Plugin Development for KReator

Matthias Thimm

October 18, 2010

Abstract. This document explains how to develop plugins for KReator using the JAVA programming language. KReator supports the inclusion of plugins that define knowledge bases, parser, writers, learner, and configuration components using a specific approach to statistical relational learning. These plugins can easily be developed by implementing a set of interfaces and using the JSPF framework.

1 Overview

KReator (Finthammer *et al.*, 2009; Thimm *et al.*, 2010; Beierle *et al.*, 2010) is an integrated development environment for various tasks involving reasoning and learning with approaches to *statistical relational learning* (Getoor and Taskar, 2007). Due to its modular architecture KReator allows the integration of different models by including plugins. By developing a plugin for KReator there is no need for such cumbersome tasks like implementing a user interface or developing the logical foundations that are used by most approaches. KReator provides for all these high-level components and the developer of a plugin can concentrate on implementing the core components for an individual approach. At the moment, there exist simple plugins for *Bayesian Logic Programs* (Getoor and Taskar, 2007; Ch. 10), *Markov Logic Programs* (Getoor and Taskar, 2007; Ch. 12), and *Relational Maximum Entropy* (Loh *et al.*, 2010). Developers of models for statistical relational learning are encouraged to provide a KReator plugin for their specific model type. The task of implementing a KReator plugin is eased by the JSFP plugin framework (Delsaux and Biedert)

This document is a tutorial that guides through the process of developing a plugin for KReator. It explains what steps are necessary for implementing a working plugin that seamlessly integrates into KReator by presenting the available interfaces.

2 Setup and Plugin architecture

In order to start developing a KReator plugin download the latest version of KReator from `kreator.cs.tu-dortmund.de`. In general, KReator comes as a Java library (with the *.jar*-extension) that my be bundled for a specific platform (for example as an *.app*-file for Mac OS X). To use the KReator JAR for your plugin you only need to download the *unbundled* JAR file.

As a first step you should create a new JAVA project using your favorite development environment and you must make sure that the KReator JAR is within the build path of this project (for example in Eclipse[1] by adding the KReator JAR as an external JAR to your project). Next, you should create your specific plugin class (by implementing the interface `edu.cs.ai.kreator.models.KreatorModelPlugin`) which is the only necessary class for a plugin. This class is responsible for providing all necessary information about your plugin to the KReator core. Specific details on how to implement this class can be found below, but first we give an overview on how to implement your actual model in terms of knowledge bases, parser, learner, writers, and configuration components.

2.1 Knowledge bases

A knowledge base is the central structure used for an approach to statistical relational learning (and—of course—knowledge representation in general). A knowledge base can be developed using the comprehensive logic library of KReator (which defines commons structures such as predicates, variables, rules, ...), see the API of `edu.cs.ai.kreator.logic` for details. A knowledge base must extend the abstract class `Knowledgebase` from the package `edu.cs.ai.kreator.models` and implement the following methods.

`public Double query(Query query)`

This method computes the probability of the given query. A query is a structure of the form $(\phi \mid \psi)$ and represents the question for the probability of ϕ (some ground atom) given that ψ (some conjunction of ground literals) is true.

`public List<AtomExpression> generateData(int numDataSets)`

This method generates data matching the knowledge base's structure and probabilities used for testing learning algorithms. For example, suppose your knowledge base consists of a simple rule "If $A(X)$ is true then $B(X)$ is true with probability 0.9" then a single data set would consist of $\{A(c_1) = true, B(c_1) = true\}$ or $\{A(c_2) = true, B(c_2) = false\}$ (for some new constants c_1, c_2). The more data sets are requested the more the distribution of the data should fit the intended probabilities in the knowledge base.

Note: The implementation of this method is optional. If you do not want to provide an implementation of this method for your model just add the line

```
    throw new UnsupportedOperationException();
```

to the body of the method.

`public String getDescription()`

This method returns a short textual description for the knowledge base type, e. g. for a Bayesian Logic Program this method returns "*Bayesian Logic Program*".

[1]`http://www.eclipse.org`

`public Set<GeneralizedPredicate> getAppearingPredicates()`

This method returns the set of all predicates that are mentioned in the knowledge base.

`public String toLatex()`

This method returns a string representation of the knowledge base formatted to be directly used for LaTeX documents.

2.2 Parser

A parser for a knowledge base is responsible for reading some knowledge base defined in some syntax into a knowledge base object and for each knowledge base type at least one parser must be defined. A knowledge base parser must extend the abstract class `KnowledgebaseParser` from the package `edu.cs.ai.kreator.models` and implement the following methods.

`public Knowledgebase parse(String kbaseString)`

Parses the given string into a knowledge base object.

`public String getSupportedFileExtension()`

Returns a string representation of the supported file extension of the parser (without the dot).

`public String getAcronymForIcons()`

Returns an acronym for knowledge bases supported by this parser. This acronym should be at most three letters long and will be attached to icons of the knowledge bases in the project tree of the main KReator window.

`public Color getColorForIcons()`

The color used for writing the acronym of the supported knowledge bases in the project tree of the main KReator window.

2.3 Learner

A learner takes the job of learning a knowledge base from data. A learner must extend the abstract class `edu.cs.ai.kreator.models.Learner` and implement the following methods.

`public Knowledgebase learnModel(List<AtomExpression> data)`

Learns a new knowledge base that fits the given data.

`public Knowledgebase learnModel(Knowledgebase startingPoint, List<AtomExpression> data)`

Learns a new knowledge base that fits the given data and uses the given knowledge base as a starting point for structure learning.

2.4 Writer

A writer is the counter part for a parser and takes the job of providing a string representation of knowledge base that can be read by its corresponding parser. A writer must be implemented when providing a learner (otherwise a learned knowledge base cannot be written). A writer has to extend the abstract class `edu.cs.ai.kreator.models.KnowledgebaseWriter` and implement the following methods.

`public String write(Knowledgebase kb)`

Writes the given knowledge base into a string.

`public String getSupportedFileExtension()`

Returns a string representation of the supported file extension of this writer (without the dot).

2.5 Configuration

Most formalisms are configurable as—for example—parameters for the process of learning can be set. Each plugin may provide some configuration options that are integrated in the KReator configuration that manages loading and saving. A configuration must implement the interface `edu.cs.ai.kreator.models.ModelConfiguration` with the following method.

`public ConfigurationCategory getConfig()`

This method returns a configuration category that may contain other configuration categories or configuration options, see the API of `edu.cs.ai.kreator.control.config` for more information.

2.6 Syntax Highlighting

If you want to provide syntax highlighting for the language of the knowledge base type of your plugin you have to implement several classes. Most importantly you have to provide for a lexer which implements the interface `jsyntaxpane.Lexer`. The easiest way to build a lexer is using JFlex. For more information on how to implement a lexer for the JSyntaxPane framework (which is used inside KReator) consult the project home page[2].

Furthermore you have to extend `jsyntaxpane.syntaxkits.KreatorSyntaxKit` and pass your lexer within the single constructor of your syntax kit.

[2]`http://code.google.com/p/jsyntaxpane/wiki/Customizing`

```
public MySyntaxKit() {
  super(new MyLexer());
}
```

In order to relate the syntax highlighting to the corresponding files you have to implement the following method.

`public String getSupportedFileExtension()`

Returns the file extension of the files, for which this syntax kit provides syntax highlighting.

If the syntax highlighting of the plugin should be configurable, you can simply add an instance of `edu.cs.ai.kreator.control.config.StyleConfigurationCategory` to the model configuration.

```
myConfigurationCategory.add(
        new StyleConfigurationCategory(new MySyntaxKit()));
```

2.7 Plugin component

Having implemented classes for knowledge bases, parser, writers, learner, and configuration, the plugin needs to provide these information to the KReator core. This is done by implementing the interface `edu.cs.ai.kreator.models.KreatorModelPlugin` with the methods

```
public List<...> providesKnowledgebaseClasses()
public List<...> providesKnowledgebaseParserClasses()
public List<...> providesKnowledgebaseWriterClasses()
public List<...> providesLearnerClasses()
public List<...> providesConfigurationClasses()
public List<...> providesSyntaxHighlightingClasses()
```

In each of these methods the corresponding classes should be returned in a list, e. g. for knowledge bases via

```
public List<Class<? extends Knowledgebase>> providesKnowledgebaseClasses(){
      List<Class<? extends Knowledgebase>> knowledgebases =
         new ArrayList<Class<? extends Knowledgebase>>();
      knowledgebases.add(MyKnowledgebase.class);
      return knowledgebases;
}
```

In order to enable KReator to discover the plugin the JSPF library[3] has to be included in the build path and the annotation

```
@PluginImplementation
```

has to be placed directly before the plugin class definition inside the class' file (and you have to import the package `net.xeoh.plugins.base.annotations.*`).

[3] `http://code.google.com/p/jspf/`

3 Threading, System output and Logging

KReator builds heavily on threading for performing all tasks such as reasoning and learning. The developer should keep this in mind when implementing a model and should pay attention to defining break points inside time-consumable computations. These break points are used by KReator to pause or abort the current process. This is done by simply adding the following line at regular intervals in time-consumable computations:

```
KreatorMain.getWorkerController().threadWaitOrAbort();
```

Whenever the above line is executed KReator checks whether the current process should be paused or aborted due to a user input.

To output some status information on the currently running process to the user the following line can be used:

```
KreatorMain.getWorkerController().getMyWorker()
          .getConsole().printVerboseLine(SOME_TEXT);
```

Text passed over in this way is outputted to the console of the current process in the main KReator window.

For debugging and especially error reporting purposes it is always a good idea to log what a process is currently doing. For this purpose KReator uses the *log4j*[4] logging system. When developing a plugin we advise to use this logger as well. In order to enable logging using log4j the developer has to add the corresponding log4j library to his build path and add the line

```
public static final Logger LOG =
          Logger.getLogger(MyClass.class);
```

inside the current class. By invoking `LOG.debug(TEXT)`, `LOG.info(TEXT)`, ..., the developer can log messages on various logging levels, see the API of log4j for more information. Messages logged this way are shown in the KReator debugging text pane.

4 Testing

In order to test your plugin inside KReator you can export you plugin project into a JAR and directly into KReator. But especially during development time making a detour by starting KReator and loading your plugin can be tedious. So, for debugging purposes you can start KReator with your plugin already installed using the following simple command:

```
KreatorMain.startWithPlugin(new MyPlugin());
```

It suffices to implement a `main` method in one of your plugin classes that comprises of just this single line to start a KReator instance with your plugin.

[4]`http://logging.apache.org/log4j/`

5 Publishing

When you have finished developing your plugin for KReator please consider publishing it in form of a JAR to the KReator community at `kreator.cs.tu-dortmund.de`.

References

Christoph Beierle, Marc Finthammer, Gabriele Kern-Isberner, and Matthias Thimm. Automated reasoning for relational probabilistic knowledge representation. In *Proceedings of the Fifth International Joint Conference on Automated Reasoning (IJCAR'10)*, Edinburgh, UK, July 2010.

Nicolas Delsaux and Ralf Biedert. JSPF: The Java Simple Plugin Framework. `http://code.google.com/p/jspf/`.

Marc Finthammer, Sebastian Loh, and Matthias Thimm. Towards a Toolbox for Relational Probabilistic Knowledge Representation, Reasoning, and Learning. In *Proceedings of the First Workshop on Relational Approaches to Knowledge Representation and Learning*, pages 34–48, Paderborn, Germany, September 2009.

Lise Getoor and Ben Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

Sebastian Loh, Matthias Thimm, and Gabriele Kern-Isberner. On the problem of grounding a relational probabilistic conditional knowledge base. In *Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR'10)*, Toronto, Canada, May 2010.

Matthias Thimm, Marc Finthammer, Sebastian Loh, Gabriele Kern-Isberner, and Christoph Beierle. A system for relational probabilistic reasoning on maximum entropy. In *Proceedings of the 23rd International FLAIRS Conference (FLAIRS'10)*, Daytona Beach, USA, May 2010.