USER MANUAL
Learning Hybrid Relational Dependency Networks with Learner of Local Models (LLM)

Irma Ravkic Jan Ramon Jesse Davis

1 Introduction

This work is positioned in the field od Statistical Relational Learning (SRL) which is concerned with developing formalisms for representing and learning from data that exhibit both uncertainty and complex, relational structure. What has received little attention in SRL is learning these relationships from hybrid data: data having both discrete and continuous variables. We developed Learner of Local Models (cite) that enables users to learn hybrid relationships from the data by learning separate local models and then combining them in a bigger network. These local models, or Conditional Probability distributions (CPDs), quantify the dependencies by giving the probability for the value of target atoms given the values of their parent variables. The underlying formalism we use for representation and learning are Relational Dependency Networks(cite): thus, we learn Hybrid Relational Dependency Networks (HRDNs).

In this manual we will guide the user on how to use our system. First we will give an overview of the components and different possibilities when using the algorithm. Each component will have illustrating examples. Throughout the guidance we will use the famous University example.(cite)

2 Learning Hybrid Relational Dependency Networks

This section is intended to give a brief overview of the representation of RDNs and HRDNs. The user is not obliged to read this section but it can give him some association with the implementation. Also, if the user wishes to know more on the foundation, he/she should refer to the paper (cite).

The representation of HRDNs has a lot in common with RDNs. The only difference is that in HRDNs we allow random variables and relational features to have numeric ranges. This however also necessitates the usage of more adequate CPDs to quantify these hybrid dependencies. The CPDs mostly used for RDNs are Conditional Probability Tables (CPTs) or Relational Probability Trees (RTPs). In HRDNs we cannot use these CPDs for numeric dependencies as they perform discretization. In this section we will the representation of (H)RDNs and discuss what kind of CPDs we use.

2.0.1 Representation

There are several ways to define (H)RDNs, but we use a definition that uses first-order logic as a template language for constructing propositional dependency networks. We first briefly review the relevant concepts from first-order logic, then we define HRDNs.

The alphabet consists of three types of symbols:

- **Constants**
  A constant represents a specific object and is denoted with a lower-case

letter (e.g., `pete`).

- **Logical variables (logvars)**
  A logical variable (*logvar*) `X` is a variable ranging over the objects in the domain. Logical variables may be *typed* in which case they represent placeholders for a specific subset of objects in the domain. For example we can specify that a logvar with symbol `S` in `intelligence(S)` ranges over students, people in general, that is, over anyone we define or allow to have intelligence.

- **Predicates**
  Predicates represent properties of objects or relations among objects. They have an associated arity (number of arguments). For example, predicate `intelligence/1`, can represent a property of a student (intelligence), and it is a property of one object (arity 1). An example of a relation predicate is for instanfe `friendship/2` and it holds between two people.

  Each predicate has the range. The range of predicates in (H)RDNs is not restricted to $\{false, true\}$ as in classical logic. For example, the range of a student's intelligence could be $\{\texttt{low}, \texttt{med}, \texttt{high}\}$. Unlike in RDNs, the range of predicates in HRDNs can be numeric, that is an interval of real numbers. For example, we can specify the range of a student's intelligence to be interval $[0, 180]$.

- **Atoms**
  An *atom* is of the form $\texttt{P}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{n})$ where `P/n` is a predicate and each $\texttt{t}_\texttt{i}$ is an object or a logvar. The range of an atom is the range of its predicate. A literal is an atom or its negation. An atom is *ground* if all its arguments are constants. For example, if `intelligence/1` is a predicate with arity 1 with numeric range $[0, 180]$ an atom can be `intelligence(S)`, and it has the same range as `intelligence/1`.

We specify **random variables** in the domain through random variable declarations (RVDs):

$$random(\texttt{H}) \leftarrow l_1, \ldots, l_n$$

where `H` is an atom, and $l_1, \ldots, l_n$ is a conjunction of literals. This is called a random variable declaration. For example, the random variable declaration for the atom `takes(S,C)`

$$random(\texttt{takes(S,C)}) \leftarrow \texttt{student(S)},\ \texttt{course(C)} \tag{1}$$

creates one randvar for each student `S` and course `C` in the domain.

To specify how one random variable depends on the other we use **relational features**. In RDNs we call these features *discrete relational feautures* because their range is discrete. On the other hand, in HRDNs *numeric relational features*

have a continuous range. In case a feature represents the property of multiple objects we use an *aggregation* function. For example, *mode* is an aggregation function that maps a multiset of values to the most frequently occurring value in the multiset. Mode is a discrete feature. On the other hand, *average* is an aggregation function that maps a multiset of numeric values to their average value which means that average is a numeric feature.

In the end what we care about is to learn *dependency statements* and *CPDs* quantifying these dependency statements. A *hybrid* dependency statement is of the form $\texttt{G} \mid \text{Parents}(\texttt{G})$ where $\texttt{G}$ is the *target* atom that has a discrete or numeric range and whose arguments are all logvars. $\text{Parents}(\texttt{G})$ is a set of discrete or numeric relational features. An HRDN is a tuple $(\mathcal{P}, RVD, dep)$, where $\mathcal{P}$ is a set of predicates, whose ranges may be discrete or continuous, RVD is a set of randvar declarations and dep is a function mapping each $\texttt{P} \in \mathcal{P}$ to a hybrid dependency statement.

Note that RDNs consist only of *discrete dependency statements*. This means that $\texttt{G}$ has only discrete range and that all relational features in $\text{Parents}(\texttt{G})$ are discrete features. In this work we mostly focus on HRDNs and not RDNs so in further discussions we part from RDNs and we will only discus how to learn hybrid dependency statements in HRDN and CPDs associated to them.

2.0.2 Local Distributions (CPDs)

As mentioned earler each dependency statement $\texttt{G} \mid \text{Parents}(\texttt{G})$ has an associated CPD. The type of model used for a CPD depends on both the range of the target atom $\texttt{G}$ and whether $\text{Parents}(\texttt{G})$ contains discrete or numeric features.

In this work, we use a *parametric* approach to density estimation and focus only on variants of Gaussian distributions to model continuous variables. Specifically, we use the following models:

Multinomial If $\texttt{G}$ has a discrete range and its parent set is empty, the CPD is modeled by a multinomial distribution.

Gaussian If $\texttt{G}$ has a continuous range and its parent set is empty, the CPD is modeled by a Gaussian distribution.

Logistic Regression (LR) This CPD is used when the target atom has a discrete range as it facilitates incorporating both discrete and continuous parents.

Linear Gaussian (LG) A linear Gaussian CPD is used when $\texttt{G}$'s range is continuous and all the features in the parent set are numeric [**?**, **?**]. An LG is a Gaussian distribution that models μ as a linear combination of the values of the features in the parent set, but assumes a fixed variance σ^2.

Conditional Linear Gaussian (CLG) A conditional linear Gaussian (CLG) is used if $\texttt{G}$'s range is continuous and its parents set contains a mix of discrete and numeric features. There is a separate linear Gaussian model for every instantiation of the discrete parents. A *conditional Gaussian* is a

special case of a CLG where the parent set only contains discrete features. Here, a separate Gaussian (mean and variance) is learned for each possible configuration of the parents.

For a detailed overview of the local distributions refer to the paper (cite) and section 3.2.

2.0.3 Learner of Local Models (LLM): algorithm overview

In this section we present our algorithm for learning the structure of an HRDN. This requires learning a dependency statement and CPD for each predicate in the domain. It is possible to use a decomposable score function to evaluate candidate structures. Thus the problem can be tackled by independently learning a locally optimal CPD for each predicate. Therefore, we refer to our approach as the *Learner of Local Models (LLM)*. When learning the CPD for each predicate, we define a space of candidate features and then greedily select those that improve the score.

High-Level Control Structure

Algorithm 1 outlines LLM and it receives as input a set of predicates $\mathcal{P}$, a set of training interpretations D, and a set of validation interpretations V. LLM assumes fully-observed data. Next, we describe in detail how to learn and evaluate local distributions.

Algorithm 1: LLM(Predicates $\mathcal{P}$, Training data D, Validation data V)

$M = \{\}$
for all $\mathtt{P} \in \mathcal{P}$ **do**
  $CPD_P = \text{LearnOneModel}(\mathtt{P}, \mathcal{P}, D, V)$
  $M = M \cup \{(P, CPD_{\mathtt{P}})\}$
end for
return: $(\mathcal{P}, M)$

Learning local distributions

Each learned CPD, regardless of its form, in an HRDN is parameterized by a set of features. Learning the structure of the CPD requires determining which features should appear in the parent set. This can be posed as the problem of searching through the space of candidate features. We adopt a greedy approach that selects one feature at a time to add to the parent set until no inclusion improves the score. Thus, in each iteration, the central procedure is finding the single best feature and adding it to the parent set.

Algorithm 2 outlines our procedure for learning the dependency for a predicate P. As input, it receives the target predicate P, the full set of predicates $\mathcal{P}$ for the domain, a training set D, and a validation set V. First, the algorithm starts by constructing the set of candidate features for P. Second, it repeatedly iterates through the set of candidate features and evaluates the utility of adding each feature to the parent set. Each feature addition is followed by learning the

CPD on the training data D and then scoring it on the validation data V. In each iteration, the single best feature is added to the parent set. If no feature improves the score, the procedure terminates.

Algorithm 2: LearnOneModel(Target Predicate P, All Predicates $\mathcal{P}$, Training Data D, Validation Data V)

$Parents(\mathtt{P}) = \emptyset$
$CPD_{\mathtt{P}}$ = learnCPD($Parents(\mathtt{P})$, D)
FS = GenerateCandidateFeatures(P, $\mathcal{P}$)
repeat
  $F_{best} = null$
  $CPD_{best} = CPD_{\mathtt{P}}$
  for F in FS **do**
    CPD_{temp} =learnCPD($Parents(\mathtt{P}) \cup \{F\}$, D)
    if $score(CPD_{temp}, V) > score(CPD_{\mathtt{best}}, V)$ **then**
      $CPD_{best} = CPD_{temp}$
      $F_{best} = F$
    end if
  end for
  if $F_{best} \neq null$ **then**
    $Parents(\mathtt{P}) = Parents(\mathtt{P}) \cup \{F_{best}\}$
    $CPD_{\mathtt{P}} = CPD_{best}$
    $FS = FS \setminus \{F_{best}\}$
  end if
until $F_{best} = null$
return: $CPD_{\mathtt{P}}$

In the following section we will cover the pipeline and all necessary components to run the experiments provided in the "examples" folder. First we will introduce all the components necessary to have an RDN:

- predicates and atoms ($\mathcal{P}$)
- random variable declatations ($\mathcal{RVD}$)
- dependencies: features and CPDs

Then we will introduce:

- data loading
- feature generator
- learning algorithm

First, we will introduce the running example we will use throughout this manual. Second, we will introduce all the components of our algorithm. And finally, we will combine everything in one example for user to try out.

3 INTERMEZZO: Running example - University example

Throughout the manual we will use the popular, but slightly modified, University example. This example has the following predicate declarations:

$$
\begin{aligned}
&random(\texttt{difficulty(C)}) \leftarrow \texttt{course(C)}\\
&random(\texttt{satisfaction(S,C)}) \leftarrow \texttt{student(S)}, \texttt{course(C)}\\
&random(\texttt{grade(S,C)}) \leftarrow \texttt{student(S)}, \texttt{course(C)}\\
&random(\texttt{takes(S,C)}) \leftarrow \texttt{student(S)}, \texttt{course(C)}\\
&random(\texttt{teaches(P,C)}) \leftarrow \texttt{professor(P)}, \texttt{course(C)}\\
&random(\texttt{friend(S,S1)}) \leftarrow \texttt{student(S)}, \texttt{student(S1)}\\
&random(\texttt{nrhours(C)}) \leftarrow \texttt{course(C)}\\
&random(\texttt{intelligence(S)}) \leftarrow \texttt{student(S)}\\
&random(\texttt{ability(P)}) \leftarrow \texttt{professor(P)}
\end{aligned}
$$

and ranges:

$$
\begin{aligned}
&range(\texttt{difficulty(C)}) = \{easy, med, hard\}\\
&range(\texttt{satisfaction(S,C)}) = \{low, med, high\}\\
&range(\texttt{grade(S,C)}) = \{low, med, high\}\\
&range(\texttt{takes(S,C)}) = \{true, false\}\\
&range(\texttt{teaches(P,C)}) = \{true, false\}\\
&range(\texttt{friend(S,S1)}) = \{true, false\}\\
&range(\texttt{nrhours(C)}) = [20.0, 180.0]\\
&range(\texttt{intelligence(S)}) = [50.0, 180.0]\\
&range(\texttt{ability(P)}) = [20.0, 100.0]
\end{aligned}
$$

where `C` represents a placeholder for courses in the domain (`course(C)`), `S`

The data we provided is obtained by sampling from the following handcrafted model:

`difficulty(C)`	
`satisfaction(S,C)`	$\mathcal{F}_{\{S,C\}:\emptyset,grade(S,C),value}$, $\mathcal{F}_{\{C\}:teaches(P,C),ability(P),value}$
`grade(S,C)`	$\mathcal{F}_{\{S\}:\emptyset,intelligence(S),value}$, $\mathcal{F}_{\{C\}:\emptyset,difficulty(C),value}$

`takes(S,C)`	$\mathcal{F}_{\{S\}:\emptyset,intelligence(S),value}$, $\mathcal{F}_{\{C\}:\emptyset,difficulty(C),value}$
`teaches(P,C)`	$\mathcal{F}_{\{P\}:\emptyset,ability(P),value}$, $\mathcal{F}_{\{C\}:\emptyset,difficulty(C),value}$
`friend(S,S1)`	$\mathcal{F}_{\{S,S1\}:\{takes(S,C),takes(S1,C)\},\emptyset,proportion}$
`nrhours(C)`	$\mathcal{F}_{\{C\}:\emptyset,difficulty(C),value}$
`intelligence(S)`	$\mathcal{F}_{\{S\}:\emptyset,grade(S,C),mode}$
`ability(P)`	

---

and following CPDs:

Predicate	Local Distribution	Parameters
`difficulty/1`	Multinomial	(0.2, 0.4, 0.4)
`satisfaction/2`	Logistic Regression	`satisfaction(S,C)`=low →(-1.28,-0.07,0.028) `satisfaction(S,C)`=med→(0.5,-0.4,0.009)
`grade/2`	Logistic Regression	`grade(S,C)`=low →(-1.77,-0.04,1.75) `grade(S,C)`=med →(-2.18,0.003,0.75)
`takes/2`	Logistic Regression	`takes(S,C)`=true →(0.4,0.009,-0.607)
`teaches/2`	Logistic Regression	`teaches(P,C)`=true →(-0.089,-0.012,0.305)
`friend/2`	Logistic Regression	`friend(S,S1)`=true →(-0.08,1.5)
`nrhours/1`	Conditional Gaussian	`difficulty(C)`=easy → N(20,6) `difficulty(C)`=med → N(50,5) `difficulty(C)`=hard → N(80,6)
`intelligence/1`	Conditional Gaussian	`grade(S,C)`=low → N(60,5) `grade(S,C)=med` → N(90,7) `grade(S,C)=high` → N(110,5)
`ability/1`	Gaussian	N(70,10)

Table 1: Local distributions used for the handcrafted model.

4 Specifying the RDN

In this section we will, through our implementation, explain the components of an RDN. Before learning the user has to state what the types, predicates and atoms there are. Note that we will use the University example as introduced in the previous section.

4.1 Specifying types

In first order logic, logvars are placeholders for objects. Additionally, a logvar can range only over a specific part of the object domain. In that case we say that the logvar is typed. We specify types in the random variable declarations of an RDN:

$$random(\texttt{satisfaction(S,C)}) \leftarrow \texttt{student(S)}, \texttt{course(C)}$$

where we state that logvar `S` and logvar `C` range over only students (`student(S)`) and courses (`course(C)`), respectively. The user specifies the types in the following way:

```
Type student=new Type("student");
Type course=new Type("course");
Type professor=new Type("prof");
```

where the constructor for Type is:

```
new Type(String symbol);
```

This means that in his/her input data the user will specify all the necessary objects in the following way:

```
...
student(peter).
student(mary).
course(math).
course(biology).
professor(drwho).
professor(balthazar).
...
```

4.2 Specifying logvars

Logvars are placeholder for objects. We use types to denote specific objects over which logvars range. The logvars are specified in the following way in our implementation:

```
Logvar S=new Logvar("S",student);
Logvar C=new Logvar("C",course);
Logvar P=new Logvar("P",professor);
```

where the constructor for Logvar is:

```
new Logvar(String symbol,Type logvar_type);
```

4.3 Specifying predicates

As we mentioned earlier predicates have predicate name, arity and range. In our implementation we support three kinds of predicates:

1. **Boolean predicates:** these predicates represent relationships between objects. As in classical logic their range is $true, false$.
2. **Predicates with categorical values:** these predicates represent an attribute or a property of one or two objects and its range is finite and categorical. For example, `grade/2` predicate in our University example has the range $\{low, med, high\}$.
3. **Predicates with real-valued values:** these predicates represent an attribute or a property of one or two objects and its range is an interval. For example, `intelligence/1` in our example has the range $[50.0, 180.0]$.

We are now going to show how to specify the predicates of the University example in our Java implementation:

```
//Boolean predicates
BooleanPred tk=new BooleanPred("takes",1);
BooleanPred tch=new BooleanPred("teaches",2);
BooleanPred fr=new BooleanPred("friend",2);
```

```
//Categorical predicates
CategoricalPred gr=new CategoricalPred("grade",2,new
    String[]{"low","mid","high"});
CategoricalPred sat=new CategoricalPred("satisfaction",2,new
    String[]{"low","mid","high"});
CategoricalPred diff=new CategoricalPred("difficulty",1,new
    String[]{"easy","medium","hard"});
```

```
//Gaussian predicates
GaussianPred intel=new GaussianPred("intelligence",1);
GaussianPred ab=new GaussianPred("ability",1,20,100);
GaussianPred nrhours=new GaussianPred("nrhours",1,0,200);
```

The user doesn't have to specify the range of categorical and Gaussian predicates, as their range can be automatically extracted from the training data.

4.4 Specifying atoms

Atoms consist of a predicate name and arguments (logvars). Therefore it's straightforward to specify them in our implementation:

```
Atom intelligence=new Atom(intel, new Logvar[]{student});
Atom grade=new Atom(gr, new Logvar[]{student,course});
Atom satisfaction=new Atom(sat, new Logvar[]{student,course});
Atom takes=new Atom(tk, new Logvar[]{student,course});
Atom ability=new Atom(ab,new Logvar[]{professor});
Atom teaches=new Atom(tch,new Logvar[]{professor,course});
Atom difficulty=new Atom(diff,new Logvar[]{course});
Atom friend=new Atom(fr,new Logvar[]{student,student});
Atom numhours=new Atom(nrhours,new Logvar[]{course});
```

4.5 Specifying the predicate declarations

Now that we have all the necessary components we can specify the network representation. This information is contained in *hybrid.network.NetworkInfo class*:

```
new NetworkInfo(new Atom[]{ability, difficulty, numhours, intelligence,
    grade, satisfaction, takes, teaches, friend}, new Type[]{stud,c,p});
```

The constructor is the following:

```
new NetworkInfo(Atom[] atoms,Type[] logvars);
```

5 Loading the data

In this section we will first give an example of the data input expected by our algorithm. Then we will introduce different ways of loading the data.

5.1 The format of the input data

The data used in our implementation comes in the form of facts ended with a period. For the domain of University, this is an example of one small interpretation:

```
course(c0).
course(c1).
course(c2).
student(s0).
student(s1).
student(s2).
professor(p0).
professor(p1).
professor(p2).
difficulty(c0,hard).
difficulty(c1,hard).
difficulty(c2,medium).
ability(p0,57.79031668756247).
ability(p1,75.53343224463569).
ability(p2,100).
nrhours(c0,78.41204621862333).
nrhours(c1,87.73823694113332).
nrhours(c2,39.695774270779225).
takes(s0,c0).
takes(s1,c0).
takes(s2,c1).
friend(s1,s0).
friend(s1,s3).
friend(s3,s0).
grade(s0,c0,high).
grade(s1,c0,high).
grade(s2,c1,high).
intelligence(s0,101.35405187696512).
intelligence(s1,109.18900881057698).
intelligence(s2,116.65135063380698).
teaches(p0,c0).
teaches(p1,c1).
```

```
teaches(p2,c2).
satisfaction(s0,c0,high).
satisfaction(s1,c0,high).
satisfaction(s2,c1,low).
```

Facts for Boolean predicates have the form of: `predicate_name(arg1)` or `predicate_name(arg1,arg2)`. A value is not needed. Facts present in an interpretation are considered true, and those that are not are false (Closed World Assumption). Facts for categorical predicates have the form of: `predicate_name(arg1 ,Value)` or `predicate_name(arg1,arg2, Value)`. The value is always positioned in the last argument. Finally, Gaussian predicate related facts are the same as categorical predicate related facts.

There are three types of data interpretations needed for our structure learning algorithm: training data, validation data and test data. Training data is used for CPD parameter estimation, validation data is used to obtain the likelihood (probablity of data given the model) and test data is used to assess the performance of the learning algorithm or learned structure. Of course, these datasets should be completely independent.

5.2 Different ways of loading the data

There are several ways of loading the data: with and without subsampling. Subsampling is useful when we have large domains and very few positive examples of specific relations and a big space of negative examples of the same. In that case we can sample the negative examples to be of the same size as the set of positive examples or in any other ratio. All of our loader use TuProlog interface to Prolog to load the facts and analyze the data. There are two types of data loader:

- With subsampling. We give it *TuPrologInterpretationCreator_Subsampling* as an argument, specifying that we want to subsample our negative examples with ratio 1. That means we will have the same amount of negative examples as the positive ones. Of course, if we put 2 we will have two times more negative examples than positive ones. And if we put 0.5 we will have the number of negative examples being half of the positive ones.

```
TuPrologDataLoader dataLoader=new TuPrologDataLoader(new
    TuPrologInterpretationCreator_Subsampling(1));
```

- Without subsampling. We give it *TuPrologInterpretationCreator_NoSubsampling* as an argument, specifying that we want to use all the negative examples during the learning.

```
TuPrologDataLoader dataLoader_no_subsampling=new
    TuPrologDataLoader(new
    TuPrologInterpretationCreator_NoSubsampling());
```

Now using these initialized loaders we can load training, validation and test data as following:

```
Data d_training= dataLoader.loadData(parameters.input_path+ "/train/",
    "interp", "pl", ntw);
Data d_validation= dataLoader.loadData(parameters.input_path+
    "/validate/", "interp", "pl",ntw);
Data d_test= dataLoader_no_subsampling.loadData(parameters.input_path+
    "/test/", "interp", "pl", ntw);
```

The general format for this is:

```
Data loadData(String pathToFiles, String name, String extension,
    NetworkInfo ntw)
```

or

```
Data loadData(String pathToFiles, String[] names_with_extension,
    NetworkInfo ntw)
```

if we want to load only specific files. Now all the interpretations are going to be stored in *hybrid.interpretations.Data* object. Each interpretaion will contain the information about the domain, ground atoms and subsampling information. Now, we are ready to define the most important part of the algorithm, querying machines as we call them.

6 Performing the structure learning

So far we have specified the network structure (all predicates and their ranges), chose the procedure of loading the data and loaded the data. Now we can proceed with structure learning. Structure learning is specified in the following way:

```
new StructureLearner(FeatureGeneratorAbstract f,StructureSearch
    struct_search,NetworkInfo ntw,QueryMachine training,QueryMachine
    validation,QueryMachine test)
```

Thus the *StructureLearner* expected the following components:

- *FeatureGeneratorAbstract* where for now we only have *FeatureGeneratorNoRestrictions* as one instance. This feature generator is constructed in the following way:

  ```
  new FeatureGeneratorNoRestrictions(int length, int
      max_num_logvars)
  ```

  where *length* denotes the maximum allowed length of feature conjunction (i.e., the number of literals in the feature), and *max_num_logvars* denotes

the maximum number of logvar renamings. This in essence determines how big the chain is to be of boolean atoms. E.g., having maximum three logvar renamings we can have the following conjunction in our feature space: `friend(S,S1),friend(S1,S2),friend(S2,S3)`. Therefore, with this value we specify how big these chains can be.

- *StructureSearch* where for now we only have *GreedySearch* as one instance. This search only has default constructor.
- *NetworkInfo* is the network info containing all the information of predicates in our problem setup (see 4.3).
- *QueryMachine* are three query machines we give to the structure learner. These are training, validation and test query machine, respectively (see 5.2).

Now we are ready to start structure learning by invoking:

```
learnStructureAndEvaluate(Atom[] atoms))
```

This list of atoms is all the atoms we want to learn the structure for. Of course the user can only learn CPD for one atom by specifyin only one atom in the list, or by setting the algorithm flag "-predicate" to the predicate name of one of the predicates in the problem.

7 The big picture

Here we give all the necessary basic components to run HRDN structure learning on your input data stored in a directory "../data/training/","../data/validate/" and "../data/test/".

7.1 Network specification

```
public class UniversityHybrid implements ApplicationNetwork {
 public NetworkInfo defineApplicationNetwork(int subsampling_ratio){
    //define types in your application
    Type stud=new Type("student");
    Type c=new Type("course");
    Type p=new Type("prof");
    //define logvars you would need to specify your initial atoms
    Logvar student=new Logvar("S",stud);
    //we will need a renaming logvar here for friend(S,S1)
    Logvar student1=new Logvar("S1",stud);
    Logvar course=new Logvar("C",c);
    Logvar professor=new Logvar("P",p);

    //define your predicates - predicate name + arity
    //Gaussian predicates
```

```
        GaussianPred intel=new GaussianPred("intelligence",1);
        GaussianPred ab=new GaussianPred("ability",1,20,100);
        GaussianPred nrhours=new GaussianPred("nrhours",1,0,200);

        //categorical predicates
        CategoricalPred gr=new CategoricalPred("grade",2,new
            String[]{"low","mid","high"});
        CategoricalPred sat=new CategoricalPred("satisfaction",2,new
            String[]{"low","mid","high"});
        CategoricalPred diff=new CategoricalPred("difficulty",1,new
            String[]{"easy","medium","hard"});

        //define your Boolean predicates
        BooleanPred tk=new BooleanPred("takes",1);
        BooleanPred tch=new BooleanPred("teaches",2);
        BooleanPred fr=new BooleanPred("friend",2);

         //Given the predicates, make atoms. They additionaly specify what
             the logvars are
        Atom intelligence=new Atom(intel, new Logvar[]{student});
        Atom grade=new Atom(gr, new Logvar[]{student,course});
        Atom satisfaction=new Atom(sat, new Logvar[]{student,course});
        Atom takes=new Atom(tk, new Logvar[]{student,course});
        Atom ability=new Atom(ab,new Logvar[]{professor});
        Atom teaches=new Atom(tch,new Logvar[]{professor,course});
        Atom difficulty=new Atom(diff,new Logvar[]{course});
        Atom friend=new Atom(fr,new Logvar[]{student,student1});
        Atom numhours=new Atom(nrhours,new Logvar[]{course});
        return new NetworkInfo(new Atom[]{ability, difficulty, numhours,
            intelligence, grade, satisfaction, takes, teaches, friend},new
            Type[]{stud,c,p});
    }

    @Override
    public HashMap<Atom, Dependency> getTrueDependencies(NetworkInfo
        ntw)throws FeatureTypeException, ConjunctionConstructionProblem {
        return null;
    }
}
```

7.2 Structure learning file

```
public class LearnUniversityHybrid {
private static NetworkInfo ntw;
    public static void main(String[] args) throws Exception{
        AlgorithmParameters parameters=new AlgorithmParameters();
          ParseArguments getAlgorithmParameters=new
              ParseArguments("Example script");
```

```
        getAlgorithmParameters.parseArgumentsHRDN(args);
        UniversityHybrid hybrid_university=new UniversityHybrid();
      ntw=hybrid_university.defineApplicationNetwork(1);
      //LOAD DATA
      TuPrologDataLoader dataLoader=new TuPrologDataLoader(new
          TuPrologInterpretationCreator_Subsampling(2));
      TuPrologDataLoader dataLoader_no_subsampling=new
          TuPrologDataLoader(new
          TuPrologInterpretationCreator_NoSubsampling());
      Data
          d_training=dataLoader.loadData(parameters.input_path+"/train/",
          "interp", "pl",ntw);
       Data
           d_validation=dataLoader.loadData(parameters.input_path+"/validate/",
           "interp", "pl",ntw);
       Data
           d_test=dataLoader_no_subsampling.loadData(parameters.input_path+"/test/",
           "interp", "pl",ntw);

       //query machines
       TuPrologQueryMachine training_data_machine=new
           TuPrologQueryMachine(d_training,
           AlgorithmParameters.getPenaltyType());
       TuPrologQueryMachine validation_machine=new
           TuPrologQueryMachine(d_validation,
           AlgorithmParameters.getPenaltyType());
       TuPrologQueryMachine test_machine=new
           TuPrologQueryMachine(d_test,
           AlgorithmParameters.getPenaltyType());

       //feature generator
       FeatureGeneratorNoRestrictions fGen=new
           FeatureGeneratorNoRestrictions(AlgorithmParameters.feature_length,
           AlgorithmParameters.nr_logvar_renamings);

       //structure learning
       StructureLearner str_learner=new StructureLearner(fGen,new
           GreedySearch(),ntw,training_data_machine,validation_machine,test_machine);
        str_learner.learnStructureAndEvaluate(ntw.getLiterals().toArray(new
            Atom[ntw.getLiterals().size()]));

  }
}
```

8 Analyzing the output data

The algorithm will output two files for each predicate with `predicate_name`:

- {predicate_name}_stat.res
- structure_learning_Log_{predicate_name}

The first file contains a short specification of the result: it gives some information on the data, domain, and then specifies the learned dependency. It also outputs weighted pseudo-loglikelihood on test and validation data together with time needed to learn the model.
The example of *_stat.res* file for `grade(S,C)` predicate is the following:

```
 DATA INFO:
 TEST DATA:
Number of interpretations : 1
Nr groundings per atom:
teaches(P,C) = 15625 nrhours(C) = 125 intelligence(S) = 800
    satisfaction(S,C) = 19201 difficulty(C) = 125 ability(P) = 125
    friend(S,S1) = 640000 takes(S,C) = 100000 grade(S,C) = 19201
---------------DOMAIN SIZES-------------------------------------
/home/irma/workspace/HybridRDN/UniversityExample/University_domain_200X75x75//test//interp1.pl
domain size:
prof --> 125 objects course --> 125 objects student --> 800 objects
*******************************
LearnedDependencyStatistics [atom=grade(S,C),
 Learned dep: NR:2 grade(S,C) | [1 Value nrhours(C) ]_[2 Value
     intelligence(S) ]_,
 WPLL_test=-1.4755950893284502,
 WPLL_validation=-1.5095959413473214,
 nr_test_instances=19201,
 nr_training_instances=2791,
 nr_validation_instances=2670,
 time_needed_to_learn=219.40355641899998 seconds]

 Inference time per atom:
grade(S,C)=1.506100842 seconds
LEARNED: 1  2
```

Here the user can read what are the learned features in the line "Learned dep:". This example lists two features with their indices in the feature space.

The second file gives a more detailed output. First it lists all the features in the feature space, together with time needed to calculate the values of the feature in data. This gives a nice overview of the feature grounding calculation for a specific domain size and feature length. In the end of the file we list five best dependencies (for an overview). And finally, we output the parameters of the model. The example of the detailed output file for `grade(S,C)` predicate *structure_learning_Log_**grade*** is the following:
(for brevity we will just show the parameters)

```
  ...
Learned dep: Score(grade(S,C) | [1 Value unsorted: nrhours(C) ]_[2 Value
    unsorted: intelligence(S) ]_) = -4030.6211633973485
LEARNED: 1 2
Parameters:
LOGISTIC REGRESSION
med ---> -0.3460976661172188+nrhours(C)
    *0.025422171104744467 2+intelligence(S) *-0.015125489620296287
low ---> -1.0049605912150188+ nrhours(C)
    *0.048619287504200422+intelligence(S) *-0.03276529239707377
high ---> Nth parameter
```

Note that for "high" value we don't have parameters. This is because we can calculate the probability of `grade(S,C)=high` based on probabilities for values "med" and "low" or:

$$P(grade(S,C) = high) = 1 - P(grade(S,C) = low) - P(grade(S,C) = med). \quad (2)$$