Appendix A

EUREEKA Documentation

This appendix provides additional information for potential users of the EUREEKA library that implements the ideas presented in our thesis. Section A.1 describes the installation of the framework and Section A.2 explains how to use the software.

A.1. Installation

In the following we provide a guide on how to install the EUREEKA framework. The dependencies (i.e., software necessary for the framework to run) are discussed in Section A.1.1, while the installation and post-install setup issues are covered in Section A.1.2.

A.1.1. Dependencies

EUREEKA is implemented in the Python programming language (see http://www.python.org). The framework is known to work in the Python versions 2.4, 2.5 and 2.6 on a variety of Linux and Windows systems. Apart of Python, the following software is necessary or recommended:

- necessary:
  - the *MySQL* database server (tested with the versions 14.12 and 14.14, however, any version supporting the SQL-92 standard should be fine); see http://www.mysql.com/ for details on the software

  - the *MySQLdb* Python package (tested with versions the 1.2.2 and 1.2.3, however, any version should be fine); see http://sourceforge.net/projects/mysql-python/ for details on the software
  - the *RDFLib* Python package for RDF data processing (tested with the versions 2.4.0 and 2.4.2, however, any version supporting named graphs, N3 format and N3 rule representation should be fine); see http://www.rdflib.net/ for details on the software

- recommended:
  - to enable native EUREEKA knowledge extraction from text:
    * the *NLTK* Python package for natural language processing (tested with the version 2.0b8, however, any version higher than 0.9.9 should be fine); see http://www.nltk.org for details on the software
  - to enable the EUREEKA DBus server:
    * the *DBus* inter-process communication framework (tested with the version 1.2.1, however, any version should be fine); see http://www.freedesktop.org/wiki/Software/dbus for details on the software
    * the *DBus Python binding* (tested with the version 0.82.4, however, any version should be fine); see http://www.freedesktop.org/wiki/Software/DBusBindings for details on the software
  - to enable the text extraction from PDF files:
    * the *PDFMiner* Python package for extracting information from PDF files (tested with the version 20100104, however, any version should be fine); see http://www.unixuser.org/~euske/python/pdfminer/ for details on the software

A.1.2. Installation and Setup

After making sure the necessary dependencies have been satisfied, the EUREEKA framework can be installed from the source Python package available at http://pypi.python.org/pypi/eureeka/0.1. The package has to be unpacked first. Switching into the unpacked directory then, one can install EUREEKA by executing the

```
python setup.py install
```

command. Note that on UNIX, Linux and Mac operating systems, one should execute the command either as a super-user, or using the

```
sudo python setup.py install
```

alternative. This installs EUREEKA into the standard location for Python packages on the particular system. Alternate installation path can be specified using the

```
python setup.py install --home=<dir>
```

command. The `<dir>` folder is then used as a base for the installation. Note that one has to tell Python where to look for the modules in an alternative installation location (see http://docs.python.org/install/#modifying-python-s-search-path for details on how to do that).

When the installation is complete, several setup steps have to be performed before one can start working with EUREEKA. First of all, a database user and a new database dedicated to the particular EUREEKA deployment have to be created in MySQL. This setup will be used for the low-level storage of the EUREEKA grounding and knowledge base later on.

Furthermore, an EUREEKA configuration file has to be created[1]. The configuration file is supposed to contain attribute-value tuples separated by the tabulator character in each line. Comments can be used, too (anything starting with the `#` character will be ignored by EUREEKA until the end of the line). The attributes that must be specified (together with corresponding values) in the configuration file are:

- `DB_HOST` – The address of the MySQL database server host (e.g., `localhost` for the server and EUREEKA present on the same machine).
- `DB_USER` – The name of the EUREEKA database user created previously.
- `DB_PASS` – The password of the EUREEKA database user created previously.
- `DB_NAME` – The name of the EUREEKA database created previously.

[1]The suggested name of the file is `eureeka.cfg`, located in the directory from which the EUREEKA scripts or server are supposed to be launched (the path is used by default). However, it is also possible to specify arbitrary alternative configuration file when executing EUREEKA scripts, as shown later in Sections A.2.2 and A.2.3.

Additional attributes may be specified in the configuration file, too, although they are not essential for some uses of the EUREEKA framework and can often be initialised to default values and/or specified later on an ad hoc basis when they are needed (as shown in Section A.2.2). The non-essential attributes are as follows:

- `N3_IN` – The directory containing N3 RDF files to be incorporated into an EUREEKA knowledge base by a script that is a part of the software package (see Section A.2.2 for details). Defaults to the current working directory (w.r.t. the location from where EUREEKA scripts are being launched).
- `N3_OUT` – The directory for N3 files produced by EUREEKA scripts (e.g., by means of querying, see Section A.2.2 for details). Defaults to the current working directory (w.r.t. the location from where EUREEKA scripts are being launched).
- `TEXT` – The directory with textual input to be processed by the native EUREEKA extraction service (see Section A.2.2 for details). Defaults to the current working directory (w.r.t. the location from where EUREEKA scripts are being launched).
- `DB_DUMP` – The directory for dumps of the EUREEKA knowledge base in the form of MySQL tables (see Section A.2.2 for details). Defaults to the current working directory (w.r.t. the location from where EUREEKA scripts are being launched).
- `RL_PATH` – The path to the N3 rule base to be used with EUREEKA. Defaults to `<cwd>/default.n3`, where `<cwd>` is the current working directory (w.r.t. the location from where EUREEKA scripts are being launched). If such a file is not present, EUREEKA uses the RDFS entailment rules by default.
- `DC_REL` – The path to a file mapping resources to their relevances concerning the knowledge aggregation (see Section 6.1 for details). Defaults to `<cwd>/relevance.txt`, where `<cwd>` is the current working directory (w.r.t. the location from where EUREEKA scripts are being launched). If such a file is not present, EUREEKA uses the default relevance value specified by the `DEF_REL` attribute for all resources.
- `DEF_REL` – The default value of a resource relevance concerning the knowledge aggregation, which is to be used for all resources that are not explicitly mentioned in the resource-to-relevance mapping file given by the `DC_REL` attribute. Defaults to 0.2.

A.2. Usage

This section describes how to use the EUREEKA library. First we discuss an API for programmatic access to the functionalities implemented by the framework in Section A.2.1. Then we give an overview of scripts that expose the most essential capabilities of EUREEKA in a straightforward manner (Section A.2.2). Besides the API and scripts, users may employ an EUREEKA daemon in order to use the framework from other applications implemented in arbitrary programming languages. This is documented in Section A.2.3. The RDF-compatible format used by EUREEKA for the data interchange is specified in Section A.2.4. Finally, Section A.2.5 presents few realistic examples of data processed by EUREEKA.

A.2.1. Library API

The EUREEKA library consists of several modules reflecting the overall architecture presented in Section 7.2 of Chapter 7.

extraction.py

The module is responsible for the knowledge extraction from text. The following main methods are available in the module:

- `callExtRE()`
  - *description*: calls an external method for extracting relations from text
  - *input*:
    - `text` – a string representing the text to be processed
    - `prov_tit` – a string representing the title of the provenance document from which the text comes
    - `np2doc` – a dictionary representing mapping from extracted noun phrases to the corresponding provenance document titles
  - *output*: a tuple consisting of a list of extracted `Relation` objects (see the following section for the details on the object) and an updated `np2doc` mapping

- `extractRels()`
  - *description*: executes the native relation extraction method
  - *input*:
    - `text` – a string representing the text to be processed
    - `prov_tit` – a string representing the title of the provenance document from which the text comes
    - `np2doc` – a dictionary representing mapping from extracted noun phrases to the corresponding provenance document titles
  - *output*: a tuple consisting of a list of extracted `Relation` objects (see the following section for the details on the object) and an updated `np2doc` mapping
- `trimRels()`
  - *description*: trims the list of the extracted relations (using the term frequency-based method introduced in Section 7.2.1 of Chapter 7)
  - *input*:
    - `relations` – a list representing the extracted relations
    - `np2doc` – a dictionary representing mapping from extracted noun phrases to the corresponding provenance document titles
  - *output*: a list of filtered relations with proper certainties set

storage.py

The module is responsible for the low-level storage of EUREEKA knowledge bases and grounding in a relational (MySQL) database. Abstract object wrappers of the low-level storage are offered here as well. The following classes are present in the module:

- `Relation` – A class implementing an intermediate representation of relations, consisting only from lexical expressions, provenance information and certainty degree. Its main purpose is to provide a layer between the input from users and/or various input processors and the EUREEKA knowledge base, as well as to present the knowledge base content back to users in human readable way. The main attributes of the class are:

- `subject` – a string representing the lexical form of the relation's first argument
- `predicate` – a string representing the lexical form of the relation name
- `object` – a string representing the lexical form of the relation's second argument
- `contexts` – a dictionary representing a mapping from the relation context names to their values
- `certainty` – a float representing the relation's certainty
- `score` – a float representing the relation's relevance score (e.g., a temporary significance assigned within the extraction process, or a relevance score of a relation returned within a query answer, computed among the knowledge base entities)
- `provenance` – a dictionary representing a mapping from unique resource provenance identifiers to corresponding human-readable titles

Apart of various rather obvious get/set methods for accessing and setting the attributes, the `Relation` class has the following important methods:

- `getTriples()`
  - *description*: creates an RDF representation of the relation object
  - *input*:
    - `ns_pref` – a string representing the namespace prefix for the URIs used in the relation's RDF representation
  - *output*: a list of triples (i.e., Python tuples) corresponding to the RDFLib format of RDF triple representations, which encodes the particular relation in the RDF interchange format (see Section A.2.4 for details)
- `getHashKey()`
  - *description*: creates a SHA1 sum of the object, which may be used as a unique identifier of the particular statement (e.g., when serialising statements)
  - *input*:
    - no input

    * *output*: a string representing the SHA1 sum of a textual representation of all the statement's lexical elements

- `Entity` – An object for temporary entity representations that are assumed to be already scoped down so that only "subject", "predicate", "object" triples with corresponding degree are taken into account. The class implements algebraic operations and similarities on such entities. The main attributes of the class are as follows:

  – `identifier` – an integer representing the unique identifier of the entity (i.e., a subject)

  – `elements` – a dictionary representing a mapping from ("predicate", "object") tuples to the corresponding certainty degrees

  Apart of the rather obvious set/get methods, the class has the following functions:

  – `__add__()`

    * *description*: element-wise addition of an entity (+)

    * `input`:

      · `other` – an another `Entity` object

    * `output`: the result of the addition of the input entity to the `self`

  – `__mul__()`

    * *description*: multiplication of the entity by a scalar (·)

    * *input*:

      · `x` – a float number

    * *output*: the result of the scalar multiplication of `self` by the input number

  – `__eq__()`

    * *description*: strong entity equality (=)

    * *input*:

      · `other` – an another `Entity` object

    * *output*:

- `__ne__()`
  - *description*: strong entity inequality ($\neq$)
  - *input*:
    - `other` – an another `Entity` object
  - *output*: a boolean value determining whether the input entity and `self` are strongly equal or not
- `__len__()`
  - *description*: entity size in terms of statements with a non-zero certainty
  - *input*:
    - `other` – an another `Entity` object
  - *output*: a boolean value determining whether the input entity and `self` are strongly unequal or not
- `isWeaklyEqualTo()`
  - *description*: weak entity equality ($\simeq$)
  - *input*:
    - `other` – an another `Entity` object
  - *output*: a boolean value determining whether the input entity and `self` are weakly equal or not
- `sim()`
  - *description*: similarity of the entity to another one
  - *input*:
    - `other` – an another `Entity` object
    - `fit_incl` – a boolean value determining whether the entity fitness should be reflected in the computed similarity or not; default value is `True`
    - `penalty` – penalisation for adjustment of the fitness computation; default value is 2.0

        · `even` – a boolean value, determining whether the similarity should be computed as a commutative similarity between `self` and `other`, or as a similarity of `self` to `other` (i.e., taking only the contextual scope of `self` into account); the default value is `False`

    * *output*: a float representing the computed similarity value

- `Grounding` – An object representing the grounding. It realises the storage of the grounding mapping in a database table, as well as a convenient retrieval of the stored values. The main attributes of the class are as follows:

  – `cfg` – a dictionary where the keys and values represent the configuration file attributes and corresponding values, respectively; the class is supposed to be initialised with the dictionary containing data parsed from a configuration file, all the other attributes of the class are initialised accordingly then

  The main methods of the class are:

  – `__len__()`

    * *description*: determines the size of the grounding mapping in terms of the number of unique entity identifiers

    * *input*:

      · no input

    * *output*: an integer representing the size

  – `size()`

    * *description*: determines the size of the grounding mapping w.r.t. either the number of unique entity identifiers, or the number of lexical expressions associated with the identifiers

    * *input*:

      · `in_terms` – a boolean value; if set to `True`, the method returns the number of lexical expressions associated with the unique identifiers by the grounding; the method behaves as `__len__()` otherwise (the default option)

    * *output*: an integer representing the chosen notion of the grounding size

- `add()`
  - *description*: adds a lexical expression to the grounding (possibly assigning a new unique identifier if the term is new)
  - *input*:
    - `term` – a string representing the expression to be added
    - `tag` – a string representing the part of speech tag according to which the expression should be lemmatised (defaults to `None`, which corresponds to a noun)
    - `force_id` – an integer; if the parameter is not `None` (default), its value is used for the unique identifier assignment instead of the automatically computed one
    - `scope` – an integer representing the lexical scope of the expression being added (defaults to `None`)
  - *output*: an integer representing the identifier assigned to the input term
- `addSyn()`
  - *description*: adds a synonymous lexical expression to an already added one
  - *input*:
    - `synonym` – a string representing the synonym expression to be added
    - `term` – a string representing the master term to be associated with the synonym
    - `tag` – a string representing the part of speech tag according to which the expression should be lemmatised (defaults to `None`, which corresponds to a noun)
    - `d` – a float representing the degree of synonymy (defaults to 0.75)
    - `scope` – an integer representing the lexical scope of the expression being added (defaults to `None`)
  - *output*: no output
- `get()`

    * *description*: retrieves a lexical expression for an identifier, or vice versa
    * *input*:
        · `key` – a string or an integer representing a lexical expression or an entity identifier
        · `tag` – a string representing the part of speech tag according to which the expression should be lemmatised (defaults to `None`, which corresponds to a noun)
        · `scope` – an integer representing the lexical scope of the expression being added (defaults to `None`)
    * *output*: either an integer representing the identifier associated with the input, or a lexical expression associated with the input (depending on whether the input is a lexical expression or an identifier, respectively); `None` is returned if no output corresponding to the input is found

– `__getitem__()`

    * *description*: an alias for the `get()` method with default `tag` and `scope` values
    * *input*:
        · `key` – a string or an integer representing a lexical expression or an entity identifier
    * *output*: same as for the `get()` method

– `__delitem__()`

    * *description*: deletes an item from the grounding
    * *input*:
        · `key` – a string or an integer representing a lexical expression or an entity identifier pointing to the entry to be deleted
        · `tag` – a string representing the part of speech tag according to which the expression should be lemmatised (defaults to `None`, which corresponds to a noun)

    * *output*: no output

- `contains()`

    * *description*: checks whether the grounding contains an identifier or a term

    * *input*:

        · `key` – a string or an integer representing a lexical expression or an entity identifier to be checked for

        · `tag` – a string representing the part of speech tag according to which the expression should be lemmatised (defaults to `None`, which corresponds to a noun)

    * *output*: a boolean value determining whether the input is or is not present in the grounding

- `has_key()`

    * *description*: an alias for the `contains` method, accepting only entity identifiers as an input (False is returned if anything else than integer is passed to the method)

    * *input*:

        · `key` – an integer representing an entity identifier to be checked for

    * *output*: a boolean value determining whether the input is or is not present in the grounding

- `isVar()`

    * *description*: checks whether the input symbol is a variable representation or not

    * *input*:

        · `key` – a string representing a lexical expression or an entity identifier to be checked for

    * *output*: a boolean value determining whether the input is or is not a variable

- `getSynonyms()`

    * *description*: retrieves synonyms of an input lexical term or entity identifier
    * *input*:
        · `key` – a string representing the lexical expression or an entity identifier
        · `scope` – an integer representing the lexical scope of the expression being added (defaults to `None`)
    * *output*: a list of (string,float) tuples, representing the lexical expressions synonymous to the input and the corresponding synonymy degrees
  - `getSenses()`
    * *description*: retrieves all entity identifiers corresponding to the input lexical expression
    * *input*:
        · `key` – a string representing the lexical expression
    * *output*: a list of integer values, representing the entity identifiers associated with the input expression by the grounding

- `Entities` – An object representing a set of entity representations (i.e., an essential part of a knowledge base). It realises the storage of entities in a database table, as well as their updates and retrieval. The main attributes of the class are as follows:
  - `cfg` – a dictionary where the keys and values represent the configuration file attributes and corresponding values, respectively; the class is supposed to be initialised with the dictionary containing data parsed from a configuration file, all the other attributes of the class are initialised accordingly then

  The main methods of the class are:

  - `__len__()`
    * *description*: retrieves the size of the object in terms of the number of unique entities
    * *input*:
        · no input
    * *output*: an integer representing the size

- `__getitem__()`
  - *description*: retrieves an entity representation (`Entity` object) from the store according to an input entity identifier
  - *input*:
    - `s` – an integer representing the input identifier
  - *output*: an `Entity` object representing the aggregated representation of all statements with `s` in the "subject" position
- `__delitem__()`
  - *description*: deletes an entity from the store
  - *input*:
    - `key` – an integer representing the identifier of the entity to be deleted
  - *output*: no output
- `get()`
  - *description*: retrieves an entity representation, focusing on a specific provenance only (resulting in plain `__get__()` if there are no statements with the given provenance)
  - *input*:
    - `s` – an integer representing the input identifier
    - `prov` – an integer representing the provenance identifier
  - *output*: an `Entity` object representing the entity composed only of statements with the specified provenance (or an aggregated entity in case there are no such statements)
- `checkConnection()`
  - *description*: checks if the underlying database connection is alive (and refreshes it if it has timed out)
  - *input*:
    - no input

    * *output*: no output

- `keys()`
    * *description*: retrieves identifiers of all stored entities
    * *input*:
        · no input
    * *output*: a list of integers representing the entity identifiers
- `values()`
    * *description*: retrieves aggregated representations of all stored entities
    * *input*:
        · no input
    * *output*: a list of `Entity` objects representing the aggregated stored entities
- `has_key()`
    * *description*: checks whether the store contains an entity or not
    * *input*:
        · `key` – an integer representing the identifier of the entity to be checked for
    * *output*: a boolean value determining whether an entity with the input identifier is present or not
- `update()`
    * *description*: updates the store with the statements from the input entity (the provenance identifier of the updated statements is set to 0, as the method is supposed to be used for updates by inferred statements that have a default provenance identifier 0)
    * *input*:
        · `ent` – an `Entity` object, according to which the store is supposed to be updated

    * *output*: no output

- `addRelations()`

    * *description*: adds multiple relations (i.e., entity representation statements in their lexicalised form) into the store

    * *input*:

        · `relations` – a list of `Relation` objects

        · `g` – a `Grounding` object used for resolving the lexical expressions in the input to entity identifiers

        · `limit` – the number of statements to be stored in the underlying database table using a bulk insert (set to 50000 by default)

    * *output*:

- `getRelations()`

    * *description*: retrieves relations corresponding to input entity identifiers

    * *input*:

        · `subj_ids` – a list of integers representing the entity identifiers for which relations should be retrieved from the store

        · `sim` – a float representing the similarity value by which the actual degree in the retrieved statements should be multiplied (1.0 by default); a non-default value is to be used for instance in rule materialisations

        · `mxsz` – maximum number of most relevant statements to be retrieved (0 by default, meaning that all statements are returned)

        · `contexts` – a dictionary representing the context name and context value identifiers ({} by default); it is used for filtering of the retrieved statements to particular contextual scopes only

        · `g` – a `Grounding` object used for resolving the entity identifiers in the store to lexical expressions in the retrieved relations (`None` by default, meaning that the object is initialised according to the class `cfg` attribute)

- `ftlr` – a dictionary supposed to restrict the relations only to particular subject,object pairs (used when generating answers to single-statement queries with variable predicate), supposed to be in the form of {`s_1:(o_1, d_1), s_2:(o_2,d_2), ... , s_n:(o_n,d_n)`} ({} by default)
- `prov` – an integer representing a provenance filter for the retrieved relations (the default value −1 means that no filter is applied; for any other value, only statements with the specified provenance are retrieved)

* *output*: a list of `Relation` objects representing the retrieved statements

– `updateScores()`

* *description*: updates relevance scores in the underlying store
* *input*:
  - `k` – maximum number of iterations of the HITS algorithm cycle (10 by default)
  - `minf` – minimum predicate frequency for the relation weight computation (25 by default)
  - `bulk_limit` – maximum number of statement score values to be updated in a bulk insert (100000 by default)
* *output*: no output

– `query()`

* *description*: evaluates queries on the entity store
* *input*:
  - `qs` – a sequence of `(s,p,o,d)` statements, where `d` is a degree used for degree-sign checks, and `s,p,o` is in the following acceptable forms `(0,a,b)`, `(a,b,0)`, `(0,c,None)`, `(None,c,0)`, where 0 indicates what one queries for and the `a, b, c` values represent the respective query conditions
  - `partial` – a boolean value determining whether the query evaluation should be soft (i.e., taking also answer entities with fitness lesser than 1 into account) or not (`True` by default)

· `idx_only` – a boolean value determining whether only answer entity indices should be returned or not (`True` by default)

· `omit` – a list of integers representing identifiers of entities to be filtered from the result (`[]` by default, meaning that nothing is filtered out)

· `mxid` – an integer representing maximum number of most relevant entities to appear in the result (0 by default, meaning that all result entities are returned)

· `mxsz` – an integer representing maximum number of most relevant statements to appear in the result (0 by default, meaning that all result statements are returned)

∗ *output*: either a list of integer values representing the answer entities, or a list of `Relation` objects representing the query answer statements (sorted by their relevance)

rules.py

The module is responsible for the import, representation and basic manipulation of rules. The following classes are present in the module:

- `Vertex` – An object for representation of vertices of an antecedent graph, which is crucial for the process of instantiating rule variables. The main attributes of the class are:
  - `identifier` – an integer representing a vertex identifier (i.e., the corresponding rule antecedent entity identifier)
  - `subj_of` – a list of (integer,integer,float) tuples representing the predicates, objects and corresponding degrees, respectively, that reflect the objects bound by the predicates to the identifier playing the role of a subject, with the degrees as specified by the particular tuples
  - `obj_of` – same as `subj_of`, only reflecting the subjects bound by the predicates to the identifier playing the role of an object, with the degrees as specified by the particular tuples

  The main methods of the class are:

- `getID()`
  - *description*: returns the vertex identifier
  - *input*:
    - no input
  - *output*: an integer representing the identifier
- `subjectOf()`
  - *description*: updates the `subj_of` list with a new tuple composed from the input parameters (in the order specified in the list below)
  - *input*:
    - `p` – an integer representing the predicate entity identifier
    - `o` – an integer representing the object entity identifier
    - `d` – a float representing the corresponding relation degree
  - *output*: no output
- `objectOf()`
  - *description*: updates the `obj_of` list with a new tuple composed from the input parameters (in the order specified in the list below)
  - *input*:
    - `s` – an integer representing the subject entity identifier
    - `p` – an integer representing the predicate entity identifier
    - `d` – a float representing the corresponding relation degree
  - *output*: no output
- `getSuccessors()`
  - *description*: retrieves identifiers of successor vertices associated with `self`
  - *input*:
    - `gr` – a list of integers representing entity identifiers to be filtered out from the result (`[]` by default, meaning no filtering is applied)

    * *output*: a list of integers representing the successor vertex identifiers

- `Rule` – An object representing rules (or complex queries with multiple variable statements). The main attributes are:
  - `ruleid` – an integer representing the identifier of the rule
  - `weight` – a float representing the weight of the rule
  - `ante` – a dictionary with integer keys and `Entity` object values, representing the rule antecedent identifiers and the corresponding entities
  - `conseq` – a dictionary with integer keys and `Entity` object values, representing the rule consequent identifiers and the corresponding entities

  The main methods of the class are:
  - `__eq__()`
    * *description*: an equality operator for rule comparison (by means of the strong equality of the antecedent/consequent entities and equality of the rule weight)
    * *input*:
      · `other` – another `Rule` object
    * *output*: a boolean value determining the (non)equality of `self` and `other`
  - `__ne__()`
    * *description*: an inverse of the equality operator
    * *input*:
      · `other` – another `Rule` object
    * *output*: a boolean value determining the (non)equality of `self` and `other`
  - `setID()`
    * *description*: sets the rule identifier
    * *input*:
      · `i` – an integer representing the identifier to be set

    * *output*: no output

- `getID()`
    * *description*: returns the rule identifier
    * *input*:
        · no input
    * *output*: an integer representing the identifier to be set

- `setWeight()`
    * *description*: sets the rule weight
    * *input*:
        · `w` – a float representing the weight to be set
    * *output*: no output

- `getWeight()`
    * *description*: returns the rule weight
    * *input*:
        · no input
    * *output*: a float representing the rule weight

- `getAnteGraph()`
    * *description*: constructs and returns the rule antecedent graph, implemented as a dictionary mapping antecedent entity identifiers to the corresponding `Vertex` objects
    * *input*:
        · `ground` – a `Grounding` object (used for determining variable entity identifiers)
    * *output*: a dictionary with integer keys and `Vertex` object values, representing the constructed antecedent graph

- `getConnection()`

    * *description*: returns a relation connection (i.e., a predicate identifier and corresponding degree) between two antecedent variables
    * *input*:
        · `v1` – an integer representing an antecedent entity identifier
        · `v2` – an integer representing an antecedent entity identifier
        · `direction` – a string determining whether `v1, v2` should be taken as a subject and object, respectively, or the other way around; the `'so', 'os'` (`'so'` is the default one) correspond to the particular options, respectively, and any other options are ignored
    * *output*: an (integer, float) tuple representing the connection predicate and degree values

- `setAntecedent()`
    * *description*: sets a rule antecedent
    * *input*:
        · `m` – an `Entity` object representing the antecedent
    * *output*: no output

- `getAntecedent()`
    * *description*: returns a rule antecedent
    * *input*:
        · `i` – an integer representing the identifier of the antecedent to be retrieved (`None` by default); if nothing is specified, first available antecedent is returned
    * *output*: an `Entity` object representing the antecedent

- `getAntecedents()`
    * *description*: returns all rule antecedents
    * *input*:
        · no input

    * *output*: a list of `Entity` objects representing the antecedents

  - `setConsequent()` beginitemize

  - *description*: sets a rule consequent

  - *input*:

    * `m` – an `Entity` object representing the consequent

  - *output*: no output

- `getConsequent()`

  - *description*: returns a rule consequent

  - *input*:

    * `i` – an integer representing the identifier of the consequent to be retrieved (`None` by default); if nothing is specified, first available consequent is returned

  - *output*: an `Entity` object representing the consequent

- `getConsequents()`

  - *description*: returns all rule consequents

  - *input*:

    * no input

  - *output*: a list of `Entity` objects representing the consequents

Apart of the classes, the module implements the following main function:

- `loadRules()`

  - *description*: loads rules stored in a file

  - *input*:

    * `f` – a string representing the path to the rule file

    * `g` – a `Grounding` object to be used for parsing the rule file

    * `format` – a string representing the format of the file to be parsed; `'n3'` and `'text'` options are accepted, where the former stands for thr N3 rule

representation standard and the latter stands for an interal testing-only EUREEKA plain text format (`'n3'` is default)

– *output*: a list of `Rule` objects parsed from the file

kb.py

The module is responsible for the higher-level knowledge base representation, together with the associated reasoning and querying services. A default entity aggregation function is implemented in the module:

- `avgF()`
  - *description*: an arithmetic mean ordered weighted averaging operator implementation
  - *input*:
    - `a` – a list of arguments that implement an `__add__()` operator and can be multiplied by a float
  - *output*: a float representing the arithmetic mean of the arguments

The following classes are present in the module:

- `KB` – An object implementing the knowledge base functionalities on the top of the `Grouding` mapping and `Entities` store database wrappers. The class has to be initialised with the `cfg` parameter, which is a dictionary representing attribute, value pairs parsed from a configuration file. The main attributes of the class are as follows:
  - `cfg` – a dictionary where the keys and values represent the configuration file attributes and corresponding values, respectively; the class is supposed to be initialised with the dictionary containing data parsed from a configuration file, all the other attributes of the class (e.g., a rule set and specific `Grounding` and `Entities` objects) are initialised accordingly then
  - `rules` – a dictionary with string keys mapped to lists of `Rule` objects; the keys are labels of particular rule sets associated with the knowledge base, while the lists represents the rule sets themselves; a default RDFS entailment rule set is loaded under the `'default'` key every time the class is initialised

  The main methods of the class are:

  - `loadRules()`
    - *description*: loads rules from a file or from a default internal representation of RDFS entailment rules and sets the corresponding `self.rules` key, value pair

    * *input*:
        · `desc` – the rule set description label (`'default'` by default)
        · `path` – the path to the rule file (`None` by default, meaning that the default internal representation of RDFS entailment rules is loaded)
        · `format` – a string representing the format of the file to be parsed; `'n3'` and `'text'` options are accepted, where the former stands for the N3 rule representation standard and the latter stands for an interal testing-only EUREEKA plain text format (`'n3'` is default)
    * *output*: no output

- `getRules()`
    * *description*: retrieves rules according to a given description label
    * *input*:
        · `desc` – a string representing the description of the rule set to be retrieved (`'default'` by default)
    * *output*: a list of `Rule` objects

- `load()`
    * *description*: imports serialised statements from an N3 file (assumed to be in the EUREEKA data interchange format, see Section A.2.4 for details)
    * *input*:
        · `path` – a string representing the path to the N3 file
    * *output*: no output

- `add()`
    * *description*: adds a relation (i.e., a statement) to the knowledge base
    * *input*:
        · `r` – a `Relation` object
    * *output*: no output

- `set()`
    * *description*: sets a new entity into the underlying store, overwriting any possible former content with the same identifier
    * *input*:
        · `m` – an `Entity` object to be set

    * *output*: no output

- `get()`
    * *description*: retrieves an entity from the knowledge base, either as an `Entity` objects, or as a list of `Relation` objects
    * *input*:
        · `x` – an integer representing the identifier of the entity to be retrieved, or a string representing the lexical expression associated with the entity (or entities) to be retrieved
        · `scope` – an integer representing the lexical scope of the string input expression (`None` by default)
        · `get_rels` – a boolean determining whether a list of `Relation` objects or an `Entity` should be returned (`False` by default)
    * *output*:

- `getAll()`
    * *description*: retrieves all entities from the knowledge base
    * *input*:
        · no input
    * *output*: a list of `Entity` objects

- `ask()`
    * *description*: evaluates a query in the EUREEKA human-centric query language
    * *input*:
        · `q` – a string representing the query
    * *output*: a list of `Relation` objects representing the answers

- `updateScores()`
    * *description*: updates relevance scores in the underlying store
    * *input*:
        · `k` – maximum number of iterations of the HITS algorithm cycle (10 by default)
        · `minf` – minimum predicate frequency for the relation weight computation (25 by default)

    * *output*: no output

- `expandRules()`
    * *description*: processes an input rule set in order to make it acceptable by the algorithms described in Chapter 6 (essentially expanding those with relation variables)
    * *input*:
        · `rule_set` – a list of `Rule` objects representing the rule set to be processed
    * *output*: a list of `Rule` objects representing the expanded input rule set
- `execute()`
    * *description*: implementation of the single rule (or query) execution algorithm
    * *input*:
        · `r` – a `Rule` object representing the rule or query to be evaluated
        · `partial` – a boolean determining whether the evaluation should be soft or crisp (`True` by default)
        · `max_res` – an integer representing the maximum number of single variable instance candidates to be retrieved from a database at once (1000 by default)
        · `contexts` – a dictionary with integer keys and values, representing the contextual filtering of the result ({} by default, meaning that no filtering is applied)
        · `focus` – a list of integers representing the focus entities to be set as traversal root (`[]` by default, meaning no focus entities are applied for the traversal)
        · `itlim` – an integer representing maximum number of the algorithm iterations within one execution (1000 by default)
    * *output*: a list of `Relation` objects representing the rule materialisation or query answers
- `apply()`
    * *description*: evaluates a rule set on the knowledge base
    * *input*:
        · `rules` – a list of `Rule` objects to be evaluated

        - `partial` – a boolean determining whether the evaluation should be soft or crisp (`True` by default)
        - `max_res` – an integer representing the maximum number of single variable instance candidates to be retrieved from a database at once (1000 by default)
        - `contexts` – a dictionary with integer keys and values, representing the contextual filtering of the result ({} by default, meaning that no filtering is applied)
        - `upd_only` – a boolean value determining if the knowledge base should be updated according to the evaluation results, or if only the results should be returned (`False` by default)
    * *output*: a list of `Relation` objects representing the evaluation results

- `extend()`
    * *description*: extends the sets of statements associated with input entities by means of applying a particular rule set to them and the knowledge base content
    * *input*:
        - `to_extend` – a list of integers representing the identifiers of entities to be extended
        - `desc` – a string representing the description label of the rule set to be applied (`'default'` by default)
        - `partial` – a boolean determining whether the evaluation should be soft or crisp (`True` by default)
        - `max_res` – an integer representing the maximum number of single variable instance candidates to be retrieved from a database at once (1000 by default)
        - `contexts` – a dictionary with integer keys and values, representing the contextual filtering of the result ({} by default, meaning that no filtering is applied)
        - `itlim` – an integer representing maximum number of the algorithm iterations within one execution (25 by default)
    * *output*: a list of `Relation` objects representing the results

- `closure()`
    * *description*: realises a full knowledge base closure according to a given rule set

  * *input*:
    - `desc` – a string representing the description label of the rule set to be applied (`'default'` by default)
    - `partial` – a boolean determining whether the evaluation should be soft or crisp (`True` by default)
    - `max_res` – an integer representing the maximum number of single variable instance candidates to be retrieved from a database at once (1000 by default)
    - `contexts` – a dictionary with integer keys and values, representing the contextual filtering of the result ({} by default, meaning that no filtering is applied)
  * *output*: no output

srvlib.py

The module provides an object wrapper of the essential EUREEKA functionalities that is to be used by the EUREEKA daemon (see Section A.2.3 for details). The following classes are present in the module:

- `EureekaServer` – An object implementing a DBus IPC wrapper of the basic EUREEKA functionalities. The class has to be initialised with two parameters – `object_path` and `config`. The former is used for initialisation of the DBus service object, while the latter initialises the `cfg` class attribute. The attributes of the class are:
  - `cfg` – a dictionary where the keys and values represent the configuration file attributes and corresponding values, respectively; the class is supposed to be initialised with the dictionary containing data parsed from a configuration file, all the other attributes of the class are initialised accordingly then
  - `kbase` – the knowledge base (a `KB` object) to be exposed via the server wrapper; initialised using the `cfg` value

  The main methods of the class are:

  - `shutdown()`
    * *description*: shuts the DBus EUREEKA server down
    * *input*:
      - no input
    * *output*: no output

- `askQuery()`
  - *description*: evaluates a query in the EUREEKA human-centric query language
  - *input*:
    - `q` – a string representing the query
  - *output*: a list containing two strings; the second string is a RDF/XML representation of the query answer statements (in the EUREEKA interchange format converted from N3 to RDF/XML), the first string is one of `'C', 'Q'`, determining whether the query result relates to a single concept retrieval or a regular query answer, respectively (this distinction is used in the result rendering by the CORAAL application, other applications may choose to use only the RDF/XML result representation, though)
- `extQuery()`
  - *description*: same as `askQuery()`, only the results are extended according to the default rule set associated with the underlying knowledge base
  - *input*:
    - `q` – a string representing the query
  - *output*: same as for `askQuery()`
- `getPForS()`
  - *description*: returns a list of possible predicate terms for the given subject term
  - *input*:
    - `subj` – a string representing the input term
  - *output*: a list of strings representing the retrieved predicate terms
- `getSForP()`
  - *description*: returns a list of possible subject terms for the given predicate term
  - *input*:
    - `pred` – a string representing the input term
  - *output*: a list of strings representing the retrieved subject terms
- `getOForP()`

    * *description*: returns a list of possible object terms for the given predicate term
    * *input*:
        · `pred` – a string representing the input term
    * *output*: a list of strings representing the retrieved object terms

- `getOForSP()`
    * *description*: returns a list of possible object terms for the given subject and predicate term
    * *input*:
        · `subj` – a string representing the input subject term
        · `pred` – a string representing the input predicate term
    * *output*: a list of strings representing the retrieved object terms

- `getPForO()`
    * *description*: returns a list of possible predicate terms for the given object term
    * *input*:
        · `obj` – a string representing the input term
    * *output*: a list of strings representing the retrieved predicate terms

- `getSForOP()`
    * *description*: returns a list of possible subject terms for the given object and predicate term
    * *input*:
        · `obj` – a string representing the input object term
        · `pred` – a string representing the input predicate term
    * *output*: a list of strings representing the retrieved subject terms

- `getPForSO()`
    * *description*: returns a list of possible predicate terms for the given subject and object term
    * *input*:
        · `subj` – a string representing the input subject term

        - `obj` – a string representing the input object term
    - *output*: a list of strings representing the retrieved predicate terms
- `getSynonyms()`
    - *description*: returns a list of all synonyms associated with a term
    - *input*:
        - `t` – a string representing the input term
    - *output*: a list of strings representing the synonyms

util.py

The module defines various global constants and provides auxiliary services for the functional parts of EUREEKA. For instance, default RDFS entailments rules are defined here, as well as features like lemmatisation of compound phrases. We do not provide detailed documentation of the functions corresponding to the auxiliary features, as they are not primarily supposed to be called by users, but rather by other EUREEKA modules. The documentation of the functions can be found in the source of the `util.py` module, though.

A.2.2. Scripts

The EUREEKA scripts realise the communication wrappers outlined in Section 7.2.4 of Chapter 7. Apart of these, EUREEKA installation package contains also auxiliary scripts for input pre-processing. Usage details of the scripts are provided in the following paragraphs. Note that a special attention has to paid to passing any multi-word arguments to any of the scripts. For instance, on most UNIX and Linux systems, an argument containing spaces has to be enclosed in double quotes (i.e., the " character) in order to be processed correctly by the scripts.

ext.py

The script is contained in the `scripts` directory of the EUREEKA installation (package). It serves for extracting statements from a set of text files. The result of the extraction is a set of N3 files that contain statements (in the EUREEKA data interchange format) extracted from the corresponding input files.

The input files are assumed to be in the XML format, adhering to the following simple structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<xml>
<doc title="DOCUMENT TITLE">
DOCUMENT TEXT
</doc>
</xml>
```

Plain text files can be transformed into this structure using the `txt2xml.py` script described below. PDF files can be transformed similarly, only applying the `pdf2txt.py` script before `txt2xml.py`.

The following command line arguments can be used with the script:

- `[-h | --help]` – no argument value necessary, results in printing a help message and terminating
- `[-c | --config]` – specifies an alternative configuration file (by default, the file `eureeka.cfg` in the current working directory is loaded)
- `[-i | --input]` – specifies an alternative path containing the input files (by default, the `TEXT` attribute value from the configuration file is used)
- `[-o | --output]` – specifies an alternative path for storing the output files (by default, the `N3_IN` attribute value from the configuration file is used)

kbm.py

The script is contained in the `scripts` directory of the EUREEKA installation (package). It serves for initialisation and updates of the EUREEKA knowledge base, as well as for relevance-based score computation.

The following command line arguments can be used with the script:

- `[-h | --help]` – no argument value necessary, results in printing a help message and terminating
- `[-c | --config]` – specifies an alternative configuration file (by default, the file `eureeka.cfg` in the current working directory is loaded)
- `[-m | --import]` – no argument value necessary, results in importing statements from the N3 files in the `N3_IN` directory specified by the configuration file (assumed to be in the EUREEKA data interchange format)
- `[-n | --input]` – specification of an alternative path for the statement import
- `[-i | --init]` – no argument value necessary, results in initialisation of the knowledge base

- `[-d | --dump]` – no argument value necessary, results in a database dump of the knowledge base into the `DB_DUMP` directory specified by the configuration file
- `[-u | --update-scores]` – no argument value necessary, results in (re)computation of the relevance-based entity scores

que.py

The script is contained in the `scripts` directory of the EUREEKA installation (package). It serves for the knowledge base querying and entity retrieval.

The following command line arguments can be used with the script:

- `[-h | --help]` – no argument value necessary, results in printing a help message and terminating
- `[-c | --config]` – specifies an alternative configuration file (by default, the file `eureeka.cfg` in the current working directory is loaded)
- `[-g | --get]` – specification of a term corresponding to an entity or entities to be retrieved; the results are stored in an N3 file in the `N3_OUT` directory specified by the configuration file; the file is in the EUREEKA data interchange format and its filename (without the extension) is a SHA1 sum of the query string
- `[-q | --query]` – specification of a query to be executed on the knowledge base; the results are stored in an N3 file in the `N3_OUT` directory specified by the configuration file; the file is in the EUREEKA data interchange format and its filename (without the extension) is a SHA1 sum of the query string
- `[-x | --extend]` – same as the above option, however, the query results are extended according to a default rule set associated with the knowledge base

pdf2txt.py

The script is contained in the `scripts` directory of the EUREEKA installation (package). It serves for converting PDF files into plain text files, making use of a pure-Python PDF processing library pdfminer.

The script is to be launched as:

```
pdf2txt.py dir1 dir2
```

where `dir1` is the path to a directory where the PDF files to be converted are stored and `dir2` is the path to a directory where the converted text files (with the same filename, only having the `.txt` extension) are to be stored.

txt2xml.py

The script is contained in the `scripts` directory of the EUREEKA installation (package). It converts plain text files into XML with title annotations described in the section dealing with the `ext.py` script. The document titles to be used in the XML files are selected according to a mapping parsed from a file specified in a command-line argument. By default, the filename of the file begin converted is selected.

The following command line arguments can be used with the script:

- `[-h | --help]` – no argument value necessary, results in printing a help message and terminating
- `[-d | --dir]` – specification of the input directory containing the plain text files to be converted in situ (i.e., the converted XML files are stored in the same directory)
- `[-m | --map]` – specification of a path to the file containing the filename to title mapping (the file is supposed to contain rows of the corresponding filename, title tuples divided by the tabulator character)

A.2.3. Daemon

The DBus daemon (or server) is located in the `scripts` directory of the EUREEKA installation (package). The daemon is launched by the `eureekad.py` script, which initialises the `EureekaServer` class (located in the `srvlib.py` module) and an binds the class with an associated DBus connection (a system bus). It is recommended to execute the script in the background, so that it runs indefinitely.

The following command line arguments can be used with the `eureekad.py` script:

- `[-h | --help]` – no argument value necessary, results in printing a help message and terminating
- `[-c | --config]` – specifies an alternative configuration file (by default, the file `eureeka.cfg` in the current working directory is loaded)

The daemon can be stopped by terminating the corresponding process in the current implementation (it safely dies then and can be relaunched without any problems).

When the daemon is running, any application that can access the corresponding DBus inter-process communication bus can call the `EureekaServer` class, pass input parameters to them (following the signatures of the particular methods) and get their outputs. The bus name of the bus on which EUREEKA exposes its functionalities is `ie.deri.sw.smile.koraal.dbus.eureeka` and the corresponding DBus object address is `/ie/deri/sw/smile/koraal/dbus/eureeka`. Note that the access of particular applications to the EUREEKA bus has to be configured in the DBus policy settings.

The DBus signatures of the methods that can be accessed via the EUREEKA daemon are as follows:

- `askQuery()`
  - input signature: `'s'`
  - output signature: `'as'`
- `extQuery()`
  - input signature: `'s'`
  - output signature: `'as'`
- `getPForS()`
  - input signature: `'s'`
  - output signature: `'as'`
- `getSForP()`
  - input signature: `'s'`
  - output signature: `'as'`
- `getOForP()`
  - input signature: `'s'`
  - output signature: `'as'`
- `getOForSP()`
  - input signature: `'ss'`
  - output signature: `'as'`
- `getPForO()`
  - input signature: `'s'`
  - output signature: `'as'`
- `getSForOP()`
  - input signature: `'ss'`
  - output signature: `'as'`
- `getPForSO()`
  - input signature: `'ss'`
  - output signature: `'as'`

- `getSynonyms()`
  - input signature: `'s'`
  - output signature: `'as'`

A.2.4. Data Interchange Format

The data interchange format specification is a pretty straightforward serialisation of the `Relation` object defined in the `storage.py` module. The particular relations (i.e., statements from which the EUREEKA entities consist) are stored as descriptions in the N3 format. A generic description is as follows:

DESCPREF:*DESC_ID*
EUPREF:`hasCertainty "`*D*`"^^<http://www.w3.org/2001/XMLSchema#float>;`
EUPREF:`hasObject "`*OBJ*`";`
EUPREF:`hasPredicate "`*PRED*`";`
EUPREF:`hasProvenance "`*PROV_ID : PROV_TIT*`";`
EUPREF:`hasScore "`*S*`"^^<http://www.w3.org/2001/XMLSchema#float>;`
EUPREF:`hasSubject "`*SUBJ*`";`
EUPREF:*CTX_NAME_1* `"`*CTX_VALUE_1*`";`
...
EUPREF:*CTX_NAME_N* `"`*CTX_VALUE_N*`"`
.

The meaning of the particular elements and expressions in the description is:

- *EUPREF* – a namespace prefix of the EUREEKA RDF presentation elements (`<http://ontologies.smile.deri.ie/eureeka/presentation>`)
- *DESCPREF* – a namespace prefix of the description (when serialising statements from EUREEKA, the prefix is automatically generated from the *EUPREF* value; applications may choose to use arbitrary prefixes, though)
- `has*` – predicates having the obvious lexical representations associated with the statement as values (string literals); two predicates deserve a special remark, though:
  - the `hasScore` value represents either a relevance-based score of the statement (when the statement is retrieved from an EUREEKA knowledge base), or a combination of the general relevance-based score derived from the knowledge base content and particular relevance of the statement to a query (when the statement represents a query answer); the values can be used for sorting the statement by applications in either case; zero or one `hasScore` predicate may occur in a single statement (zero occurrence meaning that a statement has no score associated with it)

  - the value of the `hasProvenance` predicate is a string with a special syntax representing the provenance identifier (usually a SHA1 sum of the title in the current EUREEKA applications) and the corresponding provenance document title, separated by the `':'` character; zero or multiple `hasProvenance` predicates may occur in a single statement representation

- *CTX_NAME_i*, *CTX_VALUE_i* – name and value of a context associated with a statement; zero or multiple such predicates may occur in a single statement representation

Apart of the data interchange format specified above, EUREEKA can import rules in the N3 format. A description of how to represent rules in N3 is provided for instance at http://dig.csail.mit.edu/2007/Talks/0110-rules-tbl. Note that the N3 rule representation does not currently offer any consensual means for representing degrees, thus the degrees and weights of the rules represented in N3 and imported into EUREEKA are supposed to be equal to 1.

A.2.5. Examples

In the following, we provide a simple example of real data processed by EUREEKA. The data are taken from a testing case, processing a corpora of scientific news articles. Note that a more comprehensive, realistic and, perhaps most importantly, user friendly way of browsing data processed by EUREEKA is the CORAAL application (see http://coraal.deri.ie:8080/coraal/ and also Chapter 8).

In the following, a sample of processed text is provided.

```
<?xml version="1.0" encoding="UTF-8"?>
<xml>
<doc title="Bacteria remove gold from soil and deposit it on grains
where they live, scientists say">


Australian scientists have found the strongest evidence so far that
bacteria play a key role in forming gold grains and nuggets.  They
have found bacteria that remove gold from the soil and deposit pure
grains of it around them.  Researcher Dr Frank Reith from the CRC
for Landscape Environments and Mineral Exploration, and colleagues
gathered their evidence at two separate mines and publish their results
today in the journal Science.  At the Tomakin mine on the south coast
of New South Wales and the Hit or Miss mine in tropical north Queensland,
most gold is hidden away in quartz veins, in amounts that are invisible
even to high-powered microscopes.  But the soil above the mines also
contains grains and nuggets of gold that have somehow found their
way out of the quartz.  "There are a probably a lot of processes
involved," Reith says.
```

```
He and other scientists have long suspected that bacteria play a
part, but it's an idea that has generated some scepticism.  To test
the theory, Reith sifted the soil above the mines and collected gold
grains 0.1-2.5 millimetres across, and then subjected them to several
experiments.  First, Reith looked at the grains under a high-powered
electron microscope to confirm that they contained bacteria-shaped
bubbles of gold.  "They're little lumps on the surface," Reith says.

Next, he looked for organic matter on the grains, as evidence that
bacteria had been growing on their surfaces.  Finally, he used a
technique called polymerase chain reaction to look for bacterial
DNA on the surfaces of the grains to show that living bacteria are
still there.  "The DNA would have degraded if the bacteria weren't
around any more," Reith says.  About 80% of the grains had living
bacteria on them, Reith says.  And the only bug that was found on
all those positive samples was Ralstonia metallidurans.  "These grains
come from areas that are almost at the opposite ends of Australia,"
Reith notes.  "We were pretty happy with that." Reith thinks the
Ralstonia bacteria play an important role in the microbial ecosystem
in soil, helping to rid it of the soluble gold that most other species
find toxic.  "This is the guy whose job it is to get the toxic gold
out of the environment so the other bacteria can live a happy life,"
he says.  In the future, the gold-loving bugs could prove a boon
to industry, Reith says.  Perhaps they could be used to improve gold
processing, or even be useful as a marker for the presence of gold
that's otherwise invisible.

</doc>
</xml>
```

The corresponding N3 file with extracted statements is as follows (the dots in the prefixes are a shorthand for the address specified in the prefix _4):

```
@prefix _3:
<http://.../01a290b541d4cbfd4ae968b2ea2c7861c6044c22/>.
@prefix _4:
<http://ontologies.smile.deri.ie/eureeka/presentation>.
@prefix _5:
<http://.../01a290b541d4cbfd4ae968b2ea2c7861c6044c22/3>.
@prefix _6:
<http://.../01a290b541d4cbfd4ae968b2ea2c7861c6044c22/7>.
@prefix _7:
<http://.../01a290b541d4cbfd4ae968b2ea2c7861c6044c22/474715855>.


_5:c34108884d153e43e6a61023a70c04081d0a713
_4:hasCertainty "0.259098821438"^^<http://www.w3.org/2001/XMLSchema#float>;
```

```
_4:hasObject "grain";
_4:hasPredicate "contain";
_4:hasProvenance "01a290b541d4cbfd4ae968b2ea2c7861c6044c22 :  Bacteria
remove gold from soil and deposit it on grains where they live, scientists
say";
_4:hasSubject "mine".

_7:bf4799fce2f6300388f9780dac3f4e5
_4:hasCertainty "0.0997571651644"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "0.1-2.5 millimetre";
_4:hasPredicate "collect";
_4:hasProvenance "01a290b541d4cbfd4ae968b2ea2c7861c6044c22 :  Bacteria
remove gold from soil and deposit it on grains where they live, scientists
say";
_4:hasSubject "mine".


_6:f4655c6122cf833df7e698b6797bd49fd031a2e
_4:hasCertainty "-0.323873526797"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "grain";
_4:hasPredicate "type";
_4:hasProvenance "01a290b541d4cbfd4ae968b2ea2c7861c6044c22 :  Bacteria
remove gold from soil and deposit it on grains where they live, scientists
say";
_4:hasSubject "nugget of gold".

_3:c02235dd3c41756c0d555ec4cbaf4194737a1e74
_4:hasCertainty "0.350857896689"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "bacteria";
_4:hasPredicate "live";
_4:hasProvenance "01a290b541d4cbfd4ae968b2ea2c7861c6044c22 :  Bacteria
remove gold from soil and deposit it on grains where they live, scientists
say";
_4:hasSubject "grain".

_3:c209336e392041532ef82d2d276693945cc90853
_4:hasCertainty "0.132297933821"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "nugget of gold";
_4:hasPredicate "contain";
_4:hasProvenance "01a290b541d4cbfd4ae968b2ea2c7861c6044c22 :  Bacteria
remove gold from soil and deposit it on grains where they live, scientists
say";
_4:hasSubject "mine".

_3:c4547ad70473d5287e9b73eedd7eda6d8cfb908c
_4:hasCertainty "-0.323873526797"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "nugget of gold";
```

```
_4:hasPredicate "type";
_4:hasProvenance "01a290b541d4cbfd4ae968b2ea2c7861c6044c22 :  Bacteria
remove gold from soil and deposit it on grains where they live, scientists
say";
_4:hasSubject "grain".
```

The sample above does not contain any contexts. An example of an extracted statement with actual context is as follows:

```
_3:a058df34b82ee96be3b787db72e7f38266b88c0e
_4:hasCertainty "0.106478109628"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "foray";
_4:hasPredicate "make";
_4:hasProvenance "f1a3ea6ee583d13c1e6fd5a4f74d62ccfc2bcae0 :  This
fishy creature, which was up to 3 metres long, walked on land with
fins that have arm-like bones";
_4:hasSubject "Tiktaalik fish";
_4:in "search of food";
_4:onto "land".
```

Let us add the following statements (a sample from a simple testing "metal ontology") to the knowledge extracted from the article about gold and bacteria:

```
_0:stmt1
_4:hasCertainty "1.0"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "raw gold"; _4:hasPredicate "type"; _4:hasProvenance
"54ddd54fbff1cd40bd723fa89d40e3ce21cdf2a7 :  metal ontology"
_4:hasSubject "nugget of gold".

_0:stmt2
_4:hasCertainty "1.0"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "raw metal"; _4:hasPredicate "type"; _4:hasProvenance
"54ddd54fbff1cd40bd723fa89d40e3ce21cdf2a7 :  metal ontology"
_4:hasSubject "raw gold".

_0:stmt3
_4:hasCertainty "1.0"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "transitive"; _4:hasPredicate "type"; _4:hasProvenance
"54ddd54fbff1cd40bd723fa89d40e3ce21cdf2a7 :  metal ontology"
_4:hasSubject "type".
```

Together with the rule specifying the semantics of transitive relations:

```
?r :type :transitive.  ?x ?r ?y.  ?y ?r ?z => ?x ?r ?z.
```

we can infer the following statement:

```
_0:stmt4
_4:hasCertainty "1.0"^^<http://www.w3.org/2001/XMLSchema#float>;
```

```
_4:hasObject "raw metal"; _4:hasPredicate "type"; _4:hasProvenance
"767167373c26954db00c50d8d256309b5439f032 :  inferred"
_4:hasSubject "nugget of gold".
```

Finally, submitting the query

```
?  :  NOT type :  nugget of gold
```

to EUREEKA, the following set of statements is returned:

```
_3:c4547ad70473d5287e9b73eedd7eda6d8cfb908c
_4:hasCertainty "-0.323873526797"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "nugget of gold";
_4:hasPredicate "type";
_4:hasProvenance "01a290b541d4cbfd4ae968b2ea2c7861c6044c22 :  Bacteria
remove gold from soil and deposit it on grains where they live, scientists
say";
_4:hasScore "1.0"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasSubject "grain".

_3:c02235dd3c41756c0d555ec4cbaf4194737a1e74
_4:hasCertainty "0.350857896689"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasObject "bacteria";
_4:hasPredicate "live";
_4:hasProvenance "01a290b541d4cbfd4ae968b2ea2c7861c6044c22 :  Bacteria
remove gold from soil and deposit it on grains where they live, scientists
say";
_4:hasScore "1.0"^^<http://www.w3.org/2001/XMLSchema#float>;
_4:hasSubject "grain".
```

An application for processing the query results may either render all statements, or only the one containing the query term[2], or return only the subject representing the actual query answer.

[2]This would hide the other statement, which represents a realistic example of noise in EUREEKA, as it is rather non-sensical.