Python-RDM Documentation

Release 0.1

Anze Vavpetic, Matic Perovsek

Aug 05, 2016

Contents

Contents:

CHAPTER 1

Introduction

This is the documentation for the Python-RDM package for relational data mining. The aim of this tool is to make relational learning and inductive logic programming approaches publicly accessible. The tool offers a common and easy-to-use interface to several relational learning algorithms and provides data access to several relational database management systems. To this end we have developed a stand-alone Python library and a corresponding package for the open source ClowdFlows online data mining platform.

1.1 Help

If you need help please open an issue on GitHub.

CHAPTER 2

Installation

The package was successfully installed on Linux, Windows and OS X systems.

Latest release from PyPI:

```
pip install python-rdm
```

Latest from GitHub:

```
pip install https://github.com/xflows/rdm/archive/master.zip
```

The prerequisites are listed in `requirements.txt` .

2.1 Prerequisites of specific ILP/RDM algorithms

Depending on what algorithms you wish to use, these are their dependencies.

2.1.1 Aleph and RSD

- Yap prolog (preferably compiled with `--tabling` enabled for speedups)

There are sources as well as binaries for Windows and OS X available here.

On Debian-based systems you can simply install it as:

```
apt install yap
```

2.1.2 TreeLiker, Caraf, Cardinalization, Quantiles, Relaggs

- Java

2.1.3 1BC, 1BC2, Tertius

These approaches depend on one original C program which must be compiled. The sources are included with python-rdm in `rdm/wrappers/tertius/src/` .

2.2 Documentation

You'll need Sphinx to build the documentation you are currently looking at:

```
pip install -U Sphinx
```

CHAPTER 3

Getting started

```
from rdm.db import DBVendor, DBConnection, DBContext, AlephConverter
from rdm.wrappers import Aleph

# Provide connection information
connection = DBConnection(
    'ilp',             # User
    'ilp123',          # Password
    'workflow.ijs.si', # Host
    'ilp',             # Database
)

# Define learning context
context = DBContext(connection, target_table='trains', target_att='direction')

# Convert the data and induce features using Aleph
conv = AlephConverter(context, target_att_val='east')
aleph = Aleph()
theory, features = aleph.induce('induce_features', conv.positive_examples(),
                                conv.negative_examples(),
                                conv.background_knowledge())
print theory
```

CHAPTER 4

Example use case

```
import orange

from rdm.db import DBVendor, DBConnection, DBContext, RSDConverter, mapper
from rdm.wrappers import RSD
from rdm.validation import cv_split
from rdm.helpers import arff_to_orange_table

# Provide connection information
connection = DBConnection(
    'ilp',  # User
    'ilp123',  # Password
    'ged.ijs.si',  # Host
    'imdb_top',  # Database
    vendor=DBVendor.MySQL
)

# Define learning context
context = DBContext(connection, target_table='movies', target_att='quality')

# Cross-validation loop
predictions = []
folds = 10
for train_context, test_context in cv_split(context, folds=folds, random_seed=0):
    # Find features on the train set
    conv = RSDConverter(train_context)
    rsd = RSD()
    features, train_arff, _ = rsd.induce(
        conv.background_knowledge(),    # Background knowledge
        examples=conv.all_examples(),   # Training examples
        cn2sd=False                     # Disable built-in subgroup discovery
    )

    # Train the classifier on the *train set*
    train_data = arff_to_orange_table(train_arff)
    tree_classifier = orange.TreeLearner(train_data, max_depth=5)

    # Map the *test set* using the features from the train set
    test_arff = mapper.domain_map(features, 'rsd', train_context, test_context,␣
↪format='arff')

    # Classify the test set
    test_data = arff_to_orange_table(test_arff)
```

```
    fold_predictions = [(ex[-1], tree_classifier(ex)) for ex in test_data]
    predictions.append(fold_predictions)

acc = 0
for fold_predictions in predictions:
    acc += sum([1.0 for actual, predicted in fold_predictions if actual ==␣
↪predicted])/len(fold_predictions)
acc = 100 * acc/folds

print 'Estimated predictive accuracy: {0:.2f}%'.format(acc)
```

CHAPTER 5

ClowdFlows

ClowdFlows is an open source web-based data mining platform. The python-rdm package also includes ClowdFlows widgets, which can be used to easily compose workflows for mining relational databases.

5.1 User Documentation

Here's an example workflow that demonstrates the usage of RDM widgets in Clowdflows. More specifically, the workflow constructs a decision tree on the Michalski Trains dataset (stored in a MySQL database) using Aleph to propositionalize the dataset.

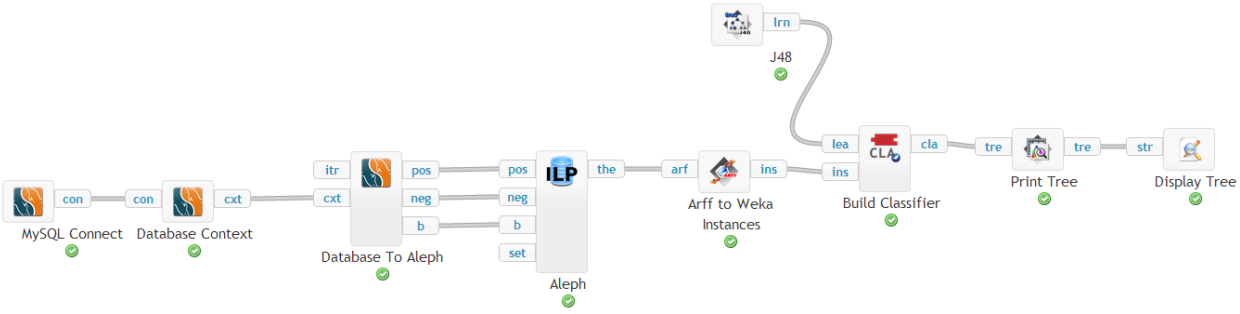

Fig. 5.1: Click the image to open the ClowdFlows workflow.

- Full ClowdFlows User docs
- Public instance of ClowdFlows

5.2 Developer Documentation

You can relatively easily extend your local ClowdFlows installation by developing new widgets. See the Developer documentation on ReadTheDocs.org, for instructions on how to develop and deploy widgets.

You are of course welcome to share your widgets with everyone. To do so, please issue a pull request.

- GitHub repository
- Wiki
- Developer documentation

The `python-rdm` ClowdFlows widgets follow the main ClowdFlows convention; `rdm.db` and `rdm.wrappers` can be imported as ClowdFlows packages and have the following internal structure:

- `rdm/<package_name>` - package root,
- `rdm/<package_name>/package_data` - widget database fixtures,
- `rdm/<package_name>/static` - widget-related static files, e.g., icons,
- `rdm/<package_name>/library.py` - main widget views,
- `rdm/<package_name>/interaction_views.py` - widget views that require a user interaction before doing a computation,
- `rdm/<package_name>/visualization_views.py` - widget views that visualize something after computation.

CHAPTER 6

API Reference

The package is divided into two main independent subpackages: `rdm.db` and `rdm.wrappers`.

6.1 Database interaction

Databases can be accessed via different so-called data sources. You can add your own data source by subclassing the base `rdm.db.datasource.DataSource` class.

6.1.1 Base DataSource

class `rdm.db.datasource.` **`DataSource`**

A data abstraction layer for accessing datasets.

This layer is typically hidden from end-users, as they only access the database through DBConnection and DBContext objects.

`column_values`(*table*, *col*)

Returns a list of distinct values for the given table and column.

param table target table

param cols list of columns to select

`connect`( )

Returns a connection object.

Return type DBConnection

`connected`(*tables*, *cols*, *find_connections=False*)

Returns a list of tuples of connected table pairs.

param tables a list of table names

param cols a list of column names

param find_connections set this to True to detect relationships from column names.

return a tuple (connected, pkeys, fkeys, reverse_fkeys)

`fetch`(*table*, *cols*)

Fetches rows for the given table and columns.

param table target table

param cols list of columns to select

return rows from the given table and columns

rtype list

fetch_types (*table*, *cols*)
Returns a dictionary of field types for the given table and columns.

param table target table

param cols list of columns to select

return a dictionary of types for each attribute

rtype dict

foreign_keys ( )

Returns a list of foreign key relations in the form (table_name, column_name, referenced_table_name, referenced_column_name).

Return type list

select_where (*table*, *cols*, *pk_att*, *pk*)
Select with where clause.

param table target table

param cols list of columns to select

param pk_att attribute for the where clause

param pk the id that the pk_att should match

return rows from the given table and cols, with the condition pk_att==pk

rtype list

table_column_names ( )

Returns a list of table / column names in the form (table, col_name).

Return type list

table_columns (*table_name*)

Parameters **table_name** – table name for which to retrieve column names

Returns a list of columns for the given table.

Return type list

table_primary_key (*table_name*)
Returns the primary key attribute name for the given table.

param table_name table name string

tables ( )

Returns a list of table names.

Return type list

6.1.2 MySQLDataSource

class rdm.db.datasource. **MySQLDataSource** (*connection*)
A DataSource implementation for accessing datasets from a MySQL DBMS.

__init__ (*connection*)

Parameters **connection** – a DBConnection instance.

6.1.3 PgSQLDataSource

class rdm.db.datasource. **PgSQLDataSource** (*connection*)
A DataSource implementation for accessing datasets from a PosgreSQL DBMS.

__init__ (*connection*)

Parameters **connection** – a DBConnection instance.

6.1.4 Database Context

A `DBContext` object represents a *view* of a particular data source that can be used for learning. Example uses include: selecting only particular tables, table columns, a target attribute, and so on.

class rdm.db.context. **DBContext** (*connection*, *target_table=None*, *target_att=None*, *find_connections=False*, *in_memory=True*)

__init__ (*connection*, *target_table=None*, *target_att=None*, *find_connections=False*, *in_memory=True*)
Initializes a new DBContext object from the given DBConnection.

Parameters

- **connection** – a DBConnection instance
- **target_table** – set a target table for learning
- **target_att** – set a target table attribute for learning
- **find_connections** – set to True if you want to detect relationships based on attribute and table names, e.g., `train_id` is the foreign key refering to `id` in table `train`.
- **in_memory** – Load the database into main memory (currently required for most approaches and pre-processing)

copy ( )
Makes a deepcopy of the DBContext object (e.g., for making folds)

returns a deep copy of `self`.

rtype DBContext

fetch (*table*, *cols*)
Fetches rows from the db.

param table table name to select

cols list of columns to select

return list of rows

rtype list

fetch_types (*table*, *cols*)
Returns a dictionary of field types for the given table and columns.

param table target table

param cols list of columns to select

return a dictionary of types for each attribute

rtype dict

rows (*table*, *cols*)
Fetches rows from the local cache or from the db if there's no cache.

param table table name to select

cols list of columns to select

return list of rows

rtype list

select_where (*table*, *cols*, *pk_att*, *pk*)
SELECT with WHERE clause.

param table target table

param cols list of columns to select

param pk_att attribute for the where clause

param pk the id that the pk_att should match

return rows from the given table and cols, with the condition pk_att==pk

rtype list

6.2 Database converters

Converters are used to change the representation of the input database to a native representation of a particular algorithm.

class rdm.db.converters. **Converter** (*dbcontext*)
Base class for converters.

__init__ (*dbcontext*)
Base class for handling converting DBContexts to various relational learning systems.

param dbcontext DBContext object for a learning problem

class rdm.db.converters. **ILPConverter** (**args*, ***kwargs*)
Base class for converting between a given database context (selected tables, columns, etc) to inputs acceptable by a specific ILP system.

param discr_intervals (optional) discretization intervals in the form:

```
>>> {'table1': {'att1': [0.4, 1.0], 'att2': [0.1, 2.0, 4.5]}, 'table2': {
↪'att2': [0.02]}}
```

given these intervals, e.g., `att1` would be discretized into three intervals: `att1 =< 0.4`, `0.4 < att1 =< 1.0`, `att1 >= 1.0`

param settings dictionary of `setting: value` pairs

mode (*predicate*, *args*, *recall=1*, *head=False*)
Emits mode declarations in Aleph-like format.

param predicate predicate name

param args predicate arguments with input/output specification, e.g.:

```
>>> [('+', 'train'), ('-', 'car')]
```

param recall recall setting (see Aleph manual)

param head set to True for head clauses

user_settings ( )
Emits prolog code for algorithm settings, such as `:-set(minpos,5).`.

class rdm.db.converters. **RSDConverter** (**args*, ***kwargs*)
Converts the database context to RSD inputs.

Inherits from ILPConverter.

all_examples (*pred_name=None*)
Emits all examples in prolog form for RSD.

param pred_name override for the emitted predicate name

background_knowledge ( )
Emits the background knowledge in prolog form for RSD.

class rdm.db.converters. **AlephConverter** (**args*, ***kwargs*)
Converts the database context to Aleph inputs.

Inherits from ILPConverter.

__init__ (**args*, ***kwargs*)

Parameters **discr_intervals** – (optional) discretization intervals in the form:

```
>>> {'table1': {'att1': [0.4, 1.0], 'att2': [0.1, 2.0, 4.5]}, 'table2': {'att2
↪': [0.02]}}
```

given these intervals, e.g., `att1` would be discretized into three intervals: `att1 =< 0.4`, `0.4 < att1 =< 1.0`, `att1 >= 1.0`

Parameters

- **settings** – dictionary of `setting: value` pairs
- **target_att_val** – target attribute *value* for learning.

background_knowledge ( )
Emits the background knowledge in prolog form for Aleph.

negative_examples ( )
Emits the negative examples in prolog form for Aleph.

positive_examples ( )
Emits the positive examples in prolog form for Aleph.

class rdm.db.converters. **OrangeConverter** (**args*, ***kwargs*)
Converts the selected tables in the given context to Orange example tables.

convert_table (*table_name*, *cls_att=None*)
Returns the specified table as an orange example table.

param table_name table name to convert

cls_att class attribute name

rtype orange.ExampleTable

orng_type (*table_name*, *col*)
Returns an Orange datatype for a given mysql column.

param table_name target table name

param col column to determine the Orange datatype

other_Orange_tables ( )
Returns the related tables as Orange example tables.

Return type list

target_Orange_table ( )
Returns the target table as an Orange example table.

rtype orange.ExampleTable

class rdm.db.converters. **TreeLikerConverter** (**args*, ***kwargs*)
Converts a db context to the TreeLiker dataset format.

param discr_intervals (optional) discretization intervals in the form:

```
>>> {'table1': {'att1': [0.4, 1.0], 'att2': [0.1, 2.0, 4.5]}, 'table2': {
↪'att2': [0.02]}}
```

given these intervals, e.g., `att1` would be discretized into three intervals: `att1 =< 0.4`, `0.4 < att1 =< 1.0`, `att1 >= 1.0`

dataset ( )
Returns the DBContext as a list of interpretations, i.e., a list of facts true for each example in the format for TreeLiker.

default_template ( )
Default learning template for TreeLiker.

6.3 Algorithm wrappers

The `rdm.wrappers` module provides classes for working with the various algorithm wrappers.

6.3.1 Aleph

This is a wrapper for the very popular ILP algorithm Aleph. Aleph is an ILP toolkit with many modes of functionality: learning theories, feature construction, incremental learning, etc. Aleph uses mode declarations to define the syntactic bias. Input relations are Prolog clauses, defined either extensionally or intensionally.

Official documentation.

See *Getting started* for an example of using Aleph in your python code.

class rdm.wrappers. **Aleph** (*verbosity=0*)
Aleph python wrapper.

__init__ (*verbosity=0*)
Creates an Aleph object.

param logging Can be DEBUG, INFO or NOTSET (default).

This controls the verbosity of the output.

__weakref__
list of weak references to the object (if defined)

induce (*mode*, *pos*, *neg*, *b*, *filestem='default'*, *printOutput=False*)
Induce a theory or features in 'mode'.

param filestem The base name of this experiment.

param mode In which mode to induce rules/features.

param pos String of positive examples.

param neg String of negative examples.

param b String of background knowledge.

return The theory as a string or an arff dataset in induce_features mode.

rtype str

set (*name*, *value*)
Sets the value of setting 'name' to 'value'.

param name Name of the setting

param value Value of the setting

setPostScript (*goal*, *script*)
After learning call the given script using 'goal'.

param goal goal name

param script prolog script to call

settingsAsFacts (*settings*)
Parses a string of settings.

param setting String of settings in the form:

```
set(name1,val1),set(name2,val2)...
```

6.3.2 RSD

RSD is a relational subgroup discovery algorithm (Zelezny et al, 2001) composed of two main steps: the propositionalization step and the (optional) subgroup discovery step. RSD effectively produces an exhaustive list of first-order features that comply with the user-defined mode constraints, similar to those of Progol (Muggleton, 1995) and Aleph.

See *Example use case* for an example of using RSD in your code.

class rdm.wrappers. **RSD** (*verbosity=0*)
RSD python wrapper.

__init__ (*verbosity=0*)
Creates an RSD object.

param logging Can be DEBUG, INFO or NOTSET (default).

This controls the verbosity of the output.

__weakref__
list of weak references to the object (if defined)

induce (*b*, *filestem='default'*, *examples=None*, *pos=None*, *neg=None*, *cn2sd=True*, *printOutput=False*)

Generate features and find subgroups.

param filestem The base name of this experiment.

param examples Classified examples; can be used instead of separate pos / neg files below.

param pos String of positive examples.

param neg String of negative examples.

param b String with background knowledge.

param cn2sd Find subgroups after feature construction?

return a tuple `(features,weka,rules)` , where:

- features is a set of prolog clauses of generated features,
- weka is the propositional form of the input data,
- rules is a set of generated cn2sd subgroup descriptions; this will be an empty string if cn2sd is set to False.

rtype tuple

set (*name*, *value*)

Sets the value of setting 'name' to 'value'.

param name Name of the setting

param value Value of the setting

settingsAsFacts (*settings*)

Parses a string of settings.

param setting String of settings in the form:

```
set(name1,val1),set(name2,val2)...
```

6.3.3 TreeLiker

TreeLiker (by Ondrej Kuzelka et al) is suite of multiple algorithms (controlled by the `algorithm` setting), RelF, Poly and HiFi:

RelF constructs a set of tree-like relational features by combining smaller conjunctive blocks. The novelty is that RelF preserves the monotonicity of feature reducibility and redundancy (instead of the typical monotonicity of frequency), which allows the algorithm to scale far better than other state-of-the-art propositionalization algorithms.

HiFi is a propositionalization approach that constructs first-order features with hierarchical structure. Due to this feature property, the algorithm performs the transformation in polynomial time of the maximum feature length. Furthermore, the resulting features are the smallest in their semantic equivalence class.

Official website

Example usage:

```
>>> context = DBContext(...)
>>> conv = TreeLikerConverter(context)
>>> treeliker = TreeLiker(conv.dataset(), conv.default_template())    # Runs RelF by␣
↪default
>>> arff, _ = treeliker.run()
```

class rdm.wrappers. **TreeLiker** (*dataset*, *template*, *test_dataset=None*, *settings={}*)

TreeLiker python wrapper.

__init__ (*dataset*, *template*, *test_dataset=None*, *settings={}*)

Parameters

- **dataset** – dataset in TreeLiker format
- **template** – feature template
- **test_dataset** – (optional) test dataset to transform with the features from the training set
- **settings** – dictionary of settings (see TreeLiker documentation)

run (*cleanup=True*, *printOutput=False*)

Runs TreeLiker with the given settings.

param cleanup deletes temporary files after completion

param printOutput print algorithm output to the terminal

6.3.4 Wordification

Wordification (Perovsek et al, 2015) is a propositionalization method inspired by text mining that can be viewed as a transformation of a relational database into a corpus of text documents. Wordification constructs simple, easily interpretable features, acting as words in the transformed Bag-Of-Words representation.

Example usage:

```
>>> context = DBContext(...)
>>> orange = OrangeConverter(context)
>>> wordification = Wordification(orange.target_Orange_table(), orange.other_Orange_
↪tables(), context)
>>> wordification.run(1)
>>> wordification.calculate_weights()
>>> arff = wordification.to_arff()
```

class rdm.wrappers. **Wordification** (*target_table*, *other_tables*, *context*, *word_att_length=1*, *idf=None*)

__init__ (*target_table*, *other_tables*, *context*, *word_att_length=1*, *idf=None*)

Wordification object constructor.

param target_table Orange ExampleTable, representing the primary table

param other_tables secondary tables, Orange ExampleTables

__weakref__

list of weak references to the object (if defined)

att_to_s (*att*)

Constructs a "wordification" word for the given attribute

param att Orange attribute

calculate_weights (*measure='tfidf'*)
Counts word frequency and calculates tf-idf values for words in every document.

param measure example weights approach (can be one of `tfidf,binary,tf`).

prune (*minimum_word_frequency_percentage=1*)
Filter out words that occur less than minimum_word_frequency times.

param minimum_word_frequency_percentage minimum frequency of words to keep

run (*num_of_processes=4*)
Applies the wordification methodology on the target table

param num_of_processes number of processes

to_arff ( )
Returns the "wordified" representation in ARFF.

rtype str

wordify ( )
Constructs string of all documents.

return document representation of the dataset, one line per document

rtype str

6.3.5 Proper

class rdm.wrappers. **Proper** (*input_dict*, *is_relaggs*)

__dict__ = dict_proxy({'__module__': 'rdm.wrappers.proper.proper', 'run': <function run>, 'init_args_list': <functio

__init__ (*input_dict*, *is_relaggs*)

__module__ = 'rdm.wrappers.proper.proper'

__weakref__
list of weak references to the object (if defined)

init_args_list (*input_dict*, *is_relaggs*)

parse_excluded_fields (*context*)

run ( )

6.3.6 Tertius

class rdm.wrappers. **Tertius** (*input_dict*)

__dict__ = dict_proxy({'__module__': 'rdm.wrappers.tertius.tertius', 'run': <function run>, 'init_args_list': <functio

__init__ (*input_dict*)

__module__ = 'rdm.wrappers.tertius.tertius'

__weakref__
list of weak references to the object (if defined)

init_args_list (*input_dict*)

run ( )

6.3.7 OneBC

class rdm.wrappers. **OneBC** (*input_dict*, *is1BC2*)

__dict__ = dict_proxy({'__module__': 'rdm.wrappers.tertius.onebc', 'run': <function run>, 'init_args_list': <function

__init__ (*input_dict*, *is1BC2*)

__module__ = 'rdm.wrappers.tertius.onebc'

__weakref__
list of weak references to the object (if defined)

init_args_list (*input_dict*)

run ( )

6.3.8 Caraf

class rdm.wrappers. **Caraf** (*input_dict*)

__dict__ = dict_proxy({'__module__': 'rdm.wrappers.caraf.caraf', 'run': <function run>, '__dict__': <attribute '__di

__init__ (*input_dict*)

__module__ = 'rdm.wrappers.caraf.caraf'

__weakref__
list of weak references to the object (if defined)

run ( )

6.4 Utilities

This section documents helper utilities provided by the python-rdm package that are useful in various scenarios.

6.4.1 Mapping unseen examples into propositional feature space

When testing classifiers (or in a real-world scenario) you'll need to map unseen (or new) examples into the feature space used by the classifier. In order to do this, use the `rdm.db.mapper` function.

See *Example use case* for usage in a cross-validation setting.

rdm.db.mapper. **domain_map** (*features*, *feature_format*, *train_context*, *test_context*, *intervals={}*, *format='arff'*, *positive_class=None*)
Use the features returned by a propositionalization method to map unseen test examples into the new feature space.

param features string of features as returned by rsd, aleph or treeliker

param feature_format 'rsd', 'aleph', 'treeliker'

param train_context DBContext with training examples

param test_context DBContext with test examples

param intervals discretization intervals (optional)

param format output format (only arff is used atm)

param positive_class required for aleph

return returns the test examples in propositional form

rtype str

Example

```
>>> test_arff = mapper.domain_map(features, 'rsd', train_context, test_
↪context)
```

6.4.2 Validation

Python-rdm provides a helper function for splitting a dataset into folds for cross-validation.

See *Example use case* for a cross-validation example using RSD.

rdm.validation. **cv_split** (*context*, *folds=10*, *random_seed=None*)

Returns a list of pairs (train_context, test_context), one for each cross-validation fold.

The split is stratified.

param context DBContext to be split

param folds number of folds

param random_seed random seed to be used

return returns a list of (train_context, test_context) pairs

rtype list

Example

```
>>> for train_context, test_context in cv_split(context, folds=10, random_
↪seed=0):
>>>     pass  # Your CV loop
```

CHAPTER 7

Licences of included approaches

Although python-rdm itself is MIT licensed, we include approaches that have their own licenses (all of the sources are unmodified). To be sure, please contact the respective authors if you want to use their approach for any commercial purposes.

- **Aleph**
  - Official page
  - Freely available for academic purposes, contact the author Ashwin Srinivasan for commercial use
  - The source code is included here (aleph.pl)
- **RSD**
  - by Filip Železný et al
  - Official page
  - The source code is included here (.pl files)
  - Included with permission by the author
- **TreeLiker** (includes HiFi, RelF and Poly)
  - Official page
  - The binaries are included here
  - GPL license
- **Wordification**
  - by Matic Perovšek et al
  - python-rdm is currently the main repository for this approach.
  - The source code is included here
  - MIT license

Nicolas Lachiche's team at the University of Strasbourg contributions:

- **1BC, 1BC2, Tertius**
  - By Peter Flach and Nicolas Lachiche
  - Sources included here here
  - Official sites: Tertius, 1BC
  - Included with permission by the authors; please contact the authors for commercial use

- **Caraf**
  - By Clement Charnay, Agnès Braud and Nicolas Lachiche et al
  - All implemented in the Caraf java binaries included here
  - Included with permission by the authors; please contact the authors for commercial use
- **Relaggs** (Krogel and Wrobel, 2001), **Quantiles**, **Cardinalization**
  - Original Proper adapted by Nicolas Lachiche et al
  - GPLv2 license
  - All implemented in the Proper java binaries included here

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

r

Symbols

A

B

C

D

F

I

M

N

O

P

R

S

T

U

W