
3:     Neg ← Negative_Examples
4:     while Pos do                                   ▷ LearnaNewRule
5:         NewRule ← Most_General_Rule
6:         NewRuleNegatives ← Neg
7:         while NewRuleNegatives do
8:             Literals ← Generate_Literals()
9:             Best_literal ← Best_Performance_literal()
10:            NewRule ← NewRule + Best_literal
11:            NewRuleNegatives ← Negative_NewRule
12:        Learned_rules ← Learned_rules + NewRule
13:        Pos ← Pos − examples_correctly_classified_by_NewRule
14:    return Learned_Rules
```

rule is returned by *Learn One Rule*, the positive examples covered by this rule are deleted.

Learn One Rule follows a top-down approach to learn rules. It starts from the most general rule and keeps adding literals to the rule until a specific rule achieves the highest accuracy. There are two functions in the *Learn One Rule* algorithm. The *Generate_Literals()* function will return all the possible literals for that rule. *Best_Performance_Literal()* function will have the mechanism of rating rules based on their positive and negative score.

Beam search is used to search for new literals to add. It is a heuristic search algorithm

where only a set of predefined threshold limit of rules with high accuracy are extended. Only these rules will be selected and a new literal will be added to these rules. Beam search is an optimization of best-first search.

2.5.2 FOIL

First order Inductive Learner (FOIL) is proposed by Quinlan [19]. FOIL employs an algorithm that tries to find a rule that covers as many positive examples as possible while covering no negative examples. Adds it to the existing hypothesis, which is initially empty. Removes the positive examples covered by the new rule. This process keeps repeating until all the positive examples are covered. When all the positive examples are covered, the rules are reviewed to remove any redundancy and re-ordered.

FOIL uses a top-down approach meaning that it starts from the most generalized rule and continues to a more specific rule. It starts from a generalized rule and keeps adding literals until no negative examples are covered. A rule can be too general if it covers negative examples, and too specific if no positive examples are covered. Among all the possible specifications, FOIL employs a heuristic evaluation based on which, it selects the best possible specifications. FOIL hill climbs using this heuristic to cover all the positive examples.

A rule is made up of literals. FOIL makes a rule specific by adding literals to the body of the rule. This process stops when no negative examples are covered by the rule or when the rule becomes too complex. In this process of adding literals, new variables that are not present on the left-hand side of the rule can also be introduced. This process of adding literals to the rule can be summarized as follows:

- Initialize the rule with the target relation (head of the rule) and the training set, T, to the positive examples that are not covered by any of the previous rules and also the negative examples.

- While T contains negative examples and is not too complex, repeat the following procedure:
  - Find a literal to add to the body of the rule.
  - Form a new training set, T1 by adding all the examples that are covered by the rule. If any new variables are introduced by the rule, then all the examples covered by that new variable shall also be added to T1.
  - Replace T with T1.
- Prune the rule by removing any unnecessary literals.

FOIL uses a greedy approach in adding literals, in a sense that if a literal is added to the rule, then other literals are not looked at. Hence, adding an appropriate literal is a key. FOIL uses two mechanisms to look out for an appropriate literal.

- A literal must help to remove any negative or unwanted examples from the training set. These type of literals are called gainful literals.
- A literal must introduce new variables that are useful for future literals. These type of literals are called determinate literals.

If a literal satisfies any of the above two mechanisms, then it is determined appropriate and is added to the body of the rule. FOIL employs a greedy search mechanism to search for literals. It uses a backtracking mechanism to get back to a point that provided a better gain instead of searching ahead with literals that provide no further gain. This mechanism is achieved by using checkpoints when a literal added to the existing rule provides only some improvement to the whole rule compared to another literal. A small number of checkpoints are allowed and above that, the search will be restarted from the best checkpoint. This process occurs infrequently, since the greedy search is a good mechanism to find a rule. The search space of literals is limited to the following constrictions:

- The literal must contain at least one existing variable. This variable can be a new variable introduced by a previous literal.
- If the new literal is same as the head of the rule, then possible arguments are restricted to prevent any unexpected or uncontrolled recursions.
- The gainful literals should allow a kind of pruning that is similar to alpha-beta

The first and third constrictions above are defined so as to reduce unnecessary search of literals. The second is defined to prevent unwanted literals that produce infinite recursions. While searching for literals to add to the developing rule, FOIL sometimes might skip the literal that would complete the rule and instead add another literal to the existing rule, that is either determinate or has higher gain. The literal that would complete the rule is stored temporarily and if the final rule is not an improvement of the rule that is stored temporarily, then the stored rule will be the final rule.

Example 2.7

Consider a case with the following background knowledge.

Background Knowledge (B):

Parent(George, Bob),

Parent(Bob,Lisa),

Parent(Alice, Jack),

Parent(Jack, John),

Parent(Bob, Stuart),

Male(Stuart),

Male(Bob),

Male(Jack),

Male(George),

Female(Lisa),

Female(Alice)

Positive Examples(E^+):

Grandfather(George, Lisa),

Grandfather(George, Stuart).

Negative Examples(E^-):

Grandfather(George,Bob), Grandfather(George, Jack), Grandfather(George, Alice), Grandfather(George, John), Grandfather(Bob,Lisa), Grandfather(Bob, Stuart), Grandfather(Bob, George), Grandfather(Bob, Jack), Grandfather(Bob, Alice), Grandfather(Bob, John), Grandfather(Lisa, George), Grandfather(Lisa, Bob), Grandfather(Lisa, Alice), Grandfather(Lisa, Stuart), Grandfather(Lisa, Jack), Grandfather(Lisa, John), Grandfather(Jack, John), Grandfather(Jack, Alice), Grandfather(Jack, Bob), Grandfather(Jack, George), Grandfather(Jack, Stuart), Grandfather(Jack, Lisa), Grandfather(Alice, Jack), Grandfather(Alice, John), Grandfather(Alice, Bob), Grandfather(Alice, George), Grandfather(Alice, Lisa), Grandfather(Alice, Stuart), Grandfather(Stuart, Bob), Grandfather(Stuart, George), Grandfather(Stuart, Alice), Grandfather(Stuart, John), Grandfather(Stuart, Jack), Grandfather(Stuart, Lisa), Grandfather(John, George), Grandfather(John, Bob), Grandfather(John, Lisa), Grandfather(John, Alice), Grandfather(John, Jack), Grandfather(John, Stuart).

A rule for Grandfather has to be generated. Let us assume that the head of the rule is as follows:

Grandfather(X,Y)

Now, in order to generate a rule for *Grandfather*, if the first literal selected is *Parent(X,Y)*, then the rule covers only negative examples without covering any positive examples. Among the search space of literals, some other options are:

Parent(Y,X) - covers no positive examples.

Parent(X,Z) - covers both the positive examples with a few negative examples.

Male(X) - covers both the positive examples with a few negative examples.

Male(Y) - covers one positive example with a few negative examples.

As observed from the above options, the best option would be to choose either *Parent(X,Z)* or *Male(X)* where *Z* is the new variable. Both the literals will be chosen. Suppose, If the first literal chosen is *parent(X,Z)* then the rule will be as follows:

Grandfather(X,Y) :- Parent(X,Z)

Now, among the search space of literals, a few options will be as follows:

Parent(Z,Y) - covers both the positives with only one negative example.

Parent(X,Y) - covers more negative examples than other alternatives.

Male(X) - covers both the positive examples with a few negative examples.

Male(Y) - covers one positive example with a few negative examples.

The best option among the above is to choose *Parent(Z,Y)*, as it covers both the positives and very few negatives. Also, this option will make use of the new variable introduced in the first literal. So, upon adding this literal to the existing rule, the rule will become

Grandfather(X,Y) :- Parent(X,Z), Parent(Z,Y)

Again, the process of searching for literals takes place and finally, the literal *Male(X)* will cover all positives and no negatives. So the final rule will be as follows:

Grandfather(X,Y) :- Parent(X,Z), Parent(Z,Y), Male(X)

2.5.3 FOCL

First order combined learner (FOCL) [17] is an extension of FOIL. It uses an explanation-based learning (EBL) component which helps in using initial domain theory in the form of partial theory, intentionally-specified background relations, and relational clichés. FOIL theory might result in rules that are either too general in the sense that they cover negative examples or too specific in the sense that they miss out on a few positive examples. In order to overcome this deficiency, FOCL uses two mechanisms to add literals. First, it can simply use the FOIL method to generate new literals and add them to existing partial rule. Second, it can create new literals by operationalizing a target concept. Operationalizing a target concept means a non-operational definition of the concept that has to be learned. In order to evaluate the gain achieved by adding the empirical literal or a literal that is learned analytically, FOCL uses FOIL's information-based evaluation function. FOCL learns rules that are of the following form:

$$rule \leftarrow O_i \wedge O_d \wedge O_f$$

O_i is the initial operational literal that is learned empirically, meaning by using the FOIL's method of learning literals. Od is the operational literal learned from the domain theory and Of is the final literal that is again learned empirically.

While choosing literals from the domain theory, the background knowledge that is now in the form of intentionally formed rules is evaluated and if it has a higher gain than the literals that are added empirically, then an appropriate part of the rule is selected and is added to the current existing partial rule. Also, the domain theory contains relational cliches that contain literals that belong together and will be in a sequence. In this way, FOCL provides a wider choice from which literals can be choosen by using the combination of empirical approach as well as the *EBL* approach.

2.5.4 Audrey II

UCI proposed another system called Audrey II [25] which is similar to FOCL. Using the mechanism proposed by FOCL, it tries to specialize the rules that are general, by adding new literals to the existing rule. However, Audrey II differs from FOCL, where apart from adding new literals, it will also replace some literals that are already in the existing rule by using four revision operators.

2.5.5 mFoil

mFoil [7] uses beam search instead of FOIL's greedy search mechanism. This method tries to make FOIL more robust while dealing with noisy data. By using beam search, the chances of finding a good literal to add to the existing rule is high. Instead of evaluating literals on information gain, the literals in mFoil are evaluated based on the estimated accuracy of the new clause. Also, it uses a statistical significance test instead of MDL to decide when a rule should be stopped from growing further.

2.5.6 Fossil

Fossil [8] presents a new search heuristic based on statistical correlation along with a stopping criterion. These heuristics are shown to perform better on chess-driven data relation *illegal* when the data was induced with moderate level of noise. This approach is efficient as there is no need to calculate separately for a heuristic function for negated literals and because of the stopping criterion, the unnecessary computations will be neglected. Also, the stopping criterion is just a comparison and hence will not take much time to compute. There is a cutoff parameter, that allows to consider only the literals that are considered to have a certain minimum correlation value. This makes the system robust as it was proved that a good cutoff value seems to be independent of the amount of noise and the number of training examples. Fossil is said to be simple in the sense that, as the cutoff parameter and its relation to search heuristic will mix learning as well as pruning.

2.5.7 Hydra

Hydra [1] is an extension of FOIL that deals with noise. Instead of just trying to find the rule for target relation, Hydra widens its learning mechanism to incorporate learning for all classes. If the target relation is R, then Hydra tries to learn rule for not R (negation of R) as well. By following this approach, it is considering both the current target relation and also other relations where it is not in target relation and thus trying to eliminate noisy data. Since, negation of a target might not be a rule that is complete and closed, it might seem that the other definition is much simple. To evaluate the reliability of a particular rule, likelihood ratios that are derived from coverage of positive and negative examples of that rule is used. Hydra uses likelihood ratios to select the literal that has to be added to the existing rule instead of information gain. This method proved to be very efficient on noisy data such as the KRK data (Chess data). Its robustness is higher when compared to FOIL.

2.5.8 Grasshopper

Grasshopper [12] uses an inductive approach to learn search control rules in planning domains. The grasshopper tries to search for similar decisions from the training instances and will produce new search control rules from that. It avoids the use of any domain theory, by generating preconditions for the rules generated from the previous step by characterising the problem solving context in which each decision occurred. This leads to reduced search cost and by using search control rules and also using more than one training problem is taken into consideration when learning a new rule.

Grasshopper tries to suggest modifications to the planner's search strategy by tracing the planning system's behaviour during a search. All the problems that are solved by the planning search are analyzed by Grasshopper and are searched to find for any interesting decisions that occurred during the search. These decisions will in turn help in learning new search control rules. These rules are combined with the rules that are already existing, like the domain rules, to act as a guide to the planner on subsequent problems. There are five important stages in Grasshopper and they are:

- **Decision Extraction**: This stage will try to extract interesting decisions from the plan search tree.
- **Decision Clustering**: Groups the decisions extracted according to the type of search control advice they provide.
- **Decision characterisation**: In this stage all the decisions in a cluster are provided with a description. This helps in recognizing a cluster when it is needed.
- **Rule Generation**: Converts the description from the cluster of decisions into a search control rule.

- **Utility Optimization**: Evaluates the utility of new rules and decides whether it has to be passed on to the planner.

Grasshopper on the whole, will increase the speed of planning a system. It is also different from the EBL based methods given that it doesn't need any domain knowledge or heuristics.

2.5.9 MPL

FOIL and many other systems learn more than one target relations one after the other. But, it is not possible to learn more than one target relation at once. There are some cases such as mutual recursion where more than one target relation has to be learnt at once. Multiple predicate learning (MPL) [5] provides a scope to learn more than one target relation together. The target relations are developed keeping in check the local as well as global consistency. Local consistency is achieved when the rule does not cover any negative examples from the actual training set. Global consistency is achieved when the rule does not cover any negative examples from the extended training set. MPL uses search heuristics that are similar to mFoil's.

2.5.10 ALEPH

A learning engine for proposing hypotheses (ALEPH) [21] is an ILP system that is based on the ideas of inverse entailment. The work for ALEPH started in 1993, but was later implemented to be on par with most of the related systems like FOIL and others. ALEPH uses a bottom up approach and is non incremental. ALEPH algorithm can be summarized using the following procedure:

1. **Select an example**: Select an example, that needs to be generalized. If no such example exists, then stop the algorithm, else proceed to next step.

2. **Construct a specific rule**: Construct a rule of the selected example that is most specific. This rule is called the *bottom clause* and it should be within language restrictions provided. This step is called the *saturation step*.

3. **Search**: Find a more general rule. Search the subsets of literals in the rule. The subset that has the best score is chosen. These subsets may not produce a more general rule, but is good enough for this sketch. This is the *reduction* step where the rule will become more general.

4. **Remove Redundant**: The rule that has the best score is added to the existing theory of rules. All the examples that become redundant are removed. If the best rule makes examples other than those covered by the rule as redundant then it is ignored. Return to step 1.

ALEPH uses mode declarations that inform if a relation can be used in a head or tail of the generated rule. ALEPH can be tuned as per the requirement. Instead of selecting one example to be generalized, more than one examples can be chosen. In this case a bottom clause has to be created for each example. After the reduction step, the example that created the best of all reductions will be added to the existing set of rules. The search mechanisms can be changed instead of just using one particular method.

3 Implementation

The aim of the system presented in this thesis is to implement the Inductive logic programming algorithm, Sequential Covering, using Java as the front end and MySQL as the back end database. In order to implement this algorithm, a mechanism has to be found where in facts, and rules can be stored in databases. The following sections talk about storing the facts and rules in the database.

3.1 Storing Facts

Predicates are the key to any ILP system. They form a rule. These predicates have to be represented in such a way, that they could be utilized in the same way as a regular ILP algorithm uses them. This is made possible by storing them as tables in the database. The system works in the following way:

- The predicate will be broken into parts, where the name as well as the arguments of that predicate are retrieved.
- The database is searched to retrieve if a table with the name of the predicate already exists or not. If not, then a table with the name of the predicate will be created along with the corresponding number of columns (arguments) that it holds. The query for this will be generated by the code written in Java.
- The arguments are added to the table in each column.

Table 3.1: An example of storing facts for Father table

Column1	Column2
John	Jack

While storing the predicate in the database the name of the predicate has to be unique. There can be predicates that have same name but different number of arguments, in such a case it is difficult to separate these two. To overcome this situation, the table names will have the number of arguments that a predicate holds as well. This will make all the predicate names unique.

Example 3.1

Consider the following predicate:

Father(John, Jack)

In this case, the database will be searched if there is a table such as *tbl_father_2Args*. If such a table exists, then the arguments *John* and *Jack* are added to table as column values. The column names will simply be as column1 and column2. Table 3.1 depicts the representation of a fact in database. The name of this table is *tbl_father_2Args*.

This representation of table names and column names will be very helpful when retrieving these values.

3.2 Storing Rules

Rules are stored in the database in the form of database views. Rules or clauses are the heart of any ILP algorithm. There will be many rules that will be produced on the way to finding a hypothesis theory and these rules generated may have different properties. The predicates in the rules may have common arguments at different places. The rules may have

same predicates that differ in arguments. Generating a view for the rules should consider all these possibilities. The outline of the steps taken to generate a view is as follows:

- Separate the rule into two parts with the first being the target predicate which is the head of the rule and the other being the predicates on the body of a rule.
- Split the body of the rule to get each predicate on the body of the rule.
- Search for common arguments in the predicates.
- If there are any common arguments, then create a join query for those two tables (predicates are represented as tables in database) on the respective columns.
- Next, the view has to have a select statement that selects the columns from the tables on the body of the rule.
- Search for arguments that are in the body of the rule as well as in the head of the rule.
- Create a select query that will select the common arguments in the head and body of the rule using the predicates in the body of the rule that are represented as tables in the database.
- Append the join query to the select query to generate a final query for the view.
- Create a view with this query and store it in the database.

The names of these views should also be unique. Same target rule can have different body. So, the view name cannot just be the name and arguments of the head of the rule. It should consider the tables on the body of the rule as well so that the view names are unique. In order to achieve this uniqueness, the names of the views will have the name of the table with number of arguments on the head of the rule along with the tables and their respective

number of arguments on the body of the rule.

The body of the rule can have two same predicates with same arguments. In this case, it is difficult to refer which table to choose while joining or while selecting. So, every table in the view will be given a unique alias. This makes it easy to select and join.

Example 3.2

I A(X) :- B(X)

In this rule, there is just one predicate in the body. So, there will be no joins in this kind of situation. The view for this rule will be simply be as follows:

create view vw_A_1Args_B_1Args as select column1 from tbl_B_1Args

This is a simple example of storing a rule that does not have any Joins.

II Father(X,Y) :- Parent(X,Y), Man(X)

In this rule, there are two predicates in the body of the rule. These predicates have a common argument, *X*. In this case, these two predicates must be joined on this common argument. So, the view for this rule will be as follow:

create view vw_father_2Args_parent_2Args_man_1Args as select table1.column1, table1. column2 from tbl_parent_2Args table1 join tbl_man_1Args table2 on table1.column1 = table2.column1

This is an example that has joining of tables. Here, both the columns are selected from the table *tbl_parent_2Args* because selection of columns will occur after the tables are joined and then it gives the same output if the column1 is selected either from *tbl_parent_2Args* or *tbl_man_1Args*.

III GrandFather(X,Y) :- Parent(X,Z), Parent(Z,Y), man(X)

In this rule, a new variable, *Z*, is introduced. There are also more than one joins with a join being on new variable. Also, two of the predicates are same, so they have to be given different aliases in the view. The view for this will be as follows:

create view vw_grandfather_2Args_parent_2Args_... as select table1.column1, table2.column2 from tbl_parent_2Args table1 join tbl_parent_2Args table2 on table1.column2 = table2.column1 tbl_man_1Args table3 on table1.column1 = table3.column1

This view has two joinings along with the introduction of a new variable. The variable *Z* is not in the head of the rule, hence it will not be in the selected from the table.

IV act(A) :- atm(A,B,o,40,C)

In this rule, there are constants. This rule is part of a theory that states if a particular atom is active or not. Constants are the values present in a particular table. Here *o* is a constant which is in the column3 of table *tbl_atm_5Args*. Similarly *40* is a constant that is in the column4 of the same table. The view for this rule will be as follows:

create view vw_act_1Args_atm_5Args as select table1.column1 from tbl_atm_5Args table1 where table1.column3 = 'o' and table1.column4 = '40'

In this view, the constants are obtained with a where clause.

3.3 Generating a Hypothesis

The process of learning a rule starts from taking the input from user. The input contains the tables from the database that can be used for rule generation. These tables as talked about in Section 3.1 will be the predicates or facts that are already existing in the database. User has to provide the table for which a rule has to be generated, along with the number of columns that the rule should have. The input should also contain the tables with their

respective columns and also the columns in which the constants have to be considered as well for the generation of a rule. The positive and negative table names have to be provided as well. The typical input to generate a rule for father table will be as follows:

1. *SCHEMA_NAME-Thesis*
2. *tbl_father_2Args-father-child*
3. *tbl_man_1Args-male(ion)*
4. *tbl_parent_2Args-parent(ion)-child(ion)*
5. *PositiveTable-tbl_Father_2Args*
6. *NegativeTable-negativeExamples_Father*

The above example has the rule that has to be generated in line 1. The lines 2 and 3 contain the names of the tables in the database that has to be used for rule generation. The '-' separates each part of the table input. Line 3 can be further divided as table name, columns of that table that have to be considered along with their mode of input. Mode describes the type of values that a column can take.The modes of input is further explained below:

Mode *i* : This mode stands for input. Input means the column value in this predicate can be a value from head of the rule. For example, *i* in *tbl_man_1Args* means that it can take a value of either *father* or *child* which are in the head of the rule.

Mode *n* : This mode stands for *NEW*, which basically means that it can take a new variable. For example, in *tbl_man_1Args* means that it can take a new variable as the value of the column.

Mode *o* : This mode stands for *OLD*, which means that the value of that particular column can take a variable that has been introduced before. For example, if a new variable has been introduced in *tbl_parent_2Args*, then this variable can be the value of column *male* in *tbl_man_1Args*.

Mode *c* : This mode stands for *constants*. If this mode is in a column, then it means that that column can take the constant values which are the values in the database table.

Mode *f* : This mode stands for *fixed*, which means that the values of that column are fixed meaning that it can only take values that are of that particular column. For example, if the value of *parent* is fixed in *tbl_parent_2Args*, then it can only take the variables of the column that are *parent*. If there is only one column, then it means that it can take only values of *parent* column.

The lines 4 and 5 contain the names of the tables that contain positive examples and negative examples respectively. The name *PositiveTable* is used to suggest that the input is of positive examples table. The word after the '-' will be the name of the table that contains the positive examples in database. Similar is the case for Negative examples table. The line numbers here are provided for understanding purposes and they should not be included in the actual input.

This input is provided in a file that is processed in the code and all the required elements to generate the rule are retrieved.

3.3.1 Initial Setup

Initial setup takes care of retrieving constants from the database if needed and also generating all the different possible combination of arguments among the input tables. Constants are retrieved based on the input provided. If the input is as follows:

tbl_parent_2Args-1-2-constants-1-2

Here the constants or values in the columns 1 and 2 of the table *parent* are retrieved and are stored in the memory. If there are a lot of constants then the retrieved constant values can be stored in a temporary table in database. But, if there aren't many constants then it is feasible to store it in memory as it avoids many database calls.

The concept of generating all the possible arguments is to test all the possible cases for a rule and to know which rule is covering more positive examples. All the possible arguments for a table such as *parent* is shown in 3.3.

Example 3.3

The input for a parent table is as follows:

tbl_parent_2Args-1-2

All the different possible combinations generated for Parent table, which is a table with two columns is as follows:

parent(column1,column2),
parent(column2,column1),
parent(new,column1),
parent(new, old),
parent(column2, new),
parent(new,column2),
parent(column1,new),
parent(old ,new),
parent(old,column1),
parent(old,column2),

parent(old,old),

parent(column1,old),

parent(column2,old),

parent(old,old)

new in a column means that a new variable has to take that place. *old* in a column means that that column should have a variable that is already introduced in the previous predicates.

GrandFather(column1,column2) :- Parent(column1,Z), Parent(Z,column2),

Man(column1)

For this rule, *Z* is a new variable which is introduced in the first predicate of the body of the rule. *Z* becomes an old variable when being used as an argument in the second predicate. So, the first predicate is derived from the combination of *Parent(column1, new)* and second predicate is derived from the combination *Parent(old, column2)*.

The *old* argument in the predicate parent should replace not just one previous new variable but, all the variables that are introduced previously in the rule generation process.

These combinations will later be used to generate rules and are scored to calculate which rule is best performing. Similar procedure is followed for all the tables that could end up on the body of the rule. The meaning of new in these predicates are that a new variable should appear in that place and old means a variable that is already introduced in the rule will appear in that place. There is no combination of *parent(new,new)* as it does not make any sense introducing two new variables in a predicate that is not joined by any other existing variable.

Since the Sequential Covering algorithm requires positive examples to be deleted, the positive example table provided in input is used to create a temporary positive table that will have its values that are covered by the current rule to be deleted.

3.3.2 Learn Rules

After the initial setup is completed, a learn rule method will be called which learns rules using the Sequential Covering algorithm described in . The initial rule consists of the head of the rule and the following process adds every possible literal to the body of the rule and finally picks out the best one.

- While positive examples exist:

    1. For each combination of arguments generated in the initial setup, append it to the rule as a new literal. Add these rules to a list of generated rules.
    2. For each generated rule in the previous step, replace the new variables with constants if needed and add these newly created rules with constants to the overall list of generated rules.
    3. For each generated rule, score each rule by creating a view and passing it to the database. A stored procedure in the database takes care of storing the rule along with its score in a table.
    4. Pick top n (any required number) rules from the database and repeat step 1 until some set number of times (4 mostly). This results in adding more literals to the body of the rule.
    5. From the table created in the database, that stores all the generated rules, pick a rule that has the best score. Scoring can be calculated based on different mechanisms. The scoring will have both positive score (number of examples covered in positive table) and negative score (number of examples covered in negative table). The sum of positive score and negative score is used mostly.
    6. Add the best scoring rule picked from previous step into the hypothesis table in the database and delete the positive examples covered by that rule from positive

table.

The old variable in step 1 will be replaced only if there is a variable that is already introduced in the previous iteration. This process of generating a hypothesis requires some intense calculations and memory. There might be a million rules generated at a point of time and scoring all these rules and storing them in memory is not feasible. In order to expedite the process of scoring the rules, threading concept in Java is used. The rules generated will run on single thread, but scoring the generated rules will be divided into multiple threads. This reduces the computation on one core and divides the process among many cores.

In order to achieve parallel computations, the database connections have to be opened every time a thread is created. This will result in opening many connections and soon the maximum number of connections that can be opened will be reached. To overcome this problem, database connection pool is used. A database connection pool will limit the number of connections used, as it closes the unnecessary connections or reallocates them to a new thread. Also, all the tables in the database were indexed so that the joining of tables take very less time.

Also, if the value of *n* in step 4 is large, then in order to reduce the memory usage, only a set number of rules can be retrieved from the database at a time. In order to the avoid the confusion of retrieving that are of next level instead of current level, a level column is added to the database table from which the rules will be retrieved.

Example 3.4

Generate a rule for *Father* table using Parent and Man tables. The tables 3.2 and 3.3 contain the background knowledge, the table 3.4 contains the positive examples and, 3.5 contains the negative examples.

Before running our system, the database has to be setup as described above. Apart from these tables, the database should have two more tables. One table for storing intermediate

Table 3.2: Man table stored in database

Column1
John
Max
Jack
Jeff

Table 3.3: Parent table stored in database

Column1	Column2
John	Max
John	Jeff
Jack	Diana
Diana	John

Table 3.4: Positive Examples table for Father rule

Column1	Column2
John	Max
John	Jeff
Jack	Diana

rules generated during the process and other to store the rules that are part of the hypothesis. The input for the system will be as follows:

tbl_father_2Args-1-2

tbl_man_1Args-1

tbl_parent_2Args-1-2

PositiveTable-tbl_Father_2Args

NegativeTable-negativeExamples_Father

After reading the input, the system generates all possible args as described in Section 3.3.1 A clause is established as follows:

Table 3.5: Negative Examples table for Father rule

Column1	Column2
Diana	John
Max	Jack
Jeff	John
Diana	Jack
Diana	Max
Diana	Jeff
Max	John
Max	Jeff
Max	Diana
Jeff	Jack
Jeff	Max
Jeff	Diana
John	Max
John	Diana
John	Jack
Jack	Max
Jack	John
Jack	Jeff

father(column1,column2) :-

The learning process starts from the head of the rule because our system follows a top-down approach. The learning process flows in the similar form as a tree generation, shown in Figure 2.2

In the first stage, all the different possible combinations are added to the rule and scored.

father(column1, column2) :- parent(column1, column2) (3.1)

For this combination, the queries that scores this rule are as follows:

Following query is a positive query:

SELECT count(distinct PositiveTable.column1,PositiveTable.column2) FROM positiveTable AS PositiveTable JOIN (SELECT distinct table1.column1 AS column1,table1.column2 AS column2 FROM tbl_parent_2Args table1) AS ruleExamples ON ruleExamples.column1 = PositiveTable.column1 AND ruleExamples.column2 = PositiveTable.column2

The number of rows that this query returns will be the positive score of the rule.

Following query is a negative query:

SELECT count(distinct NegativeTable.column1,NegativeTable.column2) FROM negative-Examples_Father AS NegativeTable JOIN (SELECT distinct table1.column1 AS column1 , table1.column2 AS column2 FROM tbl_parent_2Args table1) AS ruleExamples ON ruleExamples.column1 = NegativeTable.column1 AND ruleExamples.column2 = NegativeTable.column2

The number of rows that this query returns will be the negative score of the rule.

The rule in Equation 3.1 will have a positive score of 3 and a negative score of 4.
The positive examples that are covered are:

Father(Jack, Max)

Father(John, Jeff)

Father(Jack,Diana)

The negative examples covered are:

Father(Diana,Jack)

Father(Diana, Max)

Father(Diana, Jeff)

Father(Diana, John)

In this way all the possible combinations are generated and scored. These rules are stored in a table that stores all these intermediate rules. Later, a predefined number of top rules (300) are retrieved. A rule is classified in the top rule based on a performance mechanism. For example, if a rule has a positive score of 10 and negative score of 5, then with the positive score + negative score performance mechanism, the overall score of the rule is 5. The top rules are calculated based on this performance mechanism.

Some of the rules generated at the end of first level are:

(1) *father(column1,column2) :-parent(column1,column2)*
positive score : 3 negative score : 4

(2) *father(column1,column2) :- man(column1)*
positive score : 3 negative score : 4

(3) *father(column1,column2) :- parent(column1,newVar1)*
positive score : 3 negative score : 18

(4) *father(column1, column2):- man(column2)*
positive score : 3 negative score : 15

The top scoring rules are rule 1 and 2. However, for such short data all the rules are picked and sent to the next level. More literals will be added to the body of the rule. So, the positive or negative query will now contain joins between the common arguments in the literals. For example, the positive and negative queries generated for the following rule are:

father(column1, column2) :- parent(column1, column2), man(column1)

Following query is a positive query:

SELECT count(distinct PositiveTable.column1,PositiveTable.column2) FROM positiveTable AS PositiveTable JOIN (SELECT distinct table1.column1 AS column1, table1.column2 AS column2 FROM tbl_parent_2Args table1 JOIN tbl_man_1Args table2 ON table1.column1 = table2.column1) AS ruleExamples ON ruleExamples.column1 = PositiveTable.column1 AND ruleExamples.column2 = PositiveTable.column2

Following query is a negative query:

SELECT count(distinct NegativeTable.column1,NegativeTable.column2) FROM negative-Examples_Father AS NegativeTable JOIN (SELECT distinct table1.column1 AS column1 , table1.column2 AS column2 FROM tbl_parent_2Args table1 JOIN tbl_man_1Args table2 ON table1.column1 = table2.column1) AS ruleExamples ON ruleExamples.column1 = NegativeTable.column1 AND ruleExamples.column2 = NegativeTable.column2

Some of the rules that will be obtained in the second level are:

(1) father(column1,column2) :-parent(column1,column2), man(column1)

positive score : 3 negative score : 0

(2) father(column1,column2) :-parent(column1,column2), man(column2)

positive score: 2 negative score : 1

(3) father(column1,column2) :-parent(column1,column2), parent(column2,column1)

positive score : 0 negative score : 1

At the end of a predefined number of recursions (usually 4), a best rule is picked from the table that stores all the intermediate rules. The positive examples that are covered by this rule are deleted and then it is added to the table that stores the hypothesis rules. The table that stores intermediate rules is truncated. This process repeats until all the positive examples are deleted.

In this case, the following rule is generated by the system. This rule covers all the positive examples, without covering any negative examples.

father(column1, column2) :- parent(column1, column2), man(column1)

3.4 Storing Recursive Rules

Recursive rules are represented with a + sign that indicates that there is a recursion. The input to store a recursive rule will be as follows:

ancestor(X,Y):- Parent(X,Y) + ancestor(X,Y):- Parent(X,Z), ancestor(Z,Y)

The reason normal rules have to be distinguished from recursion is that recursion has a different type of rule, wherein it reiterates a recursive rule until it reaches a base rule. The first part of the rule is called base case rule and the second part is the recursive rule. A base case view will be generated from the base case rule. A recursive view will be generated

from the recursive part of the rule. A recursive view is generated from a recursive rule by joining the common arguments.

Example 3.5

As shown in 2.2 the recursive rule for ancestor will be

Ancestor(X,Y):- Parent(X,Z), Ancestor(Z,Y)

Ancestor(X,Y) :- Parent(X,Y)

A temporary table, say *anc_temp*, will be created from parent with *X* in column1 and any unspecified value in column2. Later column2 in *anc_temp* table is joined against column1 in parent table and the resulting values will again be added to the temporary table *anc_temp*. This process keeps repeating until there are no more additions to *anc_temp* table.

The joining part of the recursive rule is generated in a Java code and is passed to a stored procedure in database. The database uses a loop to keep joining the tables together until there are no new additions to the table.

And since there are more than one views involved in this, the recursive view is stored in a table along with the base case view and temporary table name.

3.5 Learning Recursive Rules

Recursive rules are generated in a similar way as non-recursive rules. The input to learn recursive rules is similar to non-recursive rules input described in section 3.3, except that a + sign has to be added to indicate that there could be a recursion. The initial setup is as described in section 3.3.1. In order to generate a recursive rule, a base case rule will be identified first. The base case rule is identified as a non-recursive process. This rule is

something that could be identified as a rule that covers no negative examples apart from a few positive examples it covers. The following process describes in detail the procedure to find a recursive rule.

- While positive examples exist:

    1. Using the arguments generated in initial setup, generate different rules and score them against the positive table and negative table.
    2. Keep repeating step 1 until a rule that covers no negative examples is encountered or until a set number of times.
    3. When a rule that covers no negative examples is found, set it as a base case rule and try to find a recursive rule.
    4. Generate all the possible arguments for recursive rule using the recursive table. This process is similar to generating all the possible arguments for a non-recursive rule.
    5. For each argument generated in step 4, construct a recursive rule and score them using the recursive stored procedure described in section 3.4.
    6. Keep repeating the process until a best recursive rule is found.

Example 3.6

The input for the system to learn a recursive rule is as follows:

tbl_ancestor_2Args-1-2-+

tbl_parent_2Args-1-2

PositiveTable-anc_positiveTable

NegativeTable-anc_negativeTable

Table 3.6: Parent table to generate a recursive rule

Column1	Column2
Jack	Diana
Diana	Bob
Bob	Linda
Diana	Kate

Table 3.7: Positive examples table to generate a recursive rule

Column1	Column2
Jack	Diana
Diana	Bob
Bob	Linda
Diana	Kate
Jack	Bob
Jack	Linda
Jack	Kate
Diana	Linda

The + symbol in line 1 indicates that there is recursion involved in this rule generation. The system then acts accordingly and first tries to learn a base case rule and then will learn a recursive rule.

Consider the background knowledge and positive examples provided in Example 2.2. This example demonstrates the generation of Ancestor rule for recursion. They are represented in the database as follows:

Table 3.6 contains the background knowledge required to generate a positive table. Table 3.7 contains the positive examples and Table 3.8 contains the negative examples. The process of generating a base case rule is similar to the process described in sections 3.3. The termination criterion for the base case rule here is that it should not cover any negative examples.

Table 3.8: Negative examples table to generate a recursive rule

Column1	**Column2**
Diana	Jack
Kate	Bob
Kate	Linda
Kate	Diana
Kate	Jack
Bob	Diana
Bob	Kate
Bob	Jack
Linda	Jack
Linda	Bob
Linda	Diana
Linda	Kate

The rule that does not cover any negative examples will be as follows:

$$Ancestor(X, Y) : -Parent(X, Y) \tag{3.2}$$

The rule in Equation 3.2 is the base case rule and based on this, the recursive rule will be generated.

In order to generate a recursive, generate all the possible combination of literals. These literals can be either *Ancestor* or *Parent*. So, all the possible combinations of these two predicates will be generated. Here, *Ancestor* will be the head of the rule and will also be part of the body of the rule as there is recursion involved in the rule generation.

The process of adding literals to the body of the rule remains same. The process of generating a recursive rule is as follows: The head of the rule will be initialized as follows:

Ancestor(X,Y) :-

Similar to the process of generating a rule without recursion, this process involves generating all the possible literals (different combination of arguments). These literals are added to the body of the rule. Some examples of literals that are be added to the rule are :

Ancestor(X,Y) :- Parent(Y,X)

Ancestor(X,Y) :- Parent(X,Z)

Ancestor(X,Y) :- Ancestor(X,Z)

All these rules are clubbed with the generated base case rule and are scored in the database. Some predefined number of top rules are chosen to proceed to next level. This process of picking top rules is similar to the process given in Example 3.4.This process of adding literals to each of the top rule is repeated until all the positive examples in the database are covered. Finally, the rule that covers all the positive examples in this case is as follows:

Ancestor(X,Y):- Parent(X,Z), Ancestor(Z,Y) (3.3)

The final recursive rule is obtained by clubbing equations 3.2 and 3.3.

Ancestor(X,Y):- Parent(X,Z), ancestor(Z,Y)

Ancestor(X,Y) :- Parent(X,Y)

This is the process of generating a recursive rule.

4 Results

The system described in this thesis is first tested on some simple data. A *Father* rule was generated with the tables *parent* and *man*. Using a manually synthesized data, the rule generated by the system is:

Father(X,Y) :- Parent(X,Y), Man(X)

Similarly, a *GrandFather* rule is also generated by the system using the same data. The rule is as follows:

GrandFather(X,Y) :- Parent(X,Z), Parent(Z,Y), Man(X)

A Recursion rule is also generated by the system using the same data. The rule is as follows:

*Father*Ancestor(X,Y) :- Parent(X,Z), Ancestor(Z,Y)

Ancestor(X,Y) :- Parent(X,Y)

The system described in this thesis is tested mainly on two different data sets. One is a Mutagenesis data set and another is a Chess data set. These data sets have been widely used to test many ILP systems which is the reason for choosing them.

4.1 Mutagenesis Data Set

Mutagenesis data set is a problem of predicting the mutagenicity of 230 compounds. The mutagenicity of the compounds have to be predicted using the atomic and bond structure of the compounds. These compounds are carcinogenic and can cause damage to DNA. Therefore, it is of good knowledge to understand which compounds or molecular features have mutagenic activity. With Mutagenesis data, the effectiveness of ILP programs can be tested as there is no help from any external sources. The system progol described in [22] is compared to the system presented in this thesis.

The atom and bond structures of the 230 compounds were obtained from a standard molecular modeling package QUANTA. QUANTA is a package, that automatically obtains the atoms, bonds, bond types, atom types for each of the 230 compounds. The output from this was a set of prolog facts of two forms, which are :

bond(compound, atom1, atom2, bondtype) - This form states that the *compound* has a bond of *bondtype* between the atoms *atom1* and *atom2*. Consider the following example:

bond(d2,d2_1, d2_2,7)

This states that there is an aromatic bond (bondtype *7* refers to aromatic type) between atoms *d2_1* and *d2_2*.

atm(compound, atom, element, atomtype, charge) - This form states that in *compound*, *atom* has an *element* of *atomtype* and partial charge *charge*. Consider the following example:

atm(d2,d2_1,c,22,0.067)

This states that atom *d2_1* in *d2* is an aromatic carbon atom with partial charge *0.067*.

Overall, there 12203 facts including atoms and bonds. These facts are the predicates in the representation of this system and are stored as tables. The mutagenicity of compounds, can be calculated using a linear regression model. Using this method, the compounds are divided into two classes. The compounds that are mutagenic are called *active* compounds and the compounds that are not mutagenic are called *inactive* compounds. Out of the 230 compounds available, it is given that 138 of them are *active* and 92 are *inactive*.

The compounds are again divided into two classes. They are regression friendly and regression unfriendly. The regression friendly compounds have 125 active compounds and 63 inactive compounds. So, as per the representation of this system, there are 125 positive examples and 63 negative examples. This thesis carries out the test on this regression friendly system.

The progol system uses some language specifications to generate the hypothesis. It says that constants can be only from the columns 3,4,5 in the atom table and from column 4 in bond table. Hence, only these columns will be considered for constants. In order to generate all the possible arguments for this data, mode declarations are considered. The mode declarations are constraints that restrict certain kind of rules from being generated. The mode declarations for atom are as follows:

mode(,atm(,+compound,+atomid,#element,#integer,-charge))*

mode(,atm(,+compound, - atomid,#element,#integer,-charge))*

mode(1,(+charge)=(#charge))

This says that the atom should have a new variable in column1, a new or an old variable in column2, a constant in column3, a constant in column4, a constant in column5 if there is

only one predicate or a old variable in column5.

The mode declarations for bond are as follows:

mode(,bond(+compound, - atomid, - atomid,#integer))*

mode(,bond(+compound, -atomid,+atomid,#integer))*

mode(,bond(+compound,+atomid, - atomid,#integer))*

mode(,bond(+compound,+atomid,+atomid,#integer))*

This says that bond should have a new variable in column1, an old or a new variable in column2, an old or a new variable in column3, a constant in column4. Based on these mode declarations described, a new method was implemented to generate all the possible arguments taking into considerations all these restrictions.

The input is given to the system as follows:

tbl_act_2Args-1

tbl_atm_5Args-1-2-3-4-5-constants-3-4-5

tbl_bond_4Args-1-2-3-4-constants-4

Apart from a change in generating all the possible arguments method, the rest of the system remains same and it is run to produce the following hypothesis:

1. *act(A) :-bond(A,B,C,1), atm(A,B,c,22,D), atm(A,C,c,10,E)*
   Accuracy = 88 Coverage = 28
2. *act(A) :-bond(A,B,C,1), atm(A,D,c,27,E), bond(A,C,D,7)*
   Accuracy = 86 Coverage = 19
3. *act(A) :-atm(A,B,o,40,-0.384)*
   Accuracy = 79 Coverage = 9

Table 4.1: Scores of our system

	Predicted_Active	Predicted_Inactive
Actual_Active	125	0
Actual_InActive	27	36

Table 4.2: Scores of Progol system

	Predicted_Active	Predicted_Inactive
Actual_Active	100	25
Actual_InActive	13	50

4. *act(A) :-atm(A,B,h,3,0.144)*

   Accuracy = 89 Coverage = 6

5. *act(A) :-atm(A,B,h,3,0.142)*

   Accuracy = 89 Coverage = 6

6. *act(A) :-atm(A,B,h,3,0.147)*

   Accuracy = 83 Coverage = 4

7. *act(A) :-atm(A,B,c,14,C), bond(A,D,B,1)*

   Accuracy = 67 Coverage = 5

8. *act(A) :-bond(A,B,C,1), bond(A,C,D,7), atm(A,B,n,32,E)*

   Accuracy = 67 Coverage = 5

9. *act(A) :-atm(A,B,c,195,C)*

   Accuracy = 100 Coverage = 3

10. *act(A) :-atm(A,B,h,3,0.149)*

    Accuracy = 80.0 Coverage = 3.2

This hypothesis is very similar to the hypothesis generated by progol. Among the 10 rules generated, six of them are same as the ones generated by progol. The accuracy of this system is also in comparable terms to that of the progol system. Following tables, compare this system with the progol system.

Table 4.1 projects the scores generated by our system. Table 4.2 projects the scores generated by Progol system. The accuracy of our system is *85.6%*, and for Progol system is *80%*. By comparing the results, it is pretty evident that the results obtained by the system presented in this thesis are very feasible and comparable to the Progol system.

4.2 Chess Dataset

The chess dataset was designed by keeping in mind that could a machine learn to play chess given only example positions and some simple facts about the geometry of the board? This chess dataset is well known as KRK (King Rook King) data set. White has a king and a rook and black has just a king to defend. The chess dataset is made out of this situation with black and white-to-move. Our system is compared with the system presented in [2]

The chess dataset has a total of seven columns with two columns each for the King, Rook, and King, and a column that shows the depth or number of moves in which the White can win the game. The Chess board is represented in length as numbers from 1 to 8 and in width as alphabets from a to h. For Example, Consider the following chess fact:

chess(a,1,b,3,c,2,draw)

This fact says that when the White King is at position a1, White Rook is at b3, Black King at c2 then it results in a draw.

Consider another fact.

chess(d,3,h,1,d,1,zero)

This fact says that when the White King is at position d3, White Rook is at h1, Black King at d1 then the number of moves required for White to win the game is zero which means that White has won the game.

Consider another fact.

chess(c,1,c,3,a,2,one)

This fact says that when the White King is at position c1, White Rook is at c3, Black King at a2 then the number of moves required for White to win the game is one.

The dataset contains the data for a maximum of 16 moves in which White can win the game making a total of 28056 rows overall. The results obtained by the system presented in this thesis is compared to a system implemented by Michael Bain in [ ]. The results are compared for the depths zero and one, as only the results of these are given by Bain in his thesis, even though the system was able to generate a hypothesis for other depths as well. The input for this dataset will be as follows:

tbl_zero_6Args-1-2-3-4-5-6

tbl_chess_6Args-1-2-3-4-5-6-constants-1-2-3-4-5-6

The hypothesis language for this dataset is that there can be constants at any place in the rule and hence all the 6 column constants are considered. Generation of all possible arguments is same as the normal procedure described in section 3.3.1

The hypothesis generated by the system presented in this thesis is as follows:

1. *zero (A, B, C, D, E, F) :- chess(c, G, a, H, a, 1)*

   Accuracy = 86 Coverage = 10

2. *zero (A, B, C, D, E, F) :- chess(G, 3, H, 1, I, 1)*

   Accuracy = 72 Coverage = 8

3. *zero (A, B, C, D, E, F) :- chess(c, G, a, H, a, I)*

   Accuracy = 63 Coverage = 4

This is the hypothesis generated for depth zero. The hypothesis generated by Bain does not cover any negative examples because that system can generate new predicates. A new predicate such as *greater_than(X,Y)* that calculates the difference between two variables can be introduced. With this predicate if a rule is generated such that the difference between two variables *G* and *H* should be greater than 1 will make the rule 2 in the above hypothesis give an accuracy of 100 percent. In order to come close to this hypothesis, we tried to generate rules that have 100 percent accuracy and cover no negative examples. We were successful in that, and obtained 10 rules. These rules when clubbed together is the hypothesis generated by Bain.

The following hypothesis is for depth one:

1. *one (A, B, C, D, E, F) :-chess(c, G, H, 3, a, 2)*

   Accuracy = 81 Coverage = 10

2. *one (A, B, C, D, E, F) :-chess(c, 2, G, 4, a, 1)*

   Accuracy = 88 Coverage = 6

3. *one (A, B, C, D, E, F) :-chess(c, 2, G, 5, a, 1)*

   Accuracy = 88 Coverage = 5

4. *one (A, B, C, D, E, F) :-chess(c, 2, G, 6, a, 1)*

   Accuracy = 88 Coverage = 6

5. *one (A, B, C, D, E, F) :-chess(c, 2, G, 7, a, 1)*

   Accuracy = 88 Coverage = 6

6. *one (A, B, C, D, E, F) :-chess(c, 2, G, 8, a, 1)*

   Accuracy = 88 Coverage = 6

7. *one (A, B, C, D, E, F) :-chess(c, G, a, H, b, 1)*

   Accuracy = 75 Coverage = 5

8. *one (A, B, C, D, E, F) :-chess(c, G, e, H, d, 1)*

   Accuracy = 75 Coverage = 5

9. *one (A, B, C, D, E, F) :-chess(c, 2, G, 4, a, 3)*

   Accuracy = 75 Coverage = 5

10. *one (A, B, C, D, E, F) :-chess(d, 3, b, G, c, 1)*

    Accuracy = 75 Coverage = 5

11. *one (A, B, C, D, E, F) :-chess(d, 3, f, G, e, 1)*

    Accuracy =75 Coverage = 5

Similar to hypothesis of depth zero, this hypothesis also produces results that are similar to the hypothesis of Bain except for the coverage of negative examples. The hypothesis covers all the positive examples. Similar to the case of depth zero, this hypothesis can also be generated without covering any negative examples, and can later be clubbed together to form the hypothesis generated by Bain. For example, rules 2 to 8 can be clubbed together to form one rule which says that the column 4 can be anything between 4 to 8.

It is evident that the hypothesis of this system is very similar to what was generated by Bain in his thesis except that his system will invent new predicates. Invention of new predicates is not a feature of our system and hence there are slight deviations in hypotheses

obtained. However, apart from the rules above, our system was also able to generate a hypothesis that does not cover any negative examples. It can be used as a mechanism to prove that given the right features, we could generate a hypothesis that is on par with the hypothesis generated by traditional mechanism.

The results show that the system produces very few deviations from that of the traditional systems that use Prolog. The deviations are minor and can be ignored as the ultimate result of generating a hypothesis by covering all the positive examples with as few negative examples as possible is satisfied.

5 Conclusions and Future Work

ILP programs have been using prolog for a very long time. These programs do not have the capability to deal with the real-world datasets that are in the database format. These datasets are of interest as they have some interesting problems to be solved by ILP. This thesis aims at providing an easy-to-use technique for ILP users by integrating ILP system with database. The thesis provides a novel approach of storing the facts as database tables and rules as database views by joining on the common arguments present in multiple predicates.

The results show that the hypothesis generated by the system presented in this thesis is very similar to the hypothesis generated by the traditional ILP systems. The slight variations in the results are expected as this thesis employs a best-first search approach and it may not always result in the output that is similar to systems that do not use best-first search. Nevertheless, the results are promising for an ILP user, who now can apply ILP techniques to many varied real-world datasets that are already in the database format.

The future work for this thesis can be implementing other ILP algorithms by using the existing platform provided. There can be a user interface that could be added to easily provide input by the user. The current algorithm will not invent new predicates and this could a future enhancement. The program can be made more robust by testing it on many more diverse datasets. Apart from these extensions, more research could be put into checking if there is a more efficient way to implement ILP systems in database.

Bibliography

[1] K. M. Ali and M. J. Pazzani. ``HYDRA: A noise-tolerant relational concept learning algorithm". In: *IJCAI*. Citeseer. 1993, pp. 1064–1071 (cit. on p. 31).

[2] M. Bain. ``Learning logical exceptions in chess". PhD thesis. Citeseer, 1994 (cit. on p. 62).

[3] S. Boytcheva. ``Overview of inductive logic programming (ILP) systems". In: *Cybernetics and Information Technologies* 1 (2002), pp. 27–36 (cit. on p. 11).

[4] I. Bratko and S. Muggleton. ``Applications of inductive logic programming". In: *Communications of the ACM* 38.11 (1995), pp. 65–70 (cit. on p. 8).

[5] L. De Raedt, N. Lavrac, and S. Dzeroski. ``Multiple predicate learning". In: *Proceedings of the 13th international joint conference on artificial intelligence*. 1993, pp. 1037–1042 (cit. on p. 33).

[6] S. Džeroski. *Inductive logic programming and knowledge discovery in databases*. American Association for Artificial Intelligence, 1996 (cit. on pp. 16, 17).

[7] S. Dzeroski and N. Lavrac. *Inductive logic programming: Techniques and applications*. 1994 (cit. on pp. 8, 12, 14, 30).

[8] J. Fürnkranz. ``FOSSIL: A robust relational learner". In: *Machine Learning: ECML-94*. Springer. 1994, pp. 122–137 (cit. on p. 31).

[9] *Introduction to Logic Programming*. https://bernardopires.com/2013/10/try-logic-programming-a-gentle-introduction-to-prolog/. Accessed: 2015-07-18 (cit. on p. 6).

[10] D. Kazakov and D. Kudenko. ``Machine learning and inductive logic programming for multi-agent systems". In: *Lecture notes in computer science* (2001), pp. 246–270 (cit. on p. 21).

[11] S. Kramer and B. Pfahringer. *Inductive Logic Programming: 15th International Conference, ILP 2005, Bonn, Germany, August 10-13, 2005, Proceedings*. Vol. 3625. Springer, 2005 (cit. on p. 22).

[12] C. Leckie and I. Zukerman. ``Inductive learning of search control rules for planning". In: *Artificial Intelligence* 101.1 (1998), pp. 63–98 (cit. on p. 32).

[13] S. Muggleton. ``Inductive logic programming". In: *New generation computing* 8.4 (1991), pp. 295–318 (cit. on p. 1).

[14] S. Muggleton and W. Buntine. ``Machine invention of first-order predicates by inverting resolution". In: *Proceedings of the fifth international conference on machine learning*. 1992, pp. 339–352 (cit. on p. 19).

[15] S. Muggleton and L. De Raedt. ``Inductive logic programming: Theory and methods". In: *The Journal of Logic Programming* 19 (1994), pp. 629–679 (cit. on pp. 3, 7, 19).

[16] S.-H. Nienhuys-Cheng and R. de Wolf. ``The subsumption theorem in inductive logic programming: Facts and fallacies". In: *Advances in Inductive Logic Programming* (1996), pp. 265–276 (cit. on p. 13).

[17] M. Pazzani and D. Kibler. ``The utility of knowledge in inductive learning". In: *Machine learning* 9.1 (1992), pp. 57–94 (cit. on p. 29).

[18] G. D. Plotkin. ``A note on inductive generalization". In: *Machine intelligence* 5.1 (1970), pp. 153–163 (cit. on pp. 13, 16).

[19] J. R. Quinlan. ``Learning logical definitions from relations". In: *Machine learning* 5.3 (1990), pp. 239–266 (cit. on pp. 19, 24).

[20] *Sequential Covering Algorithm*. http://www.d.umn.edu/~rmaclin/cs8751/Notes/L08_Rule_Learning.pdf. Accessed: 2015-07-18 (cit. on p. 22).

[21] A. Srinivasan. *The aleph manual*. 2001 (cit. on p. 33).

[22] A. Srinivasan, S. Muggleton, R. D. King, and M. J. Sternberg. ``Mutagenesis: ILP experiments in a non-determinate biological domain". In: *Proceedings of the 4th international workshop on inductive logic programming*. Vol. 237. Citeseer. 1994, pp. 217–232 (cit. on p. 58).

[23] *TRY LOGIC PROGRAMMING! A GENTLE INTRODUCTION TO PROLOG*. http://www.cse.buffalo.edu/faculty/alphonce/.OldPages/CPSC312/CPSC312/Lecture/LectureHTML/CS312_1.html. Accessed: 2015-07-18 (cit. on pp. 3, 5).

[24] Wikipedia. *Inductive logic programming --- Wikipedia, The Free Encyclopedia*. [Online; accessed 18-July-2015]. 2015. url: https://en.wikipedia.org/w/index.php?title=Inductive_logic_programming&oldid=655331238 (cit. on p. 4).

[25] J. Wogulis and M. J. Pazzani. ``A methodology for evaluating theory revision systems: Results with Audrey II". In: *IJCAI*. 1993, pp. 1128–1134 (cit. on p. 30).