kLog
User's Manual

Paolo Frasconi[1,2], Fabrizio Costa[2], Luc De Raedt[2], and Kurt De Grave[2]

[1]Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, via di Santa Marta 3, I-50139 Firenze, Italy
[2]Departement Computerwetenschappen, Katholieke Universiteit Leuven, Celestijnenlaan 200A B-3001 Heverlee, Belgium

June 15, 2013

Contents

1 Introduction

This document provides basic information for kLog, a logical and relational language for developing kernel-based learning systems. kLog has been developed at K.U. Leuven and at UniversitÃ degli Studi di Firenze by P. Frasconi, F. Costa, L. De Raedt and K. De Grave.

License?

2 Tutorial

In this tutorial we explain by examples how to use kLog in several relational learning data sets. You are encouraged to scan the whole tutorial since most concepts are only explained the first time they are used.

kLog learns from interpretations, i.e. training data consist of sets of ground atoms that are true in a given interpretation. An alternative view is that each interpretation is an instance of a relational database.

2.1 Bursi

This is a small molecules data set and the goal is the discrimination between mutagenic and non-mutagenic compounds. Every molecule corresponds to one interpretation so the kLog job is binary classification of interpretations.

kLog data sets

Data sets in kLog are stored as Prolog files consisting of collections of ground facts. The predefined predicate `interpretation/2` takes as its first argument an interpretation identifier and as its second argument a ground term.

In the example below `a,2/`describes atoms, `b,3/`describes chemical bonds, `sub,3/f`unctional groups, `subat,3/`atom membership in functional groups, `target,1/`describes the category of the molecule (`mutagen` is the positive class, `nonmutagen` the negative class), and `linked,2/`describes how functional groups are linked together.

```
interpretation(i788,a(a1,c)).
interpretation(i788,a(a2,c)).
interpretation(i788,a(a3,cl)).
interpretation(i788,a(a4,br)).
interpretation(i788,a(a5,n)).
interpretation(i788,a(a6,h)).
interpretation(i788,b(a1,a2,1)).
interpretation(i788,b(a1,a3,1)).
interpretation(i788,b(a1,a4,1)).
interpretation(i788,b(a2,a5,3)).
interpretation(i788,b(a1,a6,1)).
interpretation(i788,sub(f1,halide,1)).
interpretation(i788,sub(f2,halide,1)).
interpretation(i788,sub(f3,nitrile,2)).
interpretation(i788,sub(f4,aliphatic_chain,1)).
interpretation(i788,subat(f1,a3,1)).
interpretation(i788,subat(f2,a4,1)).
interpretation(i788,subat(f3,a2,1)).
interpretation(i788,subat(f3,a5,2)).
interpretation(i788,subat(f4,a1,1)).
interpretation(i788,target(mutagen)).
interpretation(i788,linked(f4,[link(f3,a2,a1),link(f1,a3,a1),link(f2,a4,a1)],[branch(1,3)],saturated)).
```

```
begin_domain.
signature atm(atom_id::self, element::property)::intensional.
atm(Atom, Element) :-
    a(Atom,Element), \+(Element=h).

signature bnd(atom_1@b::atm, atom_1@b::atm, type::property)::intensional.
bnd(Atom1,Atom2,Type) :-
    b(Atom1,Atom2,NType), describeBondType(NType,Type), atm(Atom1,_), atm(Atom2,_).

signature fgroup(fgroup_id::self, group_type::property)::intensional.
fgroup(Fg,Type) :- sub(Fg,Type,_).

signature fgmember(fg::fgroup, atom::atm)::intensional.
fgmember(Fg,Atom):- subat(Fg,Atom,_), atm(Atom,_).

signature fg_fused(fg1@nil::fgroup, fg2@nil::fgroup, nrAtoms::property)::intensional.
fg_fused(Fg1,Fg2,NrAtoms):- fus(Fg1,Fg2,NrAtoms,_AtomList).

signature fg_connected(fg1@nil::fgroup, fg2@nil::fgroup, bondType::property)::intensional.
fg_connected(Fg1,Fg2,BondType):-
    con(Fg1,Fg2,Type,_AtomList),describeBondType(Type,BondType).

signature fg_linked(fg::fgroup, alichain::fgroup, saturation::property)::intensional.
fg_linked(FG,AliChain,Sat) :-
    linked(AliChain,Links,_BranchesEnds,Saturation),
    (Saturation = saturated ->
     Sat = saturated
    ;
     Sat = unsaturated
    ),
    member(link(FG,_A1,_A2),Links).

signature mutagenic::intensional.
mutagenic :- target(mutagen).
end_domain.
```

2.2 Biodeg

2.3 UW-CSE

2.4 WebKB

2.5 IMDB

Main components of the kLog Prolog library

3.1 syntax.pl: kLog syntax

author Paolo Frasconi

To be done Safety checks, assumption checks, performance tuning

This module provides syntactic extensions to Prolog for declaring signatures. Most of the functionalities in this module are implemented via `term_expansion/2`.

History see git log

+signature +S

Declare S to be the type signature of a table, according to the following syntax:

```
<signature> ::= <header> [<sig_clauses>]
<sig_clauses> ::= <sig_clause> | [<sig_clause> <sig_clauses>]
<sig_clause> ::= <Prolog_clause>
<header> ::= <sig_name> "(" <args> ")" "::" <level> "."
<sig_name> ::= <Prolog_atom>
<args> ::= <arg> | [<arg> <args>]
<arg> ::= <column_name> [<role_overrider>] "::" <type>
<column_name> ::= <Prolog_atom>
<role_overrider> ::= "@" <role>
<role> ::= <Prolog_atom>
<type> ::= "self" | <sig_name>
<level> ::= "intensional" | "extensional"
```

Arguments are given by default a role that corresponds to their position in the argument list. The role can be overridden using the @ syntax, e.g. to represent undirected relations. By default, the role of the i-th argument is: * functor+i if the argument is an identifier * functor+t if the argument is a property and t its type

thus, e.g. if a_id and b_id are identifier names, and c_type is a property name, the following signatures are equivalent:

```
signature foo(g_id::gnus,t_id::tnus,c::property)::extensional.
signature foo(g_id@1,t_id@2,c@3::property)::extensional.
```

Forcing the role can also easily implement undirected edges. Here is an example from mutagenesis:

```
signature atm( atom_id::self,
               element::property,
               quantaval::property,
               chargeval::property).
signature bond( atom_id@b::atom,
                atom_id@b::atom,
                bondtype::property).
```

More complex cases can be easily conceived, e.g. one can assign the same role to every atom in a benzene ring but distinguish different rings in a functional group like ball3 or phenanthrene. See experiments/mutagenesis for details.

+expand_signature(+S, +Level) [*det,private*]

Auxiliary predicate for expanding a signature declaration where S is a fact with typed arguments and Level is either 'intensional' or 'extensional'. This predicate performs a number of checks and asserts various signature traits (see below). `term_expansion/2`.

+domain_traits(?TraitsSpec, ?TraitsVal) [*det*]

Query domain traits. TraitsSpec is one of:

- signatures: list of all signature names
- entities: list of all entity signature names
- relationships: list of all relationship signature names

+domain_traits [*det*]

List on the standard output all the domain traits and signature traits for all signatures.

+domain_traits(+Stream) [*det*]

List on output Stream all the domain traits and signature traits for all signatures.

+signature_traits(?Name, ?TraitsSpec, ?TraitsVal) [*det*]

Query signature traits. Name is the signature name. TraitsSpec is one of:

- kind: either entity or relationship
- arity: the number of arguments
- level: either intensional (deduced) or extensional
- column_types: list of argument types
- column_names: list of argument names
- column_roles: list of argument roles
- relational_arity: the number of non-property arguments

- ext_clause: goal for finding all ground tuples for this signature The value is returned in TraitsVal.

+id_position(+S, -Position) [*semidet*]
: If S is the name of a valid entity signature, unify Position with the position of the identifier in the argument list. Otherwise fail.

+extract_properties(+Functor, +Args, -Properties) [*semidet*]
: If Functor is the name of a valid signature whose arity matches the length of the list Args, then unify Properties with the list of items in Args that appear at positions of property type. Otherwise fail.

+extract_identifiers(+Functor, +Args, -IDs) [*semidet*]
: If Functor is the name of a valid signature whose arity matches the length of the list Args, then unify IDs with the list of items in Args that appear at positions of identifier type. Otherwise fail.

+extract_references(+S, -Refs) [*semidet*]
: If s is the name of a valid signature, then unify Refs with the subsequence of entity types referenced by S but >self< is replaced by S. If a type is referred to multiple times, it appears multiple time in Refs.

+is_kernel_point(+Functor, -YesNo) [*semidet*]
: If Functor is the name of a valid signature then unify YesNo with 'yes' iff the signature has declared a neighborhood method. All graph vertices expanded from this signature will be used as center points for the kernel calculation.

3.2 db.pl: kLog database interface

author Paolo Frasconi

To be done Maybe an interface to a DBMS such as MySQL

This module provides access to the extensional database. Currently it just consults a Prolog database ensuring that facts are loaded into the db namespace.

Syntactical conventions The extensional declarations should use the special predicate `db:interpretation/2` whose first argument is an identifier for the interpration (any Prolog term) and whose second argument is a fact.

History see git log

+db_consult(+File_s:atom_or_list) [*det*]
: Consult a collection of ground facts from File(s). Also accepts basenames of .pl.gz gzipped files.

3.3 graphicalize.pl: Convert deductive databases into graphs

author Paolo Frasconi

To be done Safety checks, assumption checks, throwing exceptions, performance tuning

This module contains predicates for converting a set of deductive databases into a set of graphs, one for each interpretation, using the mapping procedures described below.

Concept First the database is converted into a collection of ground facts by computing intensional tables. Then the graphicalizer generates a graph for each interpretation.

Database The database is assumed to be in the following 'E/R' normal form:

- Attributes are either object identifiers or properties.
- For each identifier i, there is at least one table t such that i is the primary key of t. This table is basically the class or entity-set of i.
- For each table, the primary key only consists of identifiers. So multiway relationships are always represented by a table keyed by the object identifiers involved in the relationship, augmented with properties of the relationship.

Graphicalization strategy

- There is a vertex for each entity tuple (called an i-vertex), and a vertex for each relationship tuple (an r-vertex). Vertices are labeled by their tuples. Note that because of the above assumptions there is one and only one i-vertex for each identifier.
- There is an edge between an i-vertex and an r-vertex if i belongs to the tuple r. The graph is bipartite and there are no edges between vertices of the same kind.

Usage The module provides two fundamental predicates: `attach/1`, for loading a data set (variant `attach/0` for toy datasets asserted directly in the kLog script), and `make_graphs/0`, `make_graphs/1` for generating graphs.

History git log

+attach(+DataFile:atom_or_list_of_atoms) [*det*]

Load a dataset i.e. a set of database instances (or interpretations) by reconsulting the given file(s). Any previously attached data set will be internally cleaned up so only one data set can be attached at a time. Internally the predicate just consults the file containing declarations of the predicate `db:interpretation/2` (a collection of ground facts). The predicate is stored in the 'graphicalize' module and should be never necessary to inspect it from outside this module.

+attach [*det*]

This first form is useful to attach a toy dataset written directly in the kLog script. The data need to be declared as

```
db:interpretation(Ex,Fact).
```

+detach [*det*]
Retracts all data ground atoms and results of graphicalization.

+examples(-Set:list_of_atoms) [*det*]
Returns in Set the interpretation identifiers in the attached domain.

+database [*det*]
For debugging purposes. List the extensional database.

+database(?Ex) [*det*]
For debugging purposes. List the extensional database of interpretation Ex.

+assert_background_knowledge(+Dataset:list_of_atoms) [*det*]
Deduce intensional facts from the background knowledge and interpretations listed in Dataset and assert all of them (at the top) in the form db:interpretation(Ex,Fact). Once asserted in this way, intensional and extensional groundings are indistinguishable. Use the first form to assert background knowledge for every interpretation in the domain.

+save_view(+Filename) [*det,private*]
Save a view of the data using only declared signatures. This allows to write kLog scripts for data set transformation.

+make_graphs [*det*]
Graphicalize the attached interpretations. Graphs are stored in memory.

+vertex_map(Ex:atom, SymID:atom, NumID:integer) [*det,private*]
Dynamically asserted. Dictionary mapping a symbolic vertex identifier (used in Prolog) to its numeric identifier in the internal C++ representation of the graph. The symbolic identifier is defined as follows. If v is an i-vertex, then SymID is the tuple identifier. If v is an r-vertex, then SymID is the concatenation of the signature name and the identifiers contained in it.

+add_graphs(+D:list_of_atoms) [*det,private*]
For each interpretation id in D, create a new empty C++ Graph object

+add_entity_vertices [*det,private*]
Loop through active E-relations and add the corresponding vertices to the internal collection of graphs (this is step 1 of graphicalization).

+add_relationship_vertices_and_edges [*det,private*]
Loop through active R-relations and add the corresponding vertices to the internal collection of graphs. Additionally, add edges to the collection of graphs (this is step 2 of graphicalization).

+map_prolog_ids_to_vertex_ids(+Ex:atom, +IDs:[atom], -NumericIds:[integer])
Retrieve numeric vertex IDs for a list of database identifiers IDs in interpretation Ex

+domain_identifiers(?Ex, ?Type, ?Constants) [*det*]
True if Constants are the constants of type Type in interpretation Ex.

+search_identifiers [*det,private*]

search identifiers from the ground entity sets. Results are asserted in the database as `domain_identifiers/3`.

+check_db [*det*]

Internal use. Performs some checks on the loaded data sets.

3.4 kfold.pl: k-fold cross-validation

author Kurt De Grave, Paolo Frasconi

This module contains predicates for estimating prediction accuracy using k-fold cross validation or a random train/test split. A “fold” in this context refers to a set of interpretations, several cases can be included in a given interpretation. Common testing strategies like leave-one-university-out in WebKB or leave-one-research-group-out in UW-CSE are naturally obtained by setting k to the number of interpretations in the domain. The module contains support for stratified k-fold (where strata are specified by a user-defined predicate) and prepartitioned k-fold where folds are read from file.

It is recommended that any empirical evaluation of kLog is run using predicates in this module since they all save experimental results on disk in a structured form (see below).

Usage See predicates `kfold/4`, `random_split/4`, `fixed_split/6`, `prepartitioned_kfold/4`, `stratified_kfold/5`.

Results Results are saved in a structured way in the file system. `kfold/4` and `random_split/4` create a results directory named after a SHA-1 hash of: (1) current klog flags, (2) domain traits, (3) train/test specification, e.g.

```
PWD/results/e789ba9abfa5e1ae77ec0f481dab9971f343df35
```

where PWD is the current working directory. This ensures that different experiments run with different parameters (such as kernel hyperparameters, regularization, etc) or background knowledge will be kept separate and can coexist for subsequent analysis (e.g. model selection).

For k-fold cross-valitation, rhe results directory is structured into k sub-directories, one for each fold. These contain the following files:

```
auc.log
hash_key.log       text used to generate the hash key
output.log         predicted margins for each case in the form target margin # case term
output.yyy         same after stripping the case term
output.yyy.pr      recall-precision curve
output.yyy.roc     ROC curve
output.yyy.spr     precision points calculated at 100 recall points between 0 and 1.
test.txt           list of test interpretations in this fold
train.txt          list of training interpretations in this fold
```

@tbd Prediction on attributes (multiclass, regression)

@credit AUCCalculator by Jesse Davis and Mark Goadrich

+stratified_kfold(+Signature, +K, +Models, +FeatureGenerator, +StratumFunctor) [*det*]
Perform a stratified k-fold cross validation. The target Signature defines the learning task(s). K is the numnber of folds. Models is a (list of) model(s) to be trained and FeatureGenerator defines the used kernel. StratumFunctor is the name of a user-declared Prolog predicate of the form stratum(+Interpretation,-Stratum); it should unify Stratum (e.g. the category for classification) with the stratum of the given Interpretation.

+prepartitioned_kfold(+Signature, +Models, +FeatureGenerator, +DefinitionFile) [*det*]
Perform a repeated kfold cross-validation reading folds from a file containing facts of the form test_int(Trial,Fold,IntId). Each trial is saved into a separate subfolder (which in turns contains as many folders as folds). This is useful for comparing against other methods while keeping the same partitions, so that paired statistical tests make sense.

+random_split(+Sig:atom, +Fraction:number, +Models:{atom}, +FGenerator:atom) [*det*]
Train and test Model(s) on a given Fraction of existing cases (rest used for test). TrainFraction should be either a fraction in (0.0,1.0) as a float or in (0,100) as an integer.

+fixed_split(+Sig:atom, +Comment:atom, +Train:[atom], +Test:[list], +Models:{atom}, +FGenerator:atom) [*det*]
Do train/test on a given fixed split specified by Train and Test. Comment can be used to describe the split briefly (will go into the results folder name).

+kfold(+Signature, +K, +Models, +FeatureGenerator) [*det*]
Perform a k-fold cross validation. Identical to `stratified_kfold/5` except no stratum predicate is used.

+make_results_directory(-Dir:atom) [*private*]
Creates a directory for results based on the SHA-1 code of current klog flags, e.g. results/e789ba9abfa5e1ae77ec0f481dab9971f343df35, unify Dir with the created directory, and put a log of flags in Dir/flags.log

3.5 learn.pl: Learning support in kLog

author Paolo Frasconi

Formulate kLog learning jobs. Below are some examples of target relations that kLog may be asked to learn. Currently, implemented models have abilities to cover only a fraction of these.

3.5.1 Jobs with relational arity = 0:

Binary classification of each interpretation.

Example: classification of small molecules.

```
signature signature mutagenic::extensional.
```

Regression for each interpretation.

Example: predict the affinity of small molecules binding.

```
signature affinity(strength::property)::extensional.
```

Multitask on individual interpretations.

Example: predict mutagenicity level and logP for small molecules.

```
signature
  molecule_properties(mutagenicity::property(real),
                     logp::property(real))::extensional.
```

Multiclass classification for each interpretation

Example: image categorization;

```
signature image_category(cat::property)::extensional.
```

3.5.2 Jobs with relational arity = 1:

Binary classification of entities in each interpretation

Examples: detect spam webpages as in the Web Spam Challenge (http://webspam.lip6.fr/wiki/pmwiki.php), predict blockbuster movies in IMDb

```
signature spam(url::page)::extensional.
signature blockbuster(m::movie)::extensional.
```

Multiclass classification of entities in each interpretation

Examples: WebKB, POS-Tagging, NER, protein secondary structure prediction

```
signature page(url::page,category::property)::extensional.
signature pos_tag(word::position,tag::property)::extensional.
signature named_entity(word::position,ne::property)::extensional.
signature secondary_structure(r::residue,ss::property)::extensional.
```

Multiclass classification of entities in each interpretation

Example: traffic flow forecasting

```
signature flow_value(s::station,flow::property(real))::extensional.
```

3.5.3 Jobs with relational arity = 2:

Link prediction tasks

Examples: protein beta partners, UW-CSE, Entity resolution, Protein-protein interactions

```
signature partners(r1::residue,r2::residue)::extensional.
signature advised_by(p1::person,p2::person)::extensional.
signature same_venue(v1::venue,v2::venue)::extensional.
signature phosphorylates(p1::kinase,p2::protein)::extensional.
signature regulates(g1::gene,g2::gene)::extensional.
```

Regression on pairs of entities

Example: traffic flow forecasting at different stations and different lead times, Prediction of distance between protein secondary structure elements

```
signature congestion(s::station,lead::time,
                     flow::property(float))::extensional.
signature distance(sse1:sse,sse2:sse,d::property(float))::intensional.
```

Pairwise hierarchical classification

Example: traffic congestion level forecasting at different stations and different lead times

```
signature congestion_level(s::station,lead::time,
                           level::property)::extensional.
```

3.5.4 Tasks with relational arity > 2:

Prediction of hyperedges

Example: Spatial role labeling

```
signature ttarget(w1::sp_ind_can, w2::trajector_can,
                  w3::landmark_can)::intensional.
```

Example: Metal binding geometry

```
signature binding_site(r1::residue,r2::residue,
                       r3::residue,r4::residue)::extensional.
```

Classification of hyperedges

Example: Metal binding geometry with ligand prediction

```
signature binding_site(r1::residue,r2::residue,
                       r3::residue,r4::residue,
                       metal::property)::extensional.
```

3.5.5 Subsampling

The following predefined dynamic predicate fails by default:

```
  user:klog_reject(+Case,+TrainOrPredict)
```

By defining your own version of it, it is easy to implement selective subsampling of cases. A typical usage would be for dealing with a highly imbalanced data set where you want to subsample only a fraction of negatives. Case is a Prolog callable predicate associated with a training or testing case. When kLog generates cases, it calls `user:klog_reject/2` to determine if the case should be rejected. TrainOrPredict should be either 'train' or 'predict' to do subsampling either at training or testing time. For example suppose we have a binary classification task and we want to reject 90 percent of the negatives at training time. Then the following code should be added to the kLog script.

```
klog_reject(Case,train) :-
  \+ call(Case),
  random(R),
  R>0.1.
```

3.5.6 Predicates

+depends_transitive(?S1:atom, ?S:atom) [*det,private*]

Tabled. Contains dependency analysis results to properly kill vertices in the graphs. The predicate succeeds if S1 "depends" on S. The mechanism actually goes a bit beyond a simple transitive closure of the call graph: when several targets are present in the same file it is necessary to extend the definition of dependencies. The set of dependent signatures is defined as follows:

1. All ancestors, including S itself (obvious)

2. All the descendants (less obvious; however, if a target signature S is intensional and calls some predicate Y in its definition, then Y supposedly (although not necessarily) contains some supervision information, which should be removed).
3. All the ancestors of the descendants (maybe even less obvious, but if now there is some other signature T which calls Y, then T will also contain supervision information).

   The descendants of the ancestors need not to be removed.

   Some of the ancestors of S might be legitimate predicates that have nothing to do with supervision. Therefore we restrict ourselves to the subset of the call graph with predicates that are either in the user: namespace (where the 'user' could cheat accessing supervision information) or in the data set file (in the db: namespace). In any case, since removal of vertices from the graph might surprise the user, kLog issues a warning if there are killed signatures besides the target signature of the current training or test procedure. Finally, one can declare a signature to be safe using `safe/1`. In this case it is assumed that the Prolog code inside it does not exploit any supervision information. `safe/1` should be used with care.

   The table depends on `current_depends_directly/2` which is asserted by `record_dependencies/0`, called at the end of `graphicalize:attach/1`.

+train(+S:atom, +Examples:list_of_atoms, +Model:atom, +Feature_Generator:atom) [*det*]

Train Model on a data set of interpretations, using Feature_Generator as the feature generator. Examples is a list of interpretation identifiers. S is a signature. For each possible combination of identifiers in S, a set of "cases" is generated where a case is just a pseudo-iid example. If S has no properties, positive cases are those for which a tuple exist in the interpretation and negative cases all the rest. If S has one property, this property acts as a label for the case (can be also regression if the property is a real number). Two or more properties define a multi-task problem (currently handled as a set of independent tasks). Problems like small molecules classification have a target signature with zero arity which works fine as a special case of relationship. Model should have the ability of solving the task specified in the target signature S.

Errors
- if the target signature is a kernel point (training on it would be cheating)
- if Model cannot solve the task specified by S.

To be done Multitask taking correltations into account

+kill_present(+TargetSignature:atom, +Examples:list) [*det,private*]

Predicates `kill_present/2` and `kill_future/1` are needed to setup training and test data. Let's call vertices associated with the target signature plus all their dependents (defined by `depends_transitive/2`) the set of "query" vertices. In the case of unsliced interpretations, all query vertices must be killed. The case of sliced interpretations is more tricky and is handled as follows:

Example using IMdB, slice_preceq defined by time (years)

Suppose data set contains imdb(1953),...,imdb(1997) and that we want to train on [imdb(1992),imdb(1993)] and test on [imdb(1995),imdb(1996)]. Then during training we first take the most recent year in the training set (1993) and kill **everything** strictly in the future (i.e. 1994, 1995, 1996, 1997) using `kill_future/1` plus the query vertices in the present (i.e. 1992 and 1993) using `kill_present/2`. Test is similar, we take the most recent year in the test set (1996) and kill **everything** strictly in the future (i.e. 1997) plus the query vertices in the present (i.e. 1995 and 1996). Thus, for example, movies of 1994 are not used for training but during prediction their labels are (rightfully) accessible for computing feature vectors. In a hypothetical transductive setting we might keep alive vertices in the future for "evidence" signatures. However this is not currently supported.

`kill_present/2` kills vertices associated with the TargetSignature (and dependents) in the present slices. If interpretations are not sliced vertices are killed anyway.

+kill_future(+Examples:list) [*det,private*]
Kill entire slices in the strict future (if there are no sliced interpretations `kill_future/1` does nothing since max_slice will fail)

+list_of_slices(+SlicedInterpretations, ?Slices) [*det,private*]
Slices is unified with the list of slices found in SlicedInterpretations.

+preceq_max_list(+List, ?M) [*det,private*]
M is unified with the max element in List according to the total order defined by `db:slice_preceq/2`.

+preceq_min_list(+List, ?M) [*det,private*]
M is unified with the min element in List according to the total order defined by `db:slice_preceq/2`.

+predict(+S:atom, +Examples:list_of_atoms, +Model:atom, +Feature_Generator:atom) [*det*]
Test Model on a data set of interpretations. Use Feature_Generator as the feature generator. Examples is a list of interpretation identifiers. S is a signature. Cases are generated as in `train/3`. The predicate asserts induced facts as follows:

```
induced(InterpretationId,db:interpretation(InterpretationID,Fact)).
```

+get_task(+TargetSignature:atom, -TaskIndex:integer, -TaskName:atom, -Values:list) [*nondet,private*]
Given TargetSignature, retrieve the i-th task (starting from 0), unify TaskIndex with i, TaskName with its name and Values with the list of target values found in the data set for this task. TaskName is either in the form N#P where N is the property name and P the position in the argument list, or the atom 'callme' meaning that the task is to learn a relationship.

+cases_loop(+TS, +Examples, +Model, +FG, +TName, +TType, +TOrP) [*det,private*]
Core procedure for training (if TOrP=train) or testing (if TOrP=predict). TS is the target signature. Examples is a list of interpretations (from which cases are built). Model is a kLog model. FG is a kLog feature generator. TName and TType are the task name and type as returned by `get_task/4`. The loop generates all cases for each interpretation (using `tuple_of_identifiers/3`) and creates (if necessary) all required feature vectors and output labels. In prediction mode, cases are immediately predicted. In training mode, C++-level predicates `train_model/2` and `test_dataset/3` are called at the end of the loop. In both cases, results are accumulated both in the local and the global reporters. However the local reporter is reset when this loop starts. This is useful for obtaining training set accuracy and test set accuracy of individual folds in k-fold-CV.

+tuple_of_identifiers(+Ex, +S, -List) [*nondet,private*]
Unify IDList with a tuple of identifiers in Ex whose type appears in signature S. On backtracking will retrieve all possible tuples. For tasks such as link prediction, this predicate is used to generate all pairs of candidates.

+identifier(+Ex, +S, -ID) [*nondet,private*]
Unify ID with one of the identifiers of S in Ex. On backtracking will return all data identifiers for signature S in interpretation Ex. The predicate fails if S is not an entity.

+clean_internals(IntId) [*private*]

+prolog_make_sparse_vector(Model, Feature_Generator, Ex, CaseID, ViewPoint) [*det,private*]
Wrapper around C++ method for making feature vectors. Feature_Generator is the name of the feature generator object that will be used. Ex is the interpretation name, possibly sliced. CaseID identifies the case for which the feature vector is generated. ViewPoint is a list of vertex IDs (C++ integer code) around which the feature vector is constructed. For unsliced interpretations, the feature vector is registered under a slash-separated string like ai/advised_by/person20/person240, created from the interpretation identifier (e.g. ai) followed by the target signature (e.g. advised_by) and the identifiers of the entities that define the case, (e.g. person20 and person412). For sliced interpretations the slice name is also used to construct the identifiers, e.g. imdb_1997 and imdb_1997/m441332. Model is used to determine the internal format of the feature vector being generated.

+make_case_id(Ex, S, IDTuple, CaseID) [*det,private*]
Unifies CaseID with a unique identifier for (sliced) interpretation Ex, target signature S, and tuple of identifiers IDTuple. See `prolog_make_sparse_vector/5` for details on how this is formatted.

+save_induced_facts(Signatures, Filename) [*det*]
Every 'induced' fact (asserted during test) is saved to Filename for a rough implementation of iterative relabeling. The target signature is renamed by prefixing it with 'pred_'.

3.6 flags.pl: flags

author Paolo Frasconi

Module for declaring and manipulating klog flags. For a list of supported flags use `klog_flags/2`.

Developer note To add more flags, use `flag_traits/4` as

```
flag_traits(FlagName,C_Or_Prolog,Checker,Description)
```

where FlagName is an atom naming the flag, C_Or_Prolog is either the atom c or the atom prolog, Checker is a predicate that checks whether a value for this flag is legal, and Description is an atom describing the flag and its legal values. Prolog flags are directly handled in Prolog. C flags are used by the underlying kernel calculation system in C++. The declaration mechanism and default value assignment is slightly different in the two cases.

+klog_flag(+FlagName:atom, ?Val:atom) [*det*]
If Val is a free variable, unify it with the current value of the flag FlagName. Otherwise sets FlagName to Val.

+klog_flag(+Who:atom, +FlagName:atom, ?Val:atom) [*semidet*]
If Val is a free variable, unify it with the current value of the flag FlagName. Otherwise sets FlagName to Val. Who is the handle of the kLog object for which the flag is read or set.

+set_klog_flag(+FlagName:atom, +Val:atom) [*det*]
Sets FlagName to Val.

+set_klog_flag(+Who:atom, +FlagName:atom, +Val:atom) [*det*]
Sets FlagName to Val for C++ object Who.

+get_klog_flag(+FlagName:atom, -Val) [*semidet*]
Unify Val with the current value of the flag FlagName.

+get_klog_flag(+Who:atom, +FlagName:atom, -Val:atom) [*det*]
Unify Val with the current value of the flag FlagName of C++ object Who.

+klog_flags(+Stream) [*det*]
Write a description of current flags to Stream.

+klog_flags [*det*]
Write a description of current flags to current output stream.

3.7 repair.pl: Referential integrity constraints

See also `klog_flag` `referential_integrity_repair`

A referential integrity constraint violation occurs if an R-tuple refers to an undefined identifier (i.e. there is no E-tuple identified by this foreing key). Violations can be repaired or ignored.

+check_and_repair(+Ex:atom) [*det*]
Scan the database for possible referential integrity violations and repair them. The repair strategy is defined by the flag `referential_integrity_repair`. If the value is add, then the defective E-tuple is added. If the value is delete then the offending R-tuple is deleted. If the value is ignore an exception is raised: simply don't call this predicate if no check should be performed.

4 Advanced components

4.1 timing.pl

4.2 goals.pl: goals

author Paolo Frasconi, taking inspiration from various places

Module for manipulating sequences of goals like lists.

+rmember(?A, ?B) [*nondet*]
True when B is a comma-separated sequence and A occurs in it. Similar to `member/2` for lists.

+rabsent(+A, +B) [*det*]
True when B is a comma-separated sequence and A does not occur in it.

+rappend(+A, +B, +C) [*det*]
True if A, B, and C are comma-separated sequences and C is the concatenation of A and B. Useful to generate "negative" edges Warning: does not terminate if A is a free variable.

+goals_to_list(+G, -L) [*det*]
True if G is a comma-separated sequence, and L the corresponding list of items. Warning: does not terminate if G is a free variable.

+list_to_goals(+L, -G) [*det*]
True if L is a list of items and G the corresponding comma-separated sequence Warning: does not terminate if L is a free variable.

4.3 utils.pl

+aformat(-Atom, +Format, +List)
similar to sformat but builds an atom instead of a list of char codes.

+new_progress_bar(+BarName:atom, +MaxCount:integer) [*det*]
create a progress bar called BarName. The gauged task is completed when the counter for the bar reaches the integer value MaxCount

+progress_bar(+BarName:atom) [*det*]
increase the counter for the progress bar BarName and maybe display information on screen.

+slice(?L1, +I, +K, ?L2) [*det*]

L2 is the list of the elements of L1 between index I and index K (both included).

+shuffle(+Ex:list, -Ex1:list) [*det*]

Unify Ex1 with a random permutation of list Ex.

4.4 doc/cinterface.pl: C++ interface

General mechanism: functions listed here are the interface to Prolog. All model-related predicates take an atom as first argument that identifies the model (valid atoms should be strings, not integers). The identifier is then used as a key for a dictionary stored in the singleton The_ModelPool with associates the model name to a pointer to an object of Model class (Model is the abstract class from which all Models are inherited). Typically, functions defined here should call a homologous method of the class ModelPool that will dispatch the message to the actual model.

4.4.1 Example

Prolog code in the kLog script:

```
new_model(my_model,libsvm_c_svc),
```

in c_interface, c_new_model() calls the method of the singleton The_ModelPool

```
The_ModelPool.new_model("my_model","libsvm_c_svc")
```

which eventually creates an object of class Libsvm_Model<BinaryClassifierReporter>. now back to the prolog code, the atom my_model can be used as a handle for referring to the C++ object just created, e.g.

```
kfold(target_relation, 10, my_model, my_fg)
```

that causes the invocation of the method

```
The_ModelPool.train("my_model",vector_of_interpretation_ids)
```

A similar mechanism is used for feature generators via the singleton The_FeatureGeneratorPool of class FeatureGeneratorPool.

4.4.2 Data

Data is contained in the singleton The_Dataset of type Dataset. There is a graph for each interpretation and possibly several sparse vectors associated to the scalar predictions (called cases in kLog). The former are accessed from prolog via interpretation identifiers, the latter via "case" identifiers.

The_Dataset maintains internal dictionaries (indices) to associate identifiers to graph pointers or sparse vector pointers.

Additionally, The_Dataset maintains a map from interpretation ids to a set<string> of case ids. Use The_Dataset.get_cases() to retrieve this map.

+set_c_klog_flag(+Client:atom, +Flag:atom, +Value:atom) [*det*]
Sets C-level kLog Flag to Value for Client

+get_c_klog_flag(+Client, +Flag, -Value) [*det*]
Unifies Value with the current value of Flag in Client.

+document_klog_c_flags(+Client:atom, -Documentation:atom) [*det*]
Retrieve Documentation for all flags belonging to Client.

+klog_c_flag_client(?Client) [*nondet*]
Backtrack over the IDs of the C language flag clients.

+klog_c_flag_client_alt(?Client) [*nondet*]
Backtrack over the IDs of the C language flag clients. This implementation uses less memory, it doesn't store a copy of all alternatives in memory

+c_shasum(+Filename, -HashValue) [*det*]
Unifies HashValue with the SHA1 hash value of the file (which must exist).

+add_graph(+Id) [*det*]
Create a new graph identified by Id. Id can be any valid Prolog atom.

+add_graph_if_not_exists(+Id) [*det*]
Create a new graph identified by Id unless it already exists. Id can be any valid Prolog atom

+delete_graph(+Id) [*det*]
Delete graph identified by Id.

+write_graph(+Id) [*det*]
Write graph identified by Id on the standard output.

+export_graph(+Id, +Filename, +Format)
Save graph identified by Id into Filename in specified Format. File extension is automatically created from format (it is actually identical). Valid formats are dot, csv, gml, gdl, gspan.

+save_as_libsvm_file(+Filename) [*det*]
Save a sparse vector dataset into Filename in libsvm format. Data is generated for all interpretations in the attached dataset. Every line is a case. The line is ended by a comment giving the case name, which is formed by concatenating the interpretation name and the identifiers contained in the target fact.

+cleanup_data [*det*]
Clean completely graphs and sparse vectors.

+clean_internals(+GId) [*det*]
Clean up internal data structures on GId to recover memory

+vertex_alive(+GId, +VId, ?State) [*det*]
Set/get aliveness State (yes/no) of vertex VId in graph GId

+set_signature_aliveness(+GId, +S, +State) [*det*]
Set aliveness state to State to every vertex of signature S in graph GId

+set_slice_aliveness(+GId, +Slice, +State) [*det*]
Set aliveness state to Status to every vertex in slice Slice in graph GId

+set_signature_in_slice_aliveness(+GId, +S, +SliceID, +State) [*det*]
Set aliveness state to State to every vertex of signature S in slice SliceID in graph GId

+set_all_aliveness(+GId, +State) [*det*]
Set aliveness state to State in every vertex in graph GId

+add_vertex(+GId, +SliceId, +Label, +IsKernelPoint, +IsAlive, +Kind, +SymId, -Id) [*det*]
Create new vertex in graph GId with label Label and unify Id with the identifier of the new vertex Set KernelPoint to 'yes' to indicate that this vertex is a kernel point Set IsAlive to 'yes' to indicate that this vertex is alive (i.e. actually exists). Dead vertices are used for structured output predictions. Kind should be either 'r' for relationship or 'i' for an entity set. SymId is the symbolic id of the vertex and is only used for visualization/debugging purposes. SliceId is the slice to which this vertex belongs to.

+add_edge(+GId, +V, +U, +Label, -Id) [*det*]
Create new edge between U and V with label Label and unify Id with the identifier of the new edge

+make_sparse_vector(+ModelType:atom, +FG:atom, +IntId:atom, +SliceId, +CaseId:atom, +ViewPoints:list_of_num
Use feature generator FG to make a sparse vector from graph Ex and focusing on the list of vertices in ViewPoints. The sparse vector is then identified by CaseId. ModelType is used to construct an appropriate internal representation of the feature vector.

+write_sparse_vector(+CaseId) [*det*]
Write the sparse vector identified by CaseId.

+add_feature_to_sparse_vector(+CaseId, +FeatureStr, +FeatureVal) [*det*]
Adds new feature to sparse vector identified by CaseId. FeatureStr is hashed to produce the index in the sparse vector. FeatureVal is then assigned to vector[hash(FeatureStr)] This method was introduced for ghost signatures

+set_target_label(+CaseID, +Label) [*det*]
Set the label for SVM CaseID

+set_label_name(+Label:atom, +NumericLabel:number) [*det*]
Record label name

+set_rejected(+CaseID) [*det*]
Set reject flag for SVM CaseID

+clear_rejected [*det*]
Clear all reject flags

+remap_indices [*det*]
Remap feature vector indices in a small range

+print_graph_ids [*det*]
Print all IDs for graphs in the dataset

+new_feature_generator(+FGId, +FGType) [*det*]
Create a new feature generator identified by atom FGId

+delete_feature_generator(+Id) [*det*]
Delete feature_generator identified by Id.

+new_model(+ModelId, +ModelType) [*det*]
Create a new model identified by atom ModelId

+delete_model(+Id) [*det*]
Delete model identified by Id.

+model_type(+ModelId, -ModelType) [*det*]
Unifies ModelType with type of model identified by ModelId.

+write_model(+Id) [*det*]
Write model identified by Id.

+load_model(+Id, +Filename) [*det*]
Load model identified by Id from Filename

+save_model(+Id, +Filename) [*det*]
Save model identified by Id into Filename in kLog internal format

+check_model_ability(+ModelId:atom, +Ability:atom) [*det*]
Ask ModelId whether it can perform the task specified in Ability.

+train_model(+ModelId, +DatasetSpec) [*det*]
Train model identified by ModelIdId. The list DatasetSpec should contain identifiers of interpretations. Case identifiers are then generated in Pool.h

+test_dataset(+ModelId:atom, +DatasetSpec:atom, +NewData:atom) [*det*]
Test model identified by ModelIdId on a dataset. The list DatasetSpec should contain identifiers of interpretations. Case identifiers are then generated in Pool.h. If NewData is "true" (or "yes") then it is assumed that the evaluation is on test data and results are accumulated in the global reporter of the model. Otherwise it is assumed that we are testing on training data and only the local reporter is affected.

+dot_product(+CaseId1, +CaseId2, -Value) [*det*]
Unify Value with the dot product of the feature vectors attached to case identifiers CaseId1 and CaseId2.

+set_model_wd(+ModelId, +WorkingDir) [*det*]
Inform model of the working directory. This is mainly needed for external models in order to keep a clean filesystem.

+reset_reporter(+ModelId, +Reporter) [*det*]
Reset a reporter of the model. Reporter should be either local or global.

+reset_reporter(+ModelId, +Reporter, +Description) [*det*]
Reset a reporter of the model. Reporter should be either local or global. Also sets the description message to Description.

+report(+ModelId, +Reporter) [*det*]
Ask a performance reporter to write a report on the standard output. Reporter should be either local or global.

+report(+ModelId, +Filename, +Reporter) [*det*]
Ask a performance reporter to write a report to Filename. Reporter should be either local or global.

+save_predictions(+ModelId, +Dir, +Reporter) [*det*]
Save predictions stored in a performance reporter in output.log and maybe output.yyy in Dir. Do AUC analysis for binary classification reporters. Reporter should be either local or global.

+get_prediction(+ModelId, +CaseID, -Margin) [*det*]
Obtain prediction for CaseID in the global reporter of model identified by ModelId. WARNING: fails without displaying any message if arguments are not valid.