

DL-Learner Manual

Jens Lehmann

July 8, 2011

DL-Learner is a machine learning framework for OWL and description logics. It includes several learning algorithms and is easy to extend. DL-Learner widens the scope of Inductive Logic Programming to description logics and the Semantic Web. This manual provides the entry point to using DL-Learner and explains its basic concepts.

Contents

1	What is DL-Learner?	2
2	Getting Started	3
3	DL-Learner Architecture	4
4	DL-Learner Components	5
4.1	Knowledge Sources	5
4.2	Reasoner Components	6
4.3	Learning Problems	7
4.4	Learning Algorithms	8
5	DL-Learner Interfaces	10
6	Extending DL-Learner	11
7	General Information	13

1 What is DL-Learner?

DL-Learner is an open source framework for (supervised) machine learning in OWL and Description Logics (from instance data). We further detail what this means:

OWL stands for “Web Ontology Language”. In 2004, it became the W3C¹ standard ontology language². As such it is one of the fundamental building blocks in the Semantic Web and has been used in several scenarios on and off the web. OWL is based on *description logics* (DLs), which are a family of knowledge representation languages. We refer to [Baader et al., 2007] for an introduction to description logics. Since OWL formally builds on description logics, we can apply DL-Learner to knowledge bases in OWL or a variety of description languages.

Machine Learning is a subfield of Artificial Intelligence, which focuses on detecting patterns, rules, models etc. in data. Often, this involves a training process on the input data. In *Supervised* learning, this data is labelled, i.e. we are given a number of input-output mappings. Those mappings are also called *examples*. If the output is binary, then we distinguish positive and negative examples. DL-Learner as a framework is not restricted to supervised learning, but all algorithms currently build into it, are supervised.

In the most common scenario we consider, we have a background knowledge base in OWL/DLs and additionally, we are given positive and negative examples. Each example is an individual in our knowledge base. The goal is to find an OWL *class expression*³ such that all/many of the positive examples are *instances* of this expression and none/few of the negative examples are instances of it. The primary purpose of learning is to find a class expression, which can classify unseen individuals (i.e. not belonging to the examples) correctly. It is also important that the obtained class expression is easy to understand for a domain expert. We call these criteria *accuracy* and *readability*.

As an example, consider the problem to find out whether a chemical compound can cause cancer⁴. In this case, the background knowledge contains information about chemical compounds in general and certain concrete compounds we are interested in. The positive examples are those compounds causing cancer, whereas the negative examples are those compounds not causing cancer. The prediction for the examples has been obtained from experiments and long-term research trials in this case. Of course, all examples have to be described in the considered background knowledge. A learning algorithm can now derive a class expression from examples and background knowledge, e.g. such a class expression in natural language could be “chemical compounds containing a phosphorus atom”. (Of course, in practice the expression will be more complex to obtain a reasonable accuracy.) Using this class expression, we can not classify unseen chemical compounds.

¹<http://www.w3.org>

²<http://www.w3.org/2004/OWL/>

³http://www.w3.org/TR/owl2-syntax/#Class_Expressions

⁴see <http://dl-learner.org/wiki/Carcinogenesis> for a more detailed description

2 Getting Started

DL-Learner is written in Java, i.e. it can be used on almost all platforms. Currently, Java 6 or higher is required. To install the latest release, please visit the download page⁵ and extract the file on your harddisk. In the top level directory, you will notice several executables. Those files ending with `bat` are Windows executables, whereas the corresponding files without file extension are the Non-Windows (e.g. Linux, Mac) executables. To test whether DL-Learner works, please run the following on the command line depending on your operating system:

```
dllearner examples/father.conf      (Non-Windows Operating System)
dllearner.bat examples/father.conf  (Windows Operating System)
```

Conf files, e.g. `examples/father.conf` in this case, describe the learning problem and specify which algorithm you want to use to solve it. In the simplest case they just say where to find the background knowledge to use (in the OWL file `examples/father.owl` in this case) and the positive and negative examples (marked by “+” and “-”, respectively). When running the above command, you should get something similar to the following:

```
DL-Learner 2010-08-07 command line interface
starting component manager ... OK (157ms)
initialising component "OWL file" ... OK (0ms)
initialising component "fast instance checker" ... OK (842ms)
initialising component "pos neg learning problem" ... OK (0ms)
initialising component "refinement operator based
  learning algorithm II" ... OK (14ms)

starting top down refinement with: Thing (50% accuracy)
more accurate (83,33%) class expression found: male
solutions (at most 20 are shown):
1: (male and hasChild some Thing) (accuracy 100%, length 5, depth 3)
Algorithm terminated successfully.

number of retrievals: 4
retrieval reasoning time: 0ms (0ms per retrieval)
number of instance checks: 93 (0 multiple)
instance check reasoning time: 1ms ( 0ms per instance check)
overall reasoning time: 1ms (11,016% of overall runtime)
overall algorithm runtime: 17ms
```

The first part of the output tells you which components are used (more on this in Section 4). In the second part you see output coming from the used learning algorithm, i.e. it can print information while running (“more accurate (83,33%) class description found”) and the final solutions, it computed. The results are displayed in

⁵http://sourceforge.net/project/showfiles.php?group_id=203619

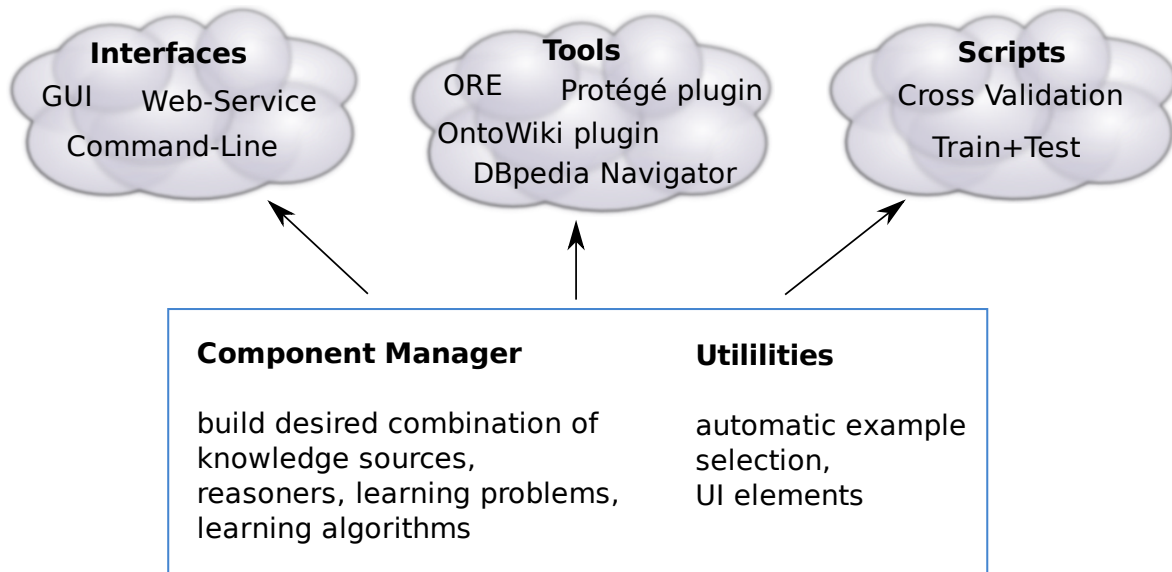


Figure 1: Overall structure of the DL-Learner software.

Manchester OWL Syntax⁶. There can be several solutions, in which case they are ordered with the most promising one in the first position. In this case the only solution is `male` and `hasChild` some `Thing` defining the class `father`. The last part of the output contains some runtime statistics.

3 DL-Learner Architecture

DL-Learner (see also [Lehmann, 2009]) consists of core functionality, which provides Machine Learning algorithms for solving the learning problem, support for different knowledge base formats, an OWL library, and reasoner interfaces. There are several interfaces for accessing this functionality, a couple of tools which use the DL-Learner algorithms, and a set of convenience scripts. The general structure is illustrated in Figure 1.

To be flexible in integrating new learning algorithms, new kinds of learning problems, new knowledge bases, and new reasoner implementations, DL-Learner uses a component based model. Adding a component can be done by subclassing the appropriate class and adding the name of the new class to the “components.ini” file (more on that in Section 6).

There are four types of components (knowledge source, reasoning service, learning problem, learning algorithm). For each type, there are several implemented components and each component can have its own configuration options as illustrated in Figure 2. Configuration options can be used to change parameters/settings of a component. In Section 4, we describe the components in DL-Learner and their configuration options.

⁶http://www.co-ode.org/resources/reference/manchester_syntax/

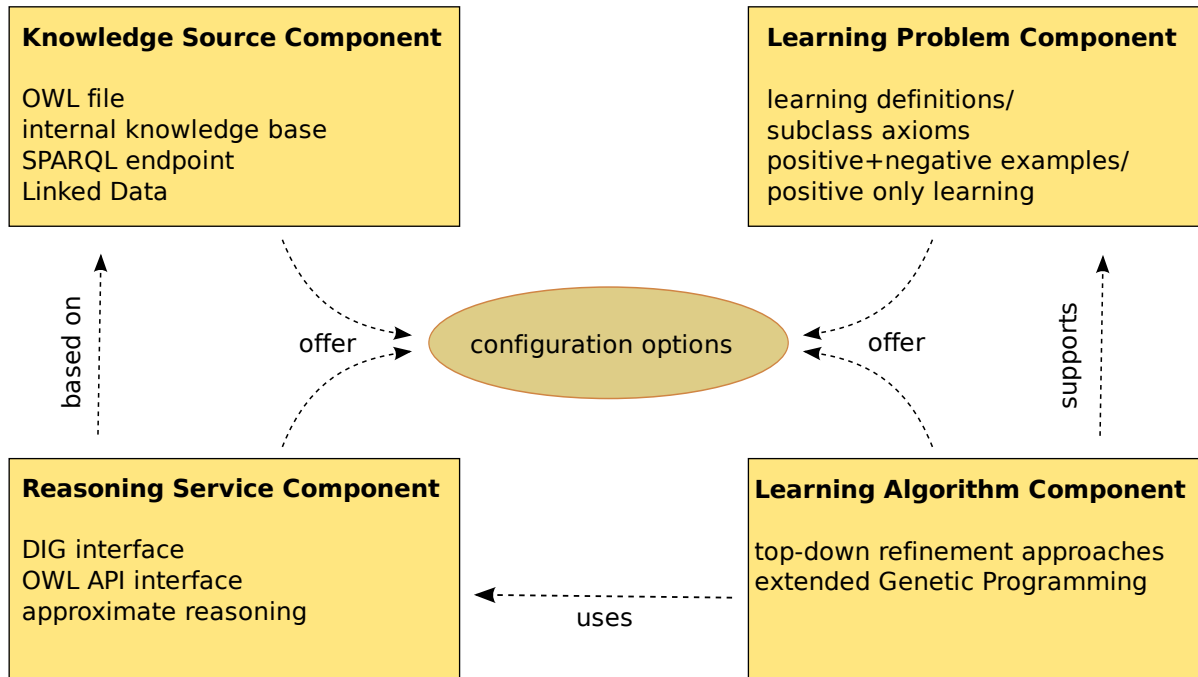


Figure 2: The architecture of DL-Learner is based on four component types, which can each have their own configuration options. DL-Learner uses a component manager to organise all components.

4 DL-Learner Components

In this part, we describe concrete components currently implemented in DL-Learner. Each of the subsections contains a list of components according to the type specified in the subsection heading. Note that this does not constitute a full description, i.e. we omit some components and many configuration options. The purpose of the manual is to obtain a general understanding of the implemented components. A full list, which is generated automatically from the source code, can be found in `doc/configOptions.txt` including the default values for all options and their usage in conf files.

4.1 Knowledge Sources

Knowledge sources have a URI and can be included in conf files using `import("$url");`, e.g. `import("ontology.owl")`. Depending on the file ending, DL-Learner will guess the correct type of knowledge source. If you want to overwrite this, you can use a second parameter with value `OWL`, `KB`, or `SPARQL`, e.g. `import("ontology.owl","OWL")`.

OWL File DL-Learner supports OWL files in different formats, e.g. RDF/XML or N-Triples. If there is a standard OWL format, you want to use, but is not supported by DL-Learner please let us know. We use the OWL API for parsing, so all formats

supported by it can be used⁷.

KB File KB files are an internal non-standardised knowledge representation format, which corresponds to description logic syntax except that the special symbols have been replaced by ASCII strings, e.g. AND instead of \sqcap . You can find several KB files in the examples folder. The `doc/kbFileSyntax.txt` contains an EBNF description of the language.

SPARQL Endpoint DL-Learner allows to use SPARQL endpoints as background knowledge source, which enables the incorporation of very large knowledge bases, e.g. DBpedia[Auer et al., 2008], in DL-Learner. This works by using a set of start instances, which usually correspond to the examples in a learning problem, and then retrieving knowledge about these instances via SPARQL queries. The obtained knowledge base fragment can be converted to OWL and consumed by a reasoner later since it is now sufficiently small to be processed in reasonable time. Please see [Hellmann et al., 2009] for details about the knowledge fragment extraction algorithm. Some options of the SPARQL component are:

- instances: Set of individuals to use for starting the knowledge fragment extraction. Example use in conf file:

```
sparql.instances = {"http://dbpedia.org/resource/Matt_Stone",
  "http://dbpedia.org/resource/Sarah_Silverman"};
```

- recursionDepth: Maximum distance of an extracted individual from a start individual. This influences the size of the extracted fragment and depends on the maximum property depth you want the learned class expression to have. Example use in conf file: `sparql.recursionDepth = 2;`
- saveExtractedFragment: Specifies whether the extracted ontology is written to a file or not. If set to true, then the OWL file is written to the cache dir. Example usage: `sparql.saveExtractedFragment = true;`

Many further options allow to modify the extracted fragment on the fly or fine-tune the extraction process. The extraction can be started separately by running and modifying `org.dllearner.test.SparqlExtractionTest`. The collected ontology will be saved in the DL-Learner directory.

4.2 Reasoner Components

Several reasoner components are implemented, which can be interfaces to concrete reasoner implementations. To select a component in a conf file, use `reasoner=$value;`, where `$value` is one of `digReasoner`, `fastInstanceChecker`, or `owlAPIReasoner`, which are explained below. Note that OWLlink reasoners can be attached via the OWL API interface.

⁷for a list see <http://owlapi.sourceforge.net>

OWL API The OWL API reasoner interface can be used in conjunction with the Pellet, FaCT++, HermiT and OWLlink reasoners. The only option allows to switch between them:

- `reasonerType`: Selects the desired reasoner. By default, Pellet is used. Usage: `owlAPIReasoner.reasonerType = fact;`. Pellet, FaCT++ and HermiT are already included in DL-Learner. Note that for FaCT++, you need to add `-Djava.library.path=lib/fact/64bit` (or 32bit) to the Java command. You can also use an external OWLlink reasoner by setting the reasoner type to `owlLink`. You can then use the option `owlLinkURL` to specify the URL of the OWLlink reasoner (`http://localhost:8080/` by default).

DIG DIG 1.1⁸ is an interface to description logic reasoners and supported by a large variety of reasoners including Pellet, FaCT++, KAON2, and Racer Pro. The major drawback is that the current version DIG 1.1 is not aligned with the OWL specification and therefore lacks several features, which are crucial to the more recent learning algorithms in DL-Learner. If you still want to use the DIG interface, you have to download a DIG capable reasoner and start the DIG server there. DL-Learner communicates with the reasoner using the XML based protocol over HTTP.

Fast Instance Checker Instance checks, i.e. testing whether an individual is instance of a class, is the major reasoner task in many learning algorithms. This reasoner is a self-development of the DL-Learner project. It remedies some problems related to Machine Learning and the Open World Assumption in OWL and therefore is not correct w.r.t. OWL semantics. (See [Badea and Nienhuys-Cheng, 2000] Section 4 for an explanation.) Furthermore, it provides an improved performance for instance checks by precomputing some inferences and keeping them in memory. The fast instance checker is build on top of Pellet and the default reasoner component in DL-Learner.

4.3 Learning Problems

In the introductory Sections 1 and 2, we described a specific learning problem where positive and negative examples are given. In practice different variations of similar problems occur. You can switch between the different problems using `problem=$value;`, where `$value` is one of `posNegLPStandard`, `posOnlyLP`, `classLearning`. The default is `posNegLPStandard`.

Positive and Negative Examples Let the name of the background ontology be \mathcal{O} . The goal in this learning problem is to find an OWL class expression C such that all/many positive examples are instances of C w.r.t. \mathcal{O} and none/few negative examples are instances of C w.r.t. \mathcal{O} . As explained previously, C should be learned such that it generalises to unseen individuals and is readable. The important

⁸<http://dl.kr.org/dig/>

configuration options of this component are obviously the positive and negative examples, which are often indicated with + and - signs in conf files as an optional shortcut to using e.g. `posNegLPStandard.positiveExamples = {...}`.

Positive Examples This learning problem is similar to the one before, but without negative examples. In this case, it is desirable to find a class expression which closely fits the positive examples while still generalising sufficiently well. For instance, you usually do not want to have `owl:Thing` as a solution for this problem, but neither do you want to have an enumeration of all examples.

Class Learning In class learning, you are given an existing class A within your ontology \mathcal{O} and want to describe it. It is similar to the previous problem in that you can use the instances of the class as positive examples. However, there are some differences, e.g. you do not want to have A itself as a proposed solution of the problem, and since this is an ontology engineering task, the focus on short and readable class expressions is stronger than for the two problems mentioned before. The learner can also take advantage of existing knowledge about the class to describe.

4.4 Learning Algorithms

The implemented algorithms vary from very simple (and usually inappropriate) algorithms to sophisticated ones. You can switch between the different algorithms using `algorithm=$value;`, where `$value` is one of `bruteForce`, `random`, `gp`, `refinement`, `refexamples`, `celoe`, `el` and `disjunctiveEL`. The default is `refexamples`.

Brute Force : This algorithm tests all class expressions up to a specified length, which you can set using e.g. `bruteForce.maxlength = 7`.

Random Guesser : This algorithm randomly generates class expressions. To do this, it creates trees, which can be mapped to class expressions. Its main parameter is the number of created trees, which you can set using e.g. `random.numberOfTrees = 5`.

Genetic Programming (GP) : GP is a well-known general problem solution method, which can be adapted to class expression learning. The adaption is straightforward. In DL-Learner, however, an additional genetic refinement operator was implemented, which has shown to improve GP performance[Lehmann, 2007]. Some options are:

- number of individuals: The individual count is the size of each generation in a GP algorithm. It is one of the most crucial parameters. Setting it to a higher value usually means investing more computational resource for increasing the likelihood that a solution will be found. Usage: `gp.numberOfIndividuals = 100`.
- refinement probability: This is used to specify how likely the usage of the genetic refinement operator should be, e.g. `gp.refinementProbability = 0.6` means that it will be selected 60% of the time.

The GP algorithm has 15 more options documented in `doc/configOptions.txt`.

Refinement This is a top down refinement operator approach, which is described in [Lehmann and Hitzler, 2008] and based on insights in [Lehmann and Hitzler, 2007]. Some options include:

- target language: The standard target language of this algorithm is $\mathcal{ALCN}(\mathcal{D})$. However, you can change the language, i.e. you can exclude the \forall constructor by using `refinement.useAllConstructor = false;`. Similar options exist for \exists , \neg , cardinality restrictions, and boolean datatypes.
- maximum execution time: If there is no perfect solution of a given problem, the algorithm can potentially run forever (in practice it will run out of memory). It is therefore often interesting to limit the execution time. You can use e.g. `refinement.maxExecutionTimeInSeconds = 100` to say that the algorithm should run for at most 100 seconds. Often, it will run slightly longer than the maximum execution time since it waits for the next internal loop of the algorithm to stop gracefully.

The algorithm supports a range of further options. For instance, one can specify which classes and properties must not occur in resulting class expressions.

Refexamples (OCEL) The previous algorithm has been extended to make more sophisticated use of background knowledge [Lehmann and Hitzler, 2010] and therefore run more efficiently on many problems. It also supports double datatypes and `hasValue` restrictions (which again can be turned on or off as desired). It also includes explicit noise handling through the `noisePercentage` option. This is currently the default and recommend algorithm for learning from positive and negative examples. More than 30 options can be set to control its behaviour. However, apart from the target language the most important setting is noise, which should be optimised for the given problem.

Class Expression Learning for Ontology Engineering (CELOE) Currently CELOE is the best class learning algorithm available within DL-Learner. It uses the same refinement operator as OCEL, but a completely different heuristics. Furthermore, it guarantees that the returned class expressions are minimal in the sense that one cannot remove parts of them without getting an inequivalent expression. Furthermore, it makes use of existing background knowledge in coverage checks. Statistical methods are used to improve the efficiency of the algorithm such that it scales to large knowledge bases. While it was originally designed for ontology engineering, it can also be used for other learning problems and might even be superior to the other algorithms in many cases (not well-tested yet). Note that many configuration options of OCEL were dropped for the sake of simplicity, but may be introduced if needed.

EL Tree Learner (ELTL) This algorithm has EL as its target language, i.e. is specialized for this relatively simple language. There are two algorithms: `e1` learns EL

concepts and `disjunctiveEL` learns disjunctions of EL concepts.

Please note that while components are interchangeable, it is not possible to arbitrarily combine them. For instance, the newer learning algorithms do not work with the DIG interface, since it does not provide the necessary inference tasks. Furthermore, a learning algorithm can specify which learning problems it can solve, i.e. we do not require it to be able to solve each learning problem. Table 1 provides a compatibility matrix. Note that this can change in future releases, because algorithms may be extended to support new learning problems or drop support for them.

learning problem	BF	RG	GP	Ref	OCEL	CELOE	ELTL
pos only	x	x				x	
pos neg	x	x	x	x	x	x	x
class learning	x	x				x	

Table 1: Learning problem - learning algorithm compatibility matrix in DL-Learner. Legend: BF = brute force, RG = random guesser, Ref = Refinement

5 DL-Learner Interfaces

One interface you have already used in Section 2 is the command line. There are two executables, which can be used for starting DL-Learner on the commandline: `dl-learner` and `quickstart`. The first one takes a conf file as argument, whereas the latter one lists all conf files in the examples folder and allows you to select one of those.

Apart from the command line, there is also a prototypical graphical interface. You can use `gui` (or `gui.bat`) to start it. Optionally, a conf file can be passed as argument. The main GUI window has four tabs corresponding to the four different types of components and a run tab to execute the learning algorithm. Using the GUI, you can assemble the desired combination of components and options. The **File** menu allows you to load a conf file or save the current configuration to a conf file. The GUI implementation is currently prototypical, so please report any bugs or feature requests you have (see Section 7). Since the GUI uses the component manager, it will automatically evolve when new components and options are added.

A third interface through which DL-Learner can be accessed programmatically is a web service. You can execute `ws` (or `ws.bat`) to start the web service. It is based on the Java API for XML Web Services (JAX-WS), which is included in Java 6 or higher. Executing the command will start a web server on port 8181 of your local machine. The WSDL can be accessed via `http://localhost:8181/services?wsdl`. You can use a WSDL viewer to see the supported operations or view the JavaDoc of the corresponding Java file⁹. Some examples for calling the web service from PHP can be found in the

⁹viewable online at <http://dl-learner.org/javadoc/org/dllearner/server/DLLearnerWS.html>

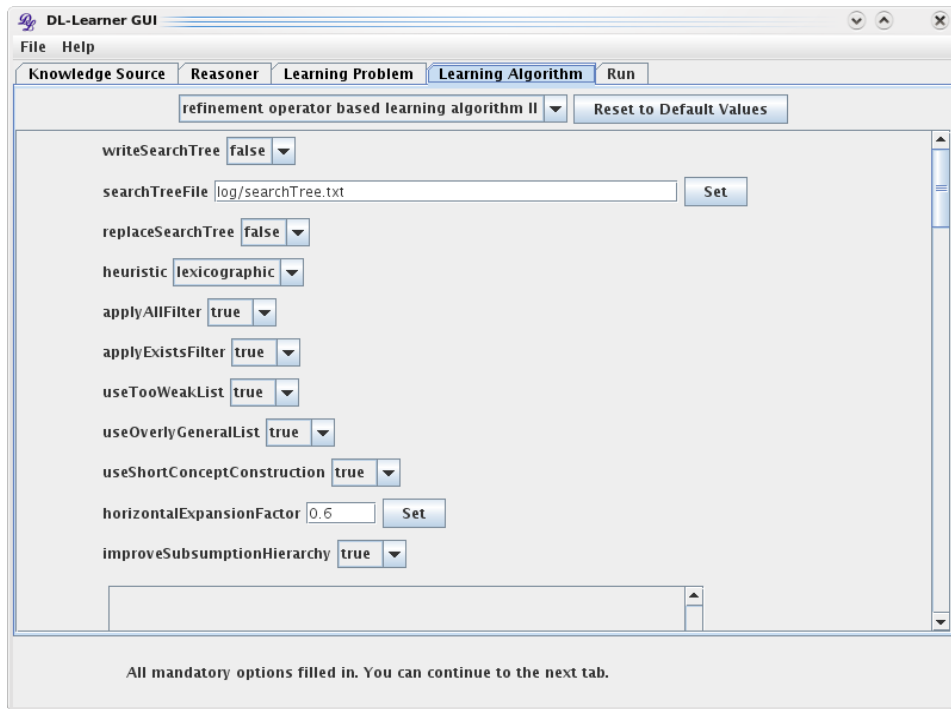


Figure 3: GUI screenshot showing the learning algorithm tab. The UI allows you to set different options and then proceed to the next tab and execute the algorithm.

DL-Learner subversion repository¹⁰.

Another means to access DL-Learner, in particular for ontology engineering, is to use the OntoWiki and Protégé plugins. The OntoWiki plugin is not officially released yet, but can be used in the SVN version of OntoWiki. The Protégé 4 plugin can be installed either by downloading it from the DL-Learner download page or directly within Protégé 4 by clicking on “File”, “Preferences”, “Plugins”, “Check for Downloads” now and selecting the DL-Learner plugin. For more information and a screencast see the Protégé plugin wiki page¹¹.

6 Extending DL-Learner

DL-Learner is open source and component based. If you want to develop a specific part or extension of a class expression learning algorithm for OWL, then you are invited to use DL-Learner as a base. This allows you to focus on the part you want to implement while being able to use DL-Learner as a library and access it through one of the interfaces.

If you want to create a new component, then you first have to decide on the type of your component. To implement a concrete component, you have to subclass one of the

¹⁰in the directory `src/php-examples/`:

<http://dl-learner.svn.sourceforge.net/viewvc/dl-learner/trunk/src/php-examples/>

¹¹<http://dl-learner.org/wiki/ProtegePlugin>

following classes and implement their abstract methods:

- `org.dllearner.core.KnowledgeSource`
- `org.dllearner.core.ReasonerComponent`
- `org.dllearner.core.LearningProblem`
- `org.dllearner.core.LearningAlgorithm`

You then have to add your component to `lib/components.ini` such that it is registered in the component manager when DL-Learner starts up. If you want to use configuration options in your component, you need to create a static method as follows:

```
public static Collection<ConfigOption<?>> createConfigOptions() {
    List<ConfigOption<?>> options = new LinkedList<ConfigOption<?>>();
    options.add(new IntegerConfigOption("maxDepth",
        "maximum depth of generated concepts/trees", 5));
    return options;
}
```

This creates an option with name `maxDepth`, the given description, and a default value of 5. To add further options, simply add more of them to the collection. If desired, running `org.dllearner.scripts.ConfigJavaGenerator` generates a file for you in package `org.dllearner.configurators` to access the options of your component programmatically. These configurator classes are particularly useful to build scripts or tools on top of DL-Learner components. An example for this can be found in `org.dllearner.scripts.NewSample`.

Currently, the following configuration option types exist (new ones can be implemented if necessary):

- boolean, e.g. `useCache`
- string (a set of allowed strings can be specified), e.g. `cacheDir`
- URL, e.g. `reasonerURL`
- int (min and max value can be specified), e.g. `maxDepth`
- double (min and max value can be specified), e.g. `noisePercentage`
- set of strings, e.g. `positiveExamples`
- list of string tuples, e.g. `replaceObject`

Restricting to these option types this gives us the possibility to build very flexible user interfaces. Whenever, a new component or a new configuration option for a component is added, the current user interfaces (GUI, web service, commandline) will automatically support it without any or only minimal code changes.

This quick introduction only serves as an entry point to get you started. For more detailed questions about how to extend DL-Learner, please drop us a message in the DL-Learner mailing list.

7 General Information

- Homepage: <http://dl-learner.org>
- Sourceforge.net project page: <http://sourceforge.net/projects/dl-learner/>
- Tracker (bugs, features): http://sourceforge.net/tracker/?group_id=203619
- Mailing Lists: http://sourceforge.net/mail/?group_id=203619
- Contact: lehmann@informatik.uni-leipzig.de (please use the mailing list if possible)
- Latest Release: http://sourceforge.net/project/showfiles.php?group_id=203619

References

- [Auer et al., 2008] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. (2008). DBpedia: A nucleus for a web of open data. In *Proceedings of the 6th International Semantic Web Conference (ISWC)*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer.
- [Baader et al., 2007] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2007). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.
- [Badea and Nienhuys-Cheng, 2000] Badea, L. and Nienhuys-Cheng, S.-H. (2000). A refinement operator for description logics. In Cussens, J. and Frisch, A., editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 40–59. Springer-Verlag.
- [Hellmann et al., 2009] Hellmann, S., Lehmann, J., and Auer, S. (2009). Learning of OWL class descriptions on very large knowledge bases. *Int. J. Semantic Web Inf. Syst.*, 5(2):25–48.

References

- [Lehmann, 2007] Lehmann, J. (2007). Hybrid learning of ontology classes. In Perner, P., editor, *Machine Learning and Data Mining in Pattern Recognition, 5th International Conference*, volume 4571 of *Lecture Notes in Computer Science*, pages 883–898. Springer.
- [Lehmann, 2009] Lehmann, J. (2009). DL-Learner: learning concepts in description logics. *Journal of Machine Learning Research (JMLR)*, 10:2639–2642.
- [Lehmann and Hitzler, 2007] Lehmann, J. and Hitzler, P. (2007). Foundations of refinement operators for description logics. In Blockeel, H., Ramon, J., Shavlik, J. W., and Tadepalli, P., editors, *Inductive Logic Programming, 17th International Conference, ILP 2007, Corvallis, OR, USA, June 19-21, 2007*, volume 4894 of *Lecture Notes in Computer Science*, pages 161–174. Springer. Best Student Paper Award.
- [Lehmann and Hitzler, 2008] Lehmann, J. and Hitzler, P. (2008). A refinement operator based learning algorithm for the ALC description logic. In Blockeel, H., Ramon, J., Shavlik, J. W., and Tadepalli, P., editors, *Proc. of 17th Int. Conf. on Inductive Logic Programming (ILP 2007)*, volume 4894 of *Lecture Notes in Computer Science*, pages 147–160. Springer. Best Student Paper.
- [Lehmann and Hitzler, 2010] Lehmann, J. and Hitzler, P. (2010). Concept learning in description logics using refinement operators. *Machine Learning journal*, 78(1-2):203–250.