Three companions for data mining in first order logic

Luc De Raedt, Hendrik Blockeel, Luc Dehaspe, Wim Van Laer

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
email: Luc.DeRaedt@cs.kuleuven.ac.be

Abstract. Three companion systems, CLAUDIEN, ICL and TILDE, are presented. They use a common representation for examples and hypotheses: each example is represented by a relational database. This contrasts with the classical inductive logic programming systems such as PROGOL and FOIL. It is argued that this representation is closer to attribute value learning and hence more natural.

Furthermore, the three systems can be considered first order upgrades of typical data mining systems, which induce association rules, classification rules or decision trees respectively.

1 Introduction

Typical data mining algorithms employ a limited attribute value representation, where each example consists of a single tuple in a relational database. This representation is inadequate for problem-domains that require reasoning about the structure of objects in the domain and relations among such objects, such as in bio-chemistry [8], natural language processing [15], and traffic control [18]. This paper presents three companion systems, where each example corresponds to a small relational database (or Prolog knowledge base). Hence, examples consist of multiple relations and each example can have multiple tuples for these relations. This setting is known in the literature as *learning from interpretations*. It contrasts with the classical inductive logic programming setting employed by the systems PROGOL [20] and FOIL [22], cf. [11]. We will show that our representation is a natural upgrade of attribute value representations that is effective for inductive logic programming.

Current data mining approaches are often distinguished on the basis of their predictive or descriptive nature. In predictive data mining one is given a set of examples or observations that are classified into a finite number of classes. Typically, there are two (or more) classes, one that is called positive, and the other that is negative. The aim then is to induce a hypothesis that correctly classifies all the given (and unseen) examples. Consider Figure 1(a), where one is given two types of examples (+ and -). H is a correct hypothesis as it correctly discriminates the positives from the negatives. The purpose of predictive data mining is thus to generate hypotheses that can be used for classification.

Fig. 1. Predictive versus descriptive induction

Common predictive data mining techniques include decision tree induction (e.g., C4.5 [21]) and rule induction (e.g., CN2 [10,9] or AQ [19]).

In descriptive data mining one is given a set of unclassified examples and the aim is to find regularities within these examples. Furthermore, it is the aim to characterize as much as possible the given examples. Therefore as many properties or regularities as possible are derived. Together they form a kind of most specific hypothesis that covers (explains) all the examples. E.g. in Figure 1(b), the hypothesis H characterizes all the given examples. The most popular descriptive data mining technique is that of discovering association rules [2,1,24].

The three companion systems, CLAUDIEN [13,12], ICL [14] and TILDE [5,6], can be considered first order upgrades of existing attribute-value data mining approaches. Claudien upgrades the descriptive association rule approach, ICL upgrades the predictive production rule approach as incorporated in, e.g., CN2 [10,9] and AQ [19], and TILDE is a recent upgrade of Quinlan's popular predictive C4.5 algorithm for decision tree induction [21].

This chapter is organised as follows: Section 2 introduces the representational framework and its relation to attribute value learning. In Section 3, the basic algorithms of the three companion systems CLAUDIEN, ICL and TILDE are introduced. Section 4 illustrates how we can use these three systems on an application and discusses some practical aspects. In Section 5 and 6, we illustrate briefly the three companions on the mutagenesis problem, and give an exercise on an artificial problem from a series of problems called *Bongard problems*. Section 7 concludes.

2 Representation

2.1 Attribute value representations

Imagine that you have just been hired by a professional seminar organizer PSO in order to discover new knowledge about the activities of PSO that is to be used for commercial purposes. PSO also informs you that they have a database about past activities. Part of this database is listed in Figure 2.

It contains information about participants in a recent Seminar on Data Mining. For each of the participants, a number of attributes are stored: Name, Job, Seniority, Company, Party (whether the person attend the party or not), and Rnumber (Registration number). From talking to the marketing people you are

PARTICIPANT Table					
NAME	JOB	SENIORITY	COMPANY	PARTY	R_NUMBER
adams	researcher	junior	scuf	no	23
blake	president	junior	jvt	yes	5
king	manager	junior	ucro	no	78
miller	manager	senior	jvt	yes	14
scott	researcher	senior	scuf	yes	94
turner	researcher	junior	ucro	no	81

Fig. 2. Table of participants of a seminar.

informed that the variable *Party* is important. So, you decide to find out what type of people attend the parties at such seminars. (This is useful in order to set the price of the party as well as to decide upon the activities at parties).

Being familiar with attribute value approaches to data mining, you decide to carry out experiments with the predictive decision tree and production rule algorithms as well as with the descriptive association rule approach. Within the attribute value representation, each participant corresponds to an example. In predictive approaches, each example will be described by a vector *participant(Job, Seniority, Company)* in addition to the Class-variable *Party* as you decide that *Name* and *Rnumber* are bad indicators for the variable *Party*. Running your favorite rule-induction algorithm could then result in the following production rules:

Party=*yes* if Seniority=*senior*
 Party=*yes* if Job=*president*

Using a Prolog representation, one could use:

```
party(yes) :- participant(_J, senior, _C)
party(yes) :- participant(president, _S, _C)
```

Using the latter representation, in order to predict whether a participant will attend the party or not, you simply assert the tuple describing the participant together with the above hypothesis in the Prolog-database and run the query ?*party(yes)*. If the query succeeds¹, the participant will attend the party; otherwise she won't.

Alternatively, you may induce a decision tree (see Figure 3) for this predictive data mining problem. For the purposes of this chapter, we'll assume that all tests in the decision trees are boolean (or binary). To use the tree for classifying

¹ A query succeeds if and only if values for its variables exist such that all the conditions in the query are true.

Fig. 3. A possible decision tree for the Participant table in Figure 2.

a participant, one first tests whether $\text{Seniority}=\text{senior}$; if it is, one classifies the participant as going to the party; otherwise one tests whether $\text{Company}=\text{jvt}$. If it is, one classifies as $\text{Party}=\text{yes}$, otherwise as $\text{Party}=\text{no}$. Again, we could represent the decision tree by a Prolog program :

```

party(yes) :- participant(_J,senior,_C), !.
party(yes) :- participant(_J,_S,jvt), !.
party(no).
```

To classify an example one then again has to assert the tuple corresponding to the participant in the Prolog knowledge base containing the above hypothesis, and then run the query $\text{?-party}(_X)$. Prolog will then compute the value of $_X$: *yes* or *no*.

Alternatively, you may decide to use the association rule approach to discover all interesting characteristics of people that go to the party. Running a descriptive data mining algorithm on this subset of the original dataset then results in rules such as the following:

```

false if Company=ucro
Job=president or Seniority=senior
...
```

or in Prolog:

```

false :- participant(_J,_S,ucro)
_J=president;_S=senior :- participant(_J,_S,_C)
...
```

The first rule reveals no one from company *ucro* will attend the party. The second rule expresses the constraint that, within the subgroup going to the party, all participants are either presidents or senior staff. Therefore PSO may decide to increase the price of the party (since it will be attended anyway by the higher personnel) or decrease the price (in order to attract also the lower ranked personnel).

The representations introduced above have two drawbacks. Firstly, there is only a fixed feature-value representation of examples. This makes it impossible to represent structural information about the examples (cf. Exercise on Bongard Problem in Section 6). Secondly, no additional background information about the examples was provided. We discuss each of these drawbacks in turn, and show how they can be alleviated.

2.2 First order representations

Examples A slightly more realistic version of our previous illustration is shown in Figure 4 (the attribute Seniority is deleted in table Participant and an extra table Subscription is added). The main distinction between the previous database and the current one is that properties of each person are now scattered around the whole database. In particular, information about each person is contained in multiple tables of the database. Furthermore, some tables contain multiple tuples for some persons. The question now is how this database can be turned into a set of examples. To obtain a set of examples, one merely has to partition the database into examples, each of which corresponds to a single person. This is shown in Figure 5(a) and (b). E.g. the partition corresponding to *adams* can be obtained by first selecting all tuples having Name=*adams* (see (a)), and then projecting on all attributes (excluding Name). Each partition thus obtained constitutes a *model* or an interpretation (see (b))².

PARTICIPANT Table					SUBSCRIPTION Table	
NAME	JOB	COMPANY	PARTY	R_NUMBER	NAME	COURSE
adams	researcher	scuf	no	23	adams	erm
blake	president	jvt	yes	5	adams	so2
king	manager	ucro	no	78	adams	srw
miller	manager	jvt	yes	14	blake	cso
scott	researcher	scuf	yes	94	blake	erm
turner	researcher	ucro	no	81	king	cso
					king	erm
					king	so2
					king	srw
					miller	so2
					scott	erm
					scott	srw
					turner	so2
					turner	srw

Fig. 4. The registration database with two tables.

² The generation of these models from the database is intuitively straightforward. It can also be automatised; we do not go into the details of the algorithm here but refer to [4].

PARTICIPANT Table					SUBSCRIPTION Table	
NAME	JOB	COMPANY	PARTY	R_NUMBER	NAME	COURSE
adams	researcher	scuf	no	23	adams	erm
blake	president	jvt	yes	5	adams	so2
king	manager	ucro	no	78	adams	srw
miller	manager	jvt	yes	14	blake	cso
scott	researcher	scuf	yes	94	blake	erm
turner	researcher	ucro	no	81	king	cso
					king	erm
					king	so2
					king	srw
					miller	so2
					scott	erm
					scott	srw
					turner	so2
					turner	srw

(a)

```
begin(model(adams)).
  participant(researcher,scuf,no,23).
  subscription(erm).
  subscription(so2).
  subscription(srw).
end(model(adams)).
```

(b)

Fig. 5. Partition corresponding to Adams.

Given the new representation, we could again run our favorite algorithms. However, before doing so, we will first consider the issue of background knowledge.

Background knowledge It is often useful to employ additional knowledge in the induction process. Let us start, e.g., with global information. Consider Figure 6, which is an extension to the database in Figure 4. The information in the tables `Company` and `Course` could be helpful when generalizing from examples. However, the information contained in these tables is not only meaningful to a single example, it contains knowledge relevant to all examples. This kind of knowledge will be included as a separate background knowledge file (see the `company` and `course` facts in Figure 7(a)), and will be considered part of each example (or partition).

Secondly, it may be useful to encode derived knowledge using inference rules (such as those offered by Prolog or using, e.g., views in a relational database). Such knowledge can take various forms: e.g., abstraction of specific values into a

PARTICIPANT Table					SUBSCRIPTION Table	
NAME	JOB	COMPANY	PARTY	R_NUMBER	NAME	COURSE
adams	researcher	scuf	no	23	adams	erm
blake	president	jvt	yes	5	adams	so2
king	manager	ucro	no	78	adams	srw
miller	manager	jvt	yes	14	blake	cso
scott	researcher	scuf	yes	94	blake	erm
turner	researcher	ucro	no	81	king	cso
					king	erm
					king	so2
					king	srw
					miller	so2
					scott	erm
					scott	srw
					turner	so2
					turner	srw

COMPANY Table		COURSE Table		
COMPANY	TYPE	COURSE	LENGTH	TYPE
jvt	commercial	cso	2	introductory
scuf	university	erm	3	introductory
ucro	university	so2	4	introductory
		srw	3	advanced

Fig. 6. The complete registration database with several tables.

taxonomy or interval (e.g., course-weight in Figure 7(b)), deriving new properties from a combination of existing ones, summarizing or aggregating values of several tuples into a single value (e.g., *sum_due* in Figure 7(b)), views on a specific attribute (e.g., *job*, *party*,... in Figure 7(a)) etc. Rules for *course_weight* (which is *hi* if the course is advanced or longer than 3, otherwise it is *low*) and *sum_due* (the amount of money a participant must pay, depending on the courses followed) can be found in Figure 7(b). Again this form of background knowledge will be included in the background knowledge file. To reason about examples, one then asserts the background theory and the partition in a Prolog knowledge base, and asks queries in Prolog. E.g. to reason about *adams*, one would assert the *model(adams)* in Prolog together with background. Then *adams* would have the following property: *sum_due(1000)*.

Taken all together, the database in Figure 6 can be partitioned as in Figure 8. The background is common to all examples.

First order hypotheses In the previous section, we merely adapted the representation of examples (and background knowledge) to first order logic. Given the representation of examples and background knowledge it is however easy to upgrade the representations for production rules, association rules and decision trees. One merely has to allow for condition parts of Prolog rules that allow for conjunctions of literals (or relations).

E.g. to predict the class *party*, one could induce: ($\backslash +$ stands for Prolog's *not* operator)

```
party(yes) :- company_type(commercial).
party(yes) :- course_len(_S, 4), \+subscription(_S).
```

```

begin(background).
    company(jvt,commercial).
    company(scuf,university).
    company(ucro,university).
    course(cso,2,introductory).
    course(erm,3,introductory).
    course(so2,4,introductory).
    course(srw,3,advanced).
    job(_J) :- participant(_J,_,_).
    party(_P) :- participant(_P,_,_P,_).
    company_type(_T) :-
        participant(_C,_,_), company(_C,_T).
    course_len(_C,_L) :- course(_C,_L,_).
    course_type(_C,_T) :- course(_C,_,_T).
end(background).

```

(a)

```

begin(background).
    <...>
    course_weight(_C,_Weight) :-
        course(_C,_L,_T),
        ((_L > 3; _T = advanced), !, _Weight = hi; _Weight = lo).
    sum_due(_SumDue) :-
        findall(_L, (subscription(_C), course(_C,_L,_)), _List),
        sum(_List,_Sum),
        company_type(_CType),
        (_CType = university, !, _SumDue is _Sum * 100
        ; _SumDue is _Sum * 1000).
    sum([], 0).
    sum(_H|_T, _S2) :- sum(_T, _S1), _S2 is _S1 + _H.
end(background).

```

(b)

Fig. 7. Background for the registration database.

Fig. 8. The registration database in Prolog.

Similarly, for decision trees:

With association rules, we can induce properties of participants that go the party, such as:

`job(manager) ; subscription(_S) :- course_len(_S,3) , course_type(_S,introductory).`

This rule says that the considered participants are either managers or they subscribe to all introductory courses of length 3.

Learning from interpretations The above introduced knowledge representation formalism for hypotheses and examples is quite natural. First, notice that attribute value representations are a straightforward special case. Secondly, notice also that examples roughly correspond to partitions of the database. As a consequence, each example can be considered independently of the other ones.

This is important in the light of data mining where one could be given a vast numbers of examples, which excludes the possibility of loading the dataset into main memory (cf. [16]). Thus one potentially gains efficiency. Thirdly, we believe this representation is also quite intuitive and elegant (cf. Exercise on Bongard Problem in Section 6).

3 Three companion systems

We now introduce the basic algorithms that underlie the three companion systems ICL, TILDE and CLAUDIEN.

3.1 ICL: Inductive Classification Logic

The ICL system upgrades the well-known CN2 algorithm by Clark and Niblett [10, 9] towards inductive logic programming. The CN2 and ICL algorithms will induce a set of production rules such as those listed in Section 2. A simplified algorithm is outlined in Table 1.

Table 1. A simplified outline of CN2 and ICL.

```

find_hypothesis(P,N)
  H := ∅
  while P is not empty do
    find_rule(P,N,class(positive) :- body)
    add class(positive) :- body to H
    remove from P all examples covered by class(positive) :- body .

find_rule(P,N, class(positive) :- body)
{finds a rule class(positive) :- body that covers some positives in P}
  body := true
  while ?-body succeeds for a negative example in N do
    for all literals lit do
      compute the heuristic value of class(positive) :- body,lit
      body := body, l where l is the literal that scores best

```

The first loop of the algorithm is the so-called "covering" algorithm. It starts off with all positive examples in P and all negatives in N. It will then repeatedly attempt to find a single rule that covers some positives and none of the negatives. Each such rule is added to the hypothesis *H* and all positives covered by this rule will be removed from P. This process terminates when all positives are covered.

To find one such rule, the algorithm performs a top-down search process, whereby it starts from the most general rule `class(positive) :- true` that covers all examples, and repeatedly refines this rule by heuristically choosing the best condition to add. This process terminates when the current rule does not cover any negative example.

Consider the WORN example in Figure 9. Suppose we are learning the de-

Fig. 9. The WORN example with two classes.

scription of class *positive*. Then ICL would start off by the following rule:

```
class(positive) :- true
```

It would then discover that this also classifies the negatives as positive and would therefore consider refinements of this rule including :

```
class(positive) :- worn(_X)
class(positive) :- not_replaceable(_X)
etc.
```

Suppose it then chooses the first rule (because it excludes more negatives than the second one). Again, it would notice that it is too general, and start refining. It could then come up with the following correct hypothesis :

```
class(positive) :- worn(_X), not_replaceable(_X).
```

This rule correctly classifies all examples.

It should be mentioned that the above description of ICL and CN2 is largely simplified. First, it may be the case that the data are noisy in which case a perfect solution may not exist. Under such conditions the algorithms will employ heuristics to find an approximation of the target concept (cf. [14]). Second, the

find-rule algorithm as outlined above performs hill-climbing : it keeps track of a single current best candidate. ICL and CN2, however, perform a beam-search, where one always keeps track of the m best rules. Finally, the above algorithm does not specify which literals can be added to the hypothesis. This is discussed in Section 4.1 on $\mathcal{DLAB}$.

Some additional features of ICL include: it can learn hypotheses in DNF as well as in CNF format, it has some number handling capabilities (using discretization), it can learn multi-classes in case there are more than 2 classes (we use a similar strategy as in CN2, but have also a Bayesian method included), it has some built-in evaluation methods (e.g., 10-fold cross-validation),... (see also [26]). Several settings can be used to tune the heuristics and a non-graphical interface is integrated that makes the use of ICL fairly easy.

3.2 TILDE: Top-down induction of logical decision trees

TILDE is a member of the popular top-down induction of decision tree family of algorithms. However, rather than using attribute-value tests in nodes of the tree, it employs logical queries. Furthermore, as in Example WORN in Figure 10, a variable name can occur in different nodes. The meaning then is that

Fig. 10. The WORN example with three classes and a corresponding tree.

all occurrences denote the same variable. This slightly complicates the classification process with first order decision trees. E.g. to classify an example with

the WORN tree, one first tests whether $?-worn(_X)$ succeeds, if it does not, if belongs to class **keep**. Otherwise, one should test whether $not_replaceable(_X)$ succeeds. However, since the $_X$ in $not_replaceable(_X)$ already appears higher in the tree, this test corresponds to testing $?-worn(_X), not_replaceable(_X)$. Only if this query succeeds, the class is **sendback**; otherwise, it is **fix**. Hence, an equivalent Prolog program corresponding to the WORN tree is:

```
class(sendback) :- worn(\_X), not\_replaceable(\_X), ! .
class(fix) :- worn(\_X), ! .
class(keep).
```

The induction algorithm underlying TILDE is similar to C4.5. TILDE will start off with the empty tree. It will then generate all possible tests and compute the heuristic value of these test. Among the possible tests, it will select the one that scores best on the heuristic and place that in the current node. It will then use the partial tree to classify all examples (in the current node) to its sub-nodes. All examples passing the test will be propagated to the left, all the other ones to the right. The procedure will then recursively analyze the left and right subtrees. When a node only contains examples of a single class (or when heuristics indicate it is uninteresting to split a node), it will be turned into a leaf.

Figure 11 illustrates the algorithm on Example WORN.

Fig. 11. TILDE illustrated on the WORN example. Each step shows the partial tree that has been built, the literals that are considered for addition to the tree, and how each literal would split the set of examples. E.g. *ffss/k* means that of four examples, one with class **fix** and two with class **sendback** are in the left branch, and one example with class **keep** is in the right branch. The best literal is indicated with an asterisk.

3.3 Claudien : Clausal Discovery

The Claudien algorithm searches for a set of clausal regularities that hold on the set of examples. Each such clause is of the form

$$h_1; \dots; h_n : -b_1, \dots, b_m$$

A clause of this form expresses that whenever the condition part (the conjunction of the literals b_i holds), that also the conclusion part holds (i.e. at least one of the h_i holds). A clause holds on an example whenever it is the case that if $?-b_1, \dots, b_m$ succeeds with a substitution θ then also $?-h\theta$ succeeds.

The clausal discovery algorithm Claudien does not employ the well-known two-phased algorithm for induction of association rules used for instance in APRIORI [1] (but see [16] for an upgrade of APRIORI). Instead, it employs an algorithm that directly enumerates all the relevant clauses.

The key idea underlying the CLAUDIEN algorithm is the following: whenever a clause does not hold on an example, it must be that there exists a substitution θ such that $?-(b_1, \dots, b_n)\theta$ holds but $(h_1; \dots; h_n)\theta$ does not hold. There are two ways to solve this problem. One can either modify the clause by adding a literal h to its head (for which $h\theta$ might hold), or by adding a literal b to its body (for which $b\theta$ might not hold).

This idea then results in the simple algorithm outlined in Table 2. The algorithm starts with a queue containing *false* : *-true* as the only clause. This clause does not hold in any example. In the while loop of the algorithm, CLAUDIEN then repeatedly deletes a clause from the queue and tests whether it holds on the data. If it does, it is added to the final hypothesis, otherwise it is refined (by adding a literal to it). The search space of the algorithm for a simple clausal discovery problem is illustrated in Figure 12, where m, f, and h denote male, female, and human, respectively.

Table 2. A simplified outline of CLAUDIEN.

```

Q := { false :- true }
H := ∅
while Q is not empty do
  delete c from Q
  if c covers all p in P (and H does not entail c)
  then add c to H
  else add all refinements of c to Q

```

Fig. 12. The search space of CLAUDIEN.

4 Practical use: getting started

In this section we use the registration application to guide you through a first session with the three systems CLAUDIEN, ICL and TILDE. Basically, they all require as inputs a knowledge base, a language bias, and some parameter settings.

Once you have constructed the knowledge base as explained in Section 2.2, this serves as input to all three systems. Assuming *app* is the name of the application, this input should be stored in a file called *app.kb*. For the definition of language bias, CLAUDIEN and ICL use *DLAB*; this definition is put in an input file *app.1*. Parameter settings are put in *app.s*. Most of these settings are system specific. Some settings apply to more than one of the companion systems. These are:

- **talking(*n*)** specifies how much information a system writes to the screen while inducing hypotheses. *n* ranges from 0 (no information) to 4 (very detailed). In the experiments described in this text **talking(4)** is used, so that the induction process can be followed closely. The default setting is system dependent.
- **classes(*L*)** defines the classes (only for the classification systems ICL and TILDE). *L* is a list of atoms C_i ; an example belongs to class C_i iff a call to C_i succeeds in the example. The default setting is **classes([pos,neg])**.

Note that for TILDE, the language bias (specified using **rmode** settings) is included in the file *app.s*.

All systems write their output to a file called *app.out*. On-line help can be obtained in each system by typing **help**.

4.1 *DLAB*: a language bias formalism for CLAUDIEN and ICL

The language bias in CLAUDIEN and ICL is defined by means of one or more so-called *DLAB* templates, written as Prolog facts with a single argument of type string. Several *DLAB* templates are shown in Figure 13(a). Such a template

Fig. 13. Examples of $\mathcal{DLAB}$ templates.

is a compact notation for a finite, but potentially very large, set of clauses. Figure 13(a) represents a single clause in a trivial way. We now make the definition of possible clause bodies intensional by combining more literals in a list, and by labelling this list with a discrete interval $Min - Max$ that constrains the minimum and maximum number of elements that can be taken from the list. For instance, the template in Figure 13(b) represents the three clauses with the literal `party(yes)` in the head and 1 or 2 literals from the list `[job(researcher), job(manager)]` in the body. In the interval specification “len” indicates the length of the associated list, as in the template in Figure 13(c), which represents 9 clauses, with three different heads, and, as before, three different bodies. Exactly the same set of clauses is obtained with the template in Figure 13(d), in which the interval specifications have moved inside to the level of terms. To exclude the apriori inconsistent conjunction `job(researcher)` and `job(manager)` we should alter the template as shown in Figure 13(e).

The examples above illustrate the basics of $\mathcal{DLAB}$. Two more advanced examples show the use of nesting and `dlab_variables`. Nesting contributes to the expressiveness, as in Figure 13(f), where the template represents 11 clauses, each with 1 or 2 literals in the body. No literal occurs twice, due to the “exclusive or” enforced by the nested 1-1 interval. As a second advanced example, the template in Figure 13(g), represents the same set of clauses, and shows how `dlab_variables` (not to be confused with logical variables that can occur in the template) contribute to readability.

Figure 14 further shows the search space that corresponds to a `dlab_template` presented to CLAUDIEN and ICL. In this example, p stands for party, j for job, r for researcher, m for manager, p for president, ct for company_type, c for commercial and u for university. The dotted lines indicate paths followed by ICL and not by CLAUDIEN. Note that CLAUDIEN’s search is optimal in that it never visits the same node twice.

4.2 Running CLAUDIEN

CLAUDIEN requires a knowledge base, in file *registration.kb* and a language, in file *registration.l*. The CLAUDIEN manual [17] contains full details on additional

Fig. 14. The search space corresponding to a *DLAB* template.

optional components such as *registration.s* with non-default settings. We briefly mention some of the more important settings.

- *min_accuracy(R)*. If this is set to 1, CLAUDIEN will only accept solutions that are valid on all data. By lowering the accuracy threshold this strong requirement can be relaxed.
- *min_coverage(N)* specifies the required minimum number of examples covered by the clause.
- *search(S)* specifies the search strategy: best-first, beam, breadth-first or depth-first.

If you want CLAUDIEN to discover clauses with at most two literals in the head that express dependencies between job, company type, party, course length, course type you might for instance create a file **registration.l** as shown in Figure 15(a). CLAUDIEN then starts up as shown in Figure 15(b). Note that the number of clauses in the search space is 71712. The start up phase ends with the display of the prompt **claudien-registration>**. Since no settings file is present, default settings apply.

To initiate the discovery process, issue the command **c** or **claudien**, and CLAUDIEN will respond as shown in Figure 15(c). The initial, most general clause in the search space is in this case **false if true**, which is not accepted as a solution but is added to the queue for further refinement. In the second step the first descendant **job(researcher) if true** is evaluated. As this clause is true in half of the examples (there are 3 researchers), its accuracy is 50% (**a(0.5)**), the body succeeds in 6 examples (**t(6)**), the clause succeeds in 3 examples (**p(3)**) and there are 3 cases where the clause fails (**n(3)**). Accuracy **a(0.5)** and coverage **t(6)** (cf. confidence and support in association rule discovery) have to be higher than user-specified thresholds. By default these thresholds are both set to one. Since the accuracy is not sufficient, this clause is not accepted as a solution, but

Fig. 15. A session with CLAUDIEN.

added back to the queue for further refinement. After each level in the breadth first search (alternative (heuristic) search strategies are available) the number of candidates in the queue is shown, in this case there are 20 nodes in the queue after level 2. The first solution is `company_type(university);party(yes)` if `true` which says that every participant either works at a university or goes to the party. Solutions are not further refined. In general, CLAUDIEN will prune away nodes that for some reason cannot lead to “interesting” clauses. For instance, `job(researcher) if job(researcher)` is pruned because it is a tautology and will remain so when it is refined. In total 61 (out of 71712) clauses are selected for being “interesting”.

All solutions are written to `registration.out`. Figure 15(d) shows an extract from this file. For instance, rule 60 says that university participants either do research or take all introductory courses. The body of this rule succeeds in 4 examples (`total(4)`), and in all of these also the entire clause succeeds (`pos(4)`), hence an accuracy of 100%. Apart from the solutions, `registration.out` also contains some search statistics, such as the percentage of search space visited, that is the share of nodes not pruned away, in this case 9.52%.

4.3 Running ICL

The input files for ICL are similar to those of CLAUDIEN:

- The **knowledge** base (file `registration.kb`) contains *classified* examples.

- The **settings** file *registration.s* can be used to change the value of some parameters (settings) by adding a fact *name(value)*, with *name* the name (identifier) of the setting and *value* the new value. For each setting there is a default value.

Some useful settings are:

- **significance_level(*SL*)** specifies the confidence level (as percentage) for the significance test; possible values include 0 (no significance testing), 0.80, 0.90 (default), 0.95 and 0.99; a higher percentage will prune more rules.
 - **beam_size(*N*)** with *N* the maximum number of rules to be kept in the beam (default is 5).
- The **language** bias is specified with *DLAB* in *registration.l* (see Section 4.1 for an introduction to *DLAB*). An important difference to *CLAUDIEN* is that the *DLAB* bias should not contain the class predicates and the templates should be read as `dlab_template('[negative literals] <-- [positive literals]')`. *ICL* allows the user to generate `dlab_variable/3` facts automatically by using `dlab_query/3` (a feature of *DLAB* added by *ICL*). The first 2 arguments of `dlab_query` are the same as in `dlab_variable`, the third is either `constants(Q, V)` or `discretize(Q, V)` (*Q* can be any predicate containing the variable *V* and is used to generate values for *V* that occur in the examples). The predicate `constants` returns the list of all possible values for *V*, while `discretize` returns a small subset of all numeric values by using a discretization algorithm (details in [26]). The returned list is used as the third argument in the corresponding `dlab_variable`.

For the *registration* application, we would like to classify participants (the examples) into people who attend the party and people who don't. So in the file *registration.s* we set `classes([party(yes), party(no)])`. A possible *DLAB* for *registration.l* can be found in Figure 16. This is very similar to the one used in *CLAUDIEN*, with the exception that the `party` predicates have been removed. The file *registration.kb* is the same as for *CLAUDIEN* and *TILDE*.

ICL is started as displayed in Figure 17 (a)³. When everything is loaded successfully, the `icl>` prompt will be displayed from which you can type commands. Type `help` to get a list of available commands (see Figure 17 (b)). Quick help on a command can be obtained with the command `help(command)`.

The *ICL* learning algorithm is started with the command `icl` (or just `i`) as shown in Figure 18(a). The system will try to find a hypothesis for each class. During search, information is written on the screen. Some parts of this are shown in Figure 18(a)-(d). It seems that a correct theory has been found for class *party(no)*:⁴

$$H = (\text{party}(\text{no}) :- \text{company_type}(\text{university}), \text{subscription}(_S), \text{course_len}(_S, 4)).$$

³ *ICL* will ask for a *configuration* name at start up. We will not discuss this feature in this text, so just type `enter`. More information can be found on the *ICL* web pages.

⁴ Note that the actual output of *ICL* omits the head of the rules

```

dlab_template(' 0-2: [ job(c_job),
                    company_type(c_company_type),
                    subscription(_S),
                    course_len(_S,c_course_len),
                    course_type(_S,c_course_type)
                  ] <--
0-len: [ job(c_job),
        company_type(c_company_type),
        subscription(_S),
        course_len(_S,c_course_len),
        course_type(_S,c_course_type)
      ]').
dlab_variable(c_job, 1-1, [researcher,manager,president]).
dlab_variable(c_company_type, 1-1, [commercial,university]).
dlab_variable(c_course_len, 1-1, [2,3,4]).
dlab_variable(c_course_type, 1-1, [introductory,advanced]).

```

Fig. 16. The language bias file `registration.l` for ICL.

```

% ICL registration
*****
*   ICL : Inductive Classification Logic   *
<...>
Now loading files for application registration with configuration
**Initialising knowledge**
loading registration.kb
  o there is no optimized file registration.kb.O.wic
  o using non-optimized registration.kb ...
-> 6 models have been loaded
there is no backgroundfile registration.bg
preparing knowledge...
**Initialising settings**
set settings to their default value ...
loading registration.s ...
**Initialising language**
loading registration.l ...
**Initialising theory**
ok

```

(a)

```

Starting interactive session
*****
For list of commands : h/0 or help/0
icl> help
The following commands are available:  %useful for tutorial
help/0 - h          help/1 - h          quit/0 - q
icl/0 - i           icl/1 - i

show_theory/0 - st   showf_theory/0 - sft   write_theory/0 - wt
show_stat_theory/0 - sst   write_stat_theory/0 - wst
show_complexity_theory/0 - sct

show_settings/0 - ss   show_settings/1 - ss   show_info/0 - si
set/2 - s             load_settings/0 - ls   set_default_settings/0 - sds

new_config/0 - nc     new_config/1 - nc     read_theory_file/0 - rtf
test_language/0 - tl

```

(b)

Fig. 17. Starting ICL for the *registration* application.

For *party(yes)* however, no theory has been found. When lowering the significance level however, ICL will find a theory for class *party(yes)*. The result is shown in Figure 18 (e).

During discovery, results and information on the learning process are written to *registration.out*. With the command `write_theory`, the learned theories are written in the file *registration.theory* so that it can be (re)loaded later on. `write_statistics` will write statistics (like accuracy) of the learned theories w.r.t. the training examples and the test examples (if any). Test examples are those examples that are left out during learning (using the setting `leave_out`).

4.4 Running TILDE

The TILDE system uses exactly the same format for the knowledge base as the other systems, but it does not need the *app.1* file. The language bias is not specified using the DLAB formalism, but by means of settings in *app.s*.

Most of TILDE's settings have default values that need not be changed. We only discuss the most important settings here, and refer to TILDE's user's manual [3] for a more detailed discussion.

- `minimal_cases(n)` specifies how many examples each leaf must cover. Leaves covering fewer examples are not allowed. The default for *n* is 2; for very small data sets it may be appropriate to set this to 1.
- `rmode` settings tell TILDE what kind of tests can be put in the nodes of the tree. They partially define the language bias. We discuss them in more detail below.
- `lookahead` settings specify in what way TILDE can perform a local search to generate tests for a node. This is also discussed in detail below.

Rmode settings For TILDE, the hypotheses space is defined mostly by what kind of tests can be put in the nodes of the tree. A set of `rmode` facts is used to specify this. Basically, `rmode(C)` tells TILDE that *C* is allowed as a test. *C* can be a literal, a conjunction of literals or a more complicated construct.

Recalling the *WORN* example in Figure 10, tests that could be allowed in the nodes of the tree are:

`worn(wheel)` does the machine contain a wheel that is worn?
`worn(X)` (if *X* a new variable) does the machine contain some worn part *X*?
 (if *X* already exists) is *X* worn?

Some predicates may require an argument to be a variable that already occurs, others need a new variable, and some may allow for both. So we distinguish three different *modes* for variables: *input* if the variable must already exist in order for it to be used here, *output* if it should not exist, or *any* if it can be either an existing or a new variable.

```

icl> icl
ICL will run for the following classes : [party(no),party(yes)]
*****
* NEXT CLASS *
*****
* CLASS : party(no)
* Number of negative models : 3
* Number of positive models : 3
* =====
local(PT, PF, NT, NF)
P: positives N: negatives
T: true    F: false

*** Find next BEST RULE ***
*****
** positive models still left : [madams,mking,mturner]
<...>
ooo Refining : ROOT
(job(president)) # heur(0.333333, cpu(0.03), local(0,3,1,2), total(0,3,1,2)
*PRUNED* : too few positive covered

```

(a)

```

<...>
(company_type(university)) # heur(0.666667, cpu(0.04),
local(3,0,1,2), total(3,0,1,2) * try to ADD to BEAM : ADDED
<...>
#### Starting next level BEAM search ####
ooo Refining ooo : (company_type(university)) # ...
<...>
(job(researcher),company_type(university)) # heur(0.6), cpu(0.39),
local(2,1,1,2), total(2,1,1,2)
*PRUNED* : refinements not possibly significant
<...>
ooo Refining ooo : (course_len(_S,4),+company_type(commercial))
# heur(0.666667), cpu(0.53), local(3,0,1,2), total(3,0,1,2)
#### NEW BEST #### with heuristic value 0.8
(subscription(_S),course_len(_S,4),+company_type(commercial))
# heur(0.8), cpu(0.56), local(3,0,0,3), total(3,0,0,3)
*PRUNED* : refinements can't become better

```

(b)

```

*****
***Best Rule***
*****
(subscription(_S),course_len(_S,4),+company_type(commercial))
# heur(0.8), cpu(0.68), local(3,0,0,3), total(3,0,0,3)
*****
FINISHED : no positive models left
Done after 0.69 cpu seconds
<...>
***** STATISTICS for ICL run *****
<...>
* Nodes pruned for syntactic reasons: 54
* Nodes pruned for semantic reasons : 187
* Nodes kept : 52
* Nodes visited: 293
<...>

```

(c)

```

* NEXT CLASS *
* CLASS : party(yes)
<...>
finished : there are still positive models left...
<...>
For this run, the following theory has been found:
*****
*Class : party(no)
*Status : complete
*Type : dnf
rule((company_type(university),subscription(_S),course_len(_S,4)),
[type(dnf),cpu(1.2),heur(0.8),local(3,0,0,3),total(3,0,0,3)])
*****
*Class : party(yes)
*Status : complete
*Type : dnf
*****
icl>

```

(d)

```

icl> set(significance_level,0.8)
icl> icl
<...>
*****
*Class : party(no)
*Status : complete
*Type : dnf
*****
rule((company_type(university),subscription(_S),course_len(_S,4)),
[type(dnf),cpu(0.51),heur(0.8),local(3,0,0,3),total(3,0,0,3)])
*****
*Class : party(yes)
*Status : complete
*Type : dnf
*****
rule((company_type(commercial)),
[type(dnf),cpu(0.45),heur(0.75),local(2,1,0,3),total(2,1,0,3)])
rule((course_len(_S,4),+subscription(_S)),
[type(dnf),cpu(0.82),heur(0.666667),local(1,0,0,3),total(2,1,0,3)])

```

(e)

Fig. 18. A session with ICL on the *registration* database.

Rmode facts tell TILDE which tests can be used, and if the tests contain variables, indicate the modes of these variables. *Input*, *any* and *output* variables are indicated by writing a +, – or nothing before them, respectively.

The following example shows what kind of tests are generated by certain **rmode** specifications. *A* represents a variable that already occurs in the tree, *B* is a new variable.

```
rmode(1:worn(gear)). worn(gear)
rmode(5:worn(X)).    worn(B)
rmode(5:worn(+X)).   worn(A)
rmode(5:worn(-X)).   both worn(A) and worn(B)
```

The numbers inside **rmode** indicate how many times it is allowed to occur in any path from the root of the tree to a leaf (i.e. how many times at most it can be tested for one example).

If there are many constants, it is tedious to write similar tests for each of these constants (e.g., **worn(gear)**, **worn(engine)**, ...). Fortunately, TILDE can generate suitable constants itself. It does this in the following way: for each example in the set of examples covered by the node, a query is made that generates one or more constants. Each of these constants is then filled in in the test that is to be put in this node. In order to keep the branching factor of the search space limited, maxima can be given for the number of examples TILDE should look at, as well as for the number of constants that can be generated from each example.

To tell TILDE that a test **worn(c)** can be used, with *c* being any constant for which **worn** could possibly succeed, the following **rmode** fact could be used:

```
rmode(5: #(15*5*X: worn(X), worn(X))).
      a  d c    b      e
```

This specification means that in at most 15 (a) examples, TILDE should run **worn(X)** (b) and see which values *X* (c) can take; it should return at most 5 (d) values per example. Finally, the test **worn(X)** (e) will be put in the node, but with *X* changed into one of the constants: **worn(gear)**, **worn(engine)**, ...

In the above example, the constant generating predicate is the same as the test predicate, but this need not always be the case. Another example of the use of constant generation is:

```
rmode(10: #(100*1*C:boundary(C), +X < C)).
```

In at most 100 models one numeric boundary will be computed, and a test should consist of comparing an already bound variable *X* with this boundary. The computation of a suitable boundary can be defined in background knowledge. It might be done by, e.g., a discretization algorithm.

While the above syntax is a bit awkward, it is very general and allows the generation of constants in many different settings.

Lookahead. In some cases it is possible that a literal will never be chosen by TILDE, because it is not useful as a test. However, such a literal, while not useful in itself, might introduce new variables that make better tests possible later on.

As an example, let us look at the registrations database. In order to allow TILDE to use information about subscriptions, we could provide the following `rmode` facts:

```
rmode(5: subscription(X)).
rmode(5: #(1*3*L: member(L, [2,3,4]), course_len(+X,L)).
```

With only these specifications TILDE will not find a tree where the first test is whether the participant has subscribed to something, and the second test is whether she has subscribed to a course of some specific length. The reason for this is that `subscription(X)` in itself, as a *test*, is never informative: every person has subscribed to at least one course. Thus, this test will never be chosen. Consequently, the test `course_len(X,2)` cannot be chosen either, because it can only test a variable that already occurs.

This problem can be solved by allowing TILDE to *look ahead* in the search space, i.e. to check, immediately after putting `subscription(X)` in a node, whether any tests making use of *X* are interesting.

In the above case one can add to the `rmode` facts one `lookahead` fact:

```
lookahead(subscription(X),
          #(1*3*L: member(L, [2,3,4]), course_len(X,L))).
```

TILDE will then build a test for one node by first considering `subscription(X)`, then `subscription(X), course_len(X,2)` and so on for 3 and 4. All these tests will be evaluated separately, and the best test is chosen. Note that both literals are put in one single node of the tree.

Sample run. We have now seen all the settings we need to get TILDE running on the registrations data set. Figure 19 shows what a suitable `registration.s` would look like. Once this file has been prepared, TILDE can be started by just typing `tilde` at the prompt. The induction is started by typing `go`. Upon finishing, TILDE creates a file `registration.out` containing among other things the tree that has been built, as is shown in Figure 19.

5 Sample application: Mutagenesis

The data in the mutagenesis domain (see [23]) consists of 188 molecules, of which 125 are active (thus mutagenic) and 63 are inactive. A molecule is described by listing its atoms `atom(AtomID,Element,Type,Charge)` (the number of atoms differs between molecules, ranging from 15 to 35) and the bonds `bond(Atom1,Atom2,BondType)` between atoms.

In [23], four different backgrounds have been defined. We will illustrate our three systems on the mutagenesis data with the simplest of these (BG1 in [23]).

Fig. 19. Settings file, screendump and output file of TILDE on the registration database.

Figure 20 shows a snapshot of the knowledge base file *muta.kb*. Examples of settings, language and result files can be found in Figures 21, 22 and 23. To experiment on this data yourself, you can retrieve the necessary input files from our website (see Section 7).

Figure 21 illustrates a setup for CLAUDIEN. The size of the language shown in Figure 21(a) is in the order of 10^8 . The settings file in Figure 21(b) is added to enforce a heuristic, best first search and accuracy and coverage thresholds of respectively 90% and 20 molecules. One of the first discovered rules, after four cpu minutes, is shown in Figure 21(c). This rule has an accuracy of 90% and the body of the rule holds for 62 of the 188 molecules.

muta.kb

```
begin(model(1)).
pos.
atom(d1_1,c,22,-0.117).      bond(d1_1,d1_2,7).
atom(d1_2,c,22,-0.117).      bond(d1_2,d1_3,7).
atom(d1_3,c,22,-0.117).      bond(d1_3,d1_4,7).
atom(d1_4,c,195,-0.087).      ...
...
end(model(1)).
```

Fig. 20. Snapshot of the knowledge base file for the mutagenesis data.

```

dlab_template('1-1: [pos,neg] <--
0-len: [len-len: [atom(A1,Elem1,Type1,Charge1),
1-1: [element(Elem1),atomtype(Type1)],
0-len: [len-len: [atom(A2,Elem2,Type2,Charge2),
1-1: [element(Elem2),atomtype(Type2)],
0-len: [len-len: [atom(A3,Elem3,Type3,Charge3),
1-1: [element(Elem3),atomtype(Type3)]
]
]
]
]
1-1:[bond(A1,A2,BT12),len-len:[bond(A1,A2,BT12),bondtype(BT12,c_bt)],
1-1:[bond(A1,A3,BT13),len-len:[bond(A1,A3,BT13),bondtype(BT13,c_bt)],
1-1:[bond(A2,A3,BT23),len-len:[bond(A2,A3,BT23),bondtype(BT23,c_bt)]
]').

dlab_variable(element,1-1,[br,c,cl,f,h,i,n,o,s]).
dlab_variable(atomtype,1-1,[a1,a10,a14,a16,a19,a194,a195,a21,a22,a230,a232,a25,a26,a27,a28,a29,a3,a31,
a32,a34,a35,a36,a38,a40,a41,a42,a45,a49,a50,a51,a52,a72,a8,a92,a93,a94,a95]).
dlab_variable(c_bt,1-1,[1,2,3,4,5,7]).

```

(a)

```

min_accuracy(0.9).
min_coverage(20).
search(best).
talking(1).

```

(b)

```

rule(2,[accuracy(0.9),total(62),pos(56),neg(6),comp(26),cpu(238.4),realtime(696)],{
pos :- atom(A1,Elem1,Type1,Charge1), a27(Type1),
atom(A2,Elem2,Type2,Charge2), c(Elem2),
atom(A3,Elem3,Type3,Charge3), a38(Type3),
bond(A2,A3,BT23)
}).

```

(c)

Fig. 21. CLAUDIEN applied to the mutagenesis data set. (a) language bias; (b) settings file; (c) extract from the results file.

For ICL, we use the same language file `muta.1` as the one for CLAUDIEN in Figure 21(a), except for `1-1: [pos,neg]` in the `dlab_template` (which must be replaced by `false`). The settings file `muta.s` contains `classes([pos,neg])`. ICL learns a DNF theory for both classes *pos* and *neg*. For class *neg*, the theory in Figure 22 has been found in about 81 seconds. The accuracy of ICL (estimated by 10-fold cross-validation) ranges from 80.9% (for BG1) up to 88.3% (for BG4) for multi-class theories.

TILDE's language bias definition is quite different from that of the other systems, as Figure 23 shows. With this language⁵, which allows TILDE to use only the background information in BG1, TILDE needs approximately 18 seconds (on a SUN Sparc Ultra-2 machine) to output the tree shown in Figure 23. A 10-fold cross-validation using the same language yields 75% accuracy.

6 An exercise

This exercise is based on some pattern recognition tasks, invented by the Russian scientist M. Bongard [7]. A number of drawings are given, each labelled with $\oplus$ or $\ominus$. The task is to find a theory that allows to discriminate between both classes based on the objects in the drawings.

Figure 24 shows a number of these drawings. Drawings can contain circles, squares and triangles. For triangles the direction in which they point (up or

⁵ No other settings need to be given here.

```

*****
*Class : neg
*Status : complete
*Type : dnf
*****
rule((atom(A1,Elem1,Type1,Charge1),a34(Type1),atom(A2,Elem2,Type2,Charge2),a10(Type2)),
     [type(dnf,cpu(11.87),heur(0.852246),local(7,56,0,125),total(7,56,0,125))])
rule((atom(A1,Elem1,Type1,Charge1),a50(Type1),atom(A2,Elem2,Type2,Charge2),a22(Type2),
     atom(A3,Elem3,Type3,Charge3),n(Elem3),bond(A2,A3,BT23)),
     [type(dnf,cpu(24.91),heur(0.820786),local(10,46,1,124),total(10,53,1,124))])
rule((atom(A1,Elem1,Type1,Charge1),a35(Type1),atom(A2,Elem2,Type2,Charge2),a22(Type2),
     atom(A3,Elem3,Type3,Charge3),a22(Type3),bond(A1,A2,BT12),bond(A2,A3,BT23)),
     [type(dnf,cpu(33.43),heur(0.81003),local(5,41,0,125),total(6,57,0,125))])
rule((atom(A1,Elem1,Type1,Charge1),a41(Type1)),
     [type(dnf,cpu(40.65),heur(0.667553),local(2,39,0,125),total(2,61,0,125))])
rule((atom(A1,Elem1,Type1,Charge1),a93(Type1),atom(A2,Elem2,Type2,Charge2),n(Elem2),
     bond(A2,A3,BT23),bondtype(BT23,1)),
     [type(dnf,cpu(47.8),heur(0.667553),local(2,37,0,125),total(2,61,0,125))])
rule((atom(A1,Elem1,Type1,Charge1),a32(Type1),atom(A2,Elem2,Type2,Charge2),a22(Type2)),
     [type(dnf,cpu(67.45),heur(0.52234),local(15,22,13,112),total(16,47,13,112))])
rule((atom(A1,Elem1,Type1,Charge1),cl(Elem1)),
     [type(dnf,cpu(73.99),heur(0.458777),local(3,19,3,122),total(8,55,3,122))])
rule((atom(A1,Elem1,Type1,Charge1),a21(Type1),atom(A2,Elem2,Type2,Charge2),a34(Type2)),
     [type(dnf,cpu(80.66),heur(0.445035),local(2,17,2,123),total(9,54,2,123))])

```

Fig. 22. ICL applied to the mutagenesis data set.

muta.s	muta.out
<pre> lookahead(bond(A1,A2,BT), #(1*7*C: member(C,[1,2,3,4,5,7]),BT=C)). lookahead(bond(A1,A2,BT), #(230*9*E: atom(A2,E,_),atom(A2,E,_))). lookahead(bond(A1,A2,BT), #(230*37*T: atom(A2,_T,_),atom(A2,_T,_))). rmode(20: #(230*37*T: atom(_,_T,_),atom(A,E,T,Ch))). rmode(20: #(230*9*E: atom(_,_E,_),atom(A,E,T,Ch))). rmode(20: bond(+A1,-A2,BT)). </pre>	<pre> atom(A,B,41,C)? +--yes: neg[2/2] +--no: atom(D,E,50,F)? +--yes: neg[10/12] +--no: atom(G,H,35,I)? +--yes: neg[5/6] +--no: atom(J,K,34,L)? +--yes: bond(J,M,N),atom(M,c,O,P)? +--yes: neg[9/10] +--no: pos[2/2] +--no: atom(Q,R,93,S)? +--yes: neg[5/7] +--no: atom(T,U,1,V)? +--yes: atom(W,X,14,Y)? +--yes: pos[4/5] +--no: neg[14/24] +--no: atom(Z,A_1,92,B_1)? +--yes: atom(C_1,D_1,29,E_1)? +--yes: pos[2/2] +--no: neg[4/5] +--no: pos[100/113] </pre>

Fig. 23. TILDE applied to the mutagenesis data set.

Fig. 24. Bongard problems

down) may be relevant. Finally, whether an object is inside another object may also be relevant. So, the aim is to find a theory that discriminates positive from negative drawings by looking at the kind of objects they consist of, their configuration (for triangles, up or down) and their relative positions (objects inside one another).

You might want to take a look at Figure 24 to see whether you can find a good rule yourself. There exists a rule predicting examples to be positive that is correct and almost complete (i.e. the condition of the rule holds for nearly all positives and for no negatives) for the examples in this drawing. If you find a good rule, then you can run the inductive systems and see whether they yield the same rule, better rules, or worse ones.

With respect to using the inductive systems on these data, we suggest the following actions:

- First think about a representation for these drawings. Write down some examples from the drawing.
- Download the whole data set (`bongard.kb`, > 300 examples) from the URL given below. Compare the representation used in this file with yours.
- Try to run CLAUDIEN, ICL and/or TILDE on these data sets. You will need to construct the `bongard.l` and `bongard.s` files yourself now. See what results you get, try to improve them.
- If you are satisfied with your results or get stuck, download the example `bongard.s` and `bongard.l` files from the above site. Compare them with your files.
- Finally, run the systems with the settings and language files you downloaded. See whether you get better or worse results than with your files. Also compare these results with the ones you have found manually.

7 Conclusions and practical info

We have presented three companion systems for first order data mining. They all learn from interpretations and use the same or similar inputs. This tutorial is meant as a gentle introduction to the three systems and their use. However, we wish to stress that all three systems have a number of additional features that were not mentioned here. These are described in the manuals [17, 3, 25].

CLAUDIEN, ICL and TILDE are freely available (for academic use) via the internet. Information on how to obtain them can be found at the following URL:

<http://www.cs.kuleuven.ac.be/~ml/Tutorial/>

This web page also contains links to the data files mentioned in this text.

Acknowledgements

Luc De Raedt and Wim Van Laer are supported by the Fund for Scientific Research, Flanders. Hendrik Blockeel is supported by the Flemish Institute for the Promotion of Scientific and Technological Research in the Industry (IWT). This work is supported by the European Community Esprit project no. 20237, Inductive Logic Programming 2.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. The MIT Press, 1996.
2. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 207–216. ACM, Washington, D.C., USA, May 1993.
3. H. Blockeel. Tilde 1.3 user's manual, 1997. <http://www.cs.kuleuven.ac.be/~ml/Tilde/>.
4. H. Blockeel. *Top-down induction of first order logical decision trees*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1998. <http://www.cs.kuleuven.ac.be/~ml/PS/blockeel98:phd.ps.gz>.
5. H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, pages 77–85. Springer-Verlag, 1997.
6. H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
7. M. Bongard. *Pattern Recognition*. Spartan Books, 1970.
8. I. Bratko and S. Muggleton. Applications of inductive logic programming. *Communications of the ACM*, 38(11):65–70, 1995.
9. P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Proceedings of the Fifth European Working Session on Learning*, pages 151–163. Springer-Verlag, 1991.

10. P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–284, 1989.
11. L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.
12. L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann, 1993.
13. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.
14. L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, pages 80–94. Springer-Verlag, 1995.
15. L. Dehaspe and L. De Raedt. Mining a natural language corpus for multi-relational association rules. In *Proceedings of the Workshop on Empirical Learning of Natural Language Processing Tasks*, pages 35–48, 1997. Invited keynote paper.
16. L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, pages 125–132. Springer-Verlag, Berlin, 1997.
17. L. Dehaspe, L. De Raedt, and W. Van Laer. Claudien : a clausal discovery engine: a user manual. Technical Report CW-239, Department of Computer Science, Katholieke Universiteit Leuven, 1996.
18. S. Džeroski, N. Jacobs, M. Molina, C. Moure, S. Muggleton, and W. Van Laer. Detecting traffic problems with ILP. In *Proceedings of the Eighth International Conference on Inductive Logic Programming*. Springer-Verlag, 1998.
19. R. Michalski. A theory and methodology of inductive learning. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning: an artificial intelligence approach*. Morgan Kaufmann, 1983.
20. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
21. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
22. J. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
23. A. Srinivasan, S. Muggleton, M. Sternberg, and R. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1,2), 1996.
24. H. Toivonen, M. Klemettinen, P. Ronkainen, K. Hättönen, and H. Mannila. Pruning and grouping discovered association rules. In *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, pages 47–52, Heraklion, Crete, Greece, 1995.
25. W. Van Laer. Web pages for the ICL system, 1997.
<http://www.cs.kuleuven.ac.be/~ml/ICL/>.
26. W. Van Laer, L. De Raedt, and S. Džeroski. On multi-class problems and discretization in inductive logic programming. In *Proceedings of the Tenth International Symposium on Methodologies for Intelligent Systems*, pages 277–286. Springer-Verlag, 1997.