Knime: The Konstanz Information Miner

Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Koetter, Thorsten Meinl, Peter Ohl, Christoph Sieb, and Bernd Wiswedel.

Abstract— **The Konstanz Information Miner is a modular environment which enables easy visual assembly and interactive execution of a data pipeline. It is designed as a teaching, research and collaboration platform, which enables easy integration of new algorithms, data manipulation or visualization methods as new modules or nodes. In this White Paper we describe some of the design aspects of the underlying architecture and briefly sketch how new nodes can be incorporated.**

I. Overview

The need for modular data analysis environments has increased dramatically over the past years. In order to make use of the vast variety of data analysis methods around, it is essential that such an environment is easy and intuitive to use, allows for quick and interactive changes to the analysis and enables the user to visually explore the results. To meet these challenges a data pipelining environment is an appropriate model. It allows the user to visually assemble and adapt the analysis flow from standardized building blocks, at the same time offering an intuitive, graphical way to document what has been done.

Knime, the Konstanz Information Miner provides such an environment. Figure 1 shows a screenshot of an example analysis flow. In the center, a flow is reading in data from three sources and processes it in several, also parallel analysis flows, consisting of preprocessing, modeling, and visualization nodes. On the left a repository of nodes is shown. From this large variety of nodes, one can select data sources, data preprocessing steps, model building algorithms, visualization techniques as well as model I/O tools and drag them onto the workbench where they can be connected to other nodes. The ability to have all views interact graphically creates a powerful environment to explore the data sets at hand. Knime is written in Java and it's graphical workflow editor is implemented as an Eclipse [1] plug-in. It is easy to extend through an open API and a data abstraction framework, which allows for new nodes to be quickly added in a well-defined way.

In this white paper we will describe some of the internals of Knime in more detail. More information as well as downloads can be found at `http://www.knime.org`.

II. Architecture

The architecture of Knime was designed with three main principles in mind:

- visual, interactive framework: data flows should be combined by simple drag&drop from a variety of processing units. Customized applications can be modelled through individual data pipelines.
- modularity: Processing units and data containers should not depend on each other in order to enable easy distribution of computation and allow for independent development of different algorithms. Data Types are encapsulated, that is, no types are predefined, new types can easily be added bringing along type specific renderers and comparators. New types can be declared compatible to existing types.
- easy expandability: It should be easy to add new processing nodes, or views and distribute them through a simple plug&play principle without the need for complicated install/deinstall procedures.

In order to achieve this, a data analysis process consists of a pipeline of nodes, connected by edges that transport either data or models. Each node processes the arriving data and/or model(s) and produces results on its outputs. Figure 2 schematically illustrates this process. The type of processing ranges from simple data operations such as filtering or merging to more complex statistical functions, such as computations of mean, standard deviation or linear regression coefficients to computation intensive data modeling operators (clustering, decision trees, neural networks, to name just a few). In addition most of the modeling nodes allow to interactively explore their results through accompanying views. In the following we will briefly describe the underlying schemata of data, node, workflow management and how the interactive views communicate.

A. Data Structures

All data flowing between nodes is wrapped within a class called `DataTable` which holds meta-information concerning the type of its columns and the actual data. The data can be accessed by iterating over instances of `DataRow`. Each row contains a unique identifier (or primary key) and a specific number of `DataCell` objects which hold the actual data. The reason to avoid access by Row ID or index is scalability, that is, the desire to be able to process large amounts of data and therefore not be forced to keep all of the rows in memory for fast, random access. Figure 3 shows an UML diagram of the main underlying data structure.

B. Nodes

Nodes in Knime are the most general processing unit and usually resemble one visual node in the workflow. The class `Node` wraps all functionality and makes use of user defined implementations of a `NodeModel`, possibly a `NodeDialog`, and one or more `NodeView` instances if appropriate. Nei-

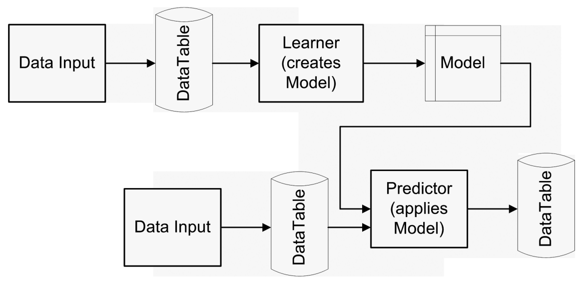

Fig. 2. A schematic for the flow of data and models in a Knime-workflow.

Konstanz University, Department of Computer and Information Science, Fach M712, 78457 Konstanz, Germany. email: `Michael.Berthold@uni-konstanz.de`

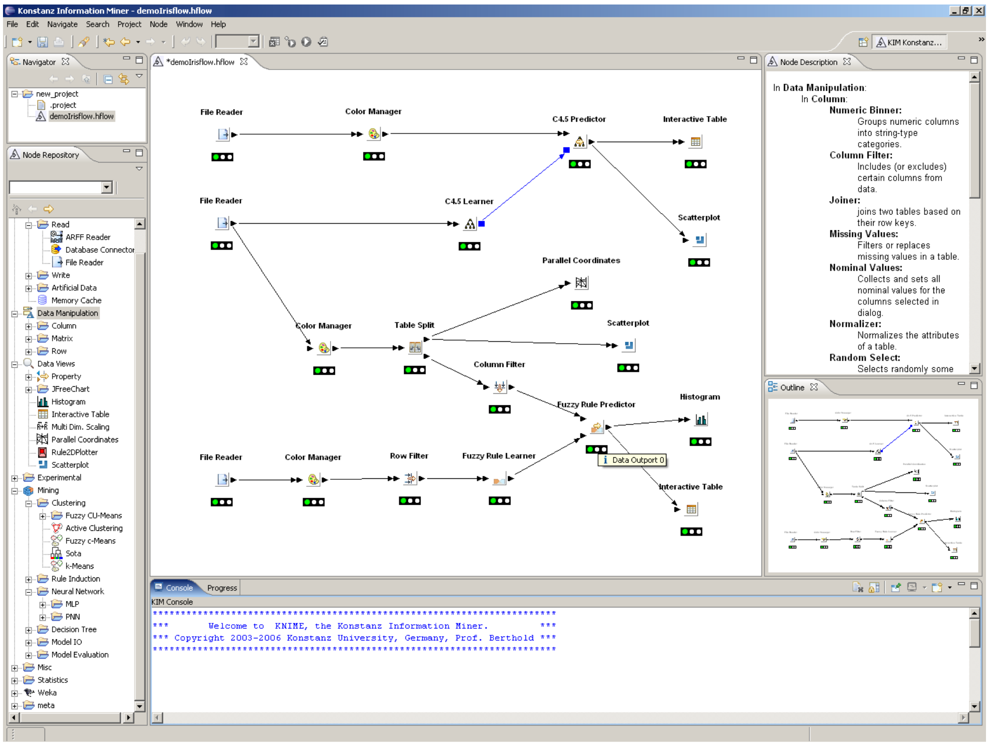

Fig. 1. An example analysis flow inside Knime.

ther dialog nor view must be implemented if no user settings or views are needed. This schema follows the well-known Model-View-Controller design pattern. In addition, for the input and output connections, each node has a number of `Inport` and `Outport` instances which can either transport data or model(s). Figure 4 shows an UML diagram of this structure.

C. Workflow Management

Workflows in Knime are essentially graphs connecting nodes, or more formally, a direct acyclic graph (DAG). The `WorkflowManager` allows to insert new nodes and to add directed edges (connections) between two nodes. It also keeps track of the status of nodes (configured, executed, ...) and returns, on demand, a pool of executable nodes. This way the surrounding framework can freely distribute the workload among a couple of parallel threads or – in the future – even a distributed cluster of servers. Thanks to the underlying graph structure, the workflow manager is able to determine all nodes required to be executed along the paths leading to the node the user actually wants to execute.

D. Views and Interactive Brushing

Each Node can have an arbitrary number of views associated with it. Through receiving events from a `HiLiteHandler` (and sending events to it) it is possible to mark (the so-called *HiLiting*) selected points in such a view to enable visual brushing. Views can range from simple table views to more complex views on the underlying data or the generated model.

III. Repository

Knime already offers a large variety of nodes, among them are nodes for various types of data I/O, manipulation, and transformation, as well as data mining and machine learning, and visualization components:
- data I/O: generic file reader, ARFF and Hitlist file reader, database connector, CSV, Hitlist and ARFF writer.
- data manipulation: row and column filtering, data partitioning and sampling, random shuffling or sorting, data joiner and merger,
- data transformation: missing value replacer, matrix transposer, binners, nominal value generators
- mining algorithms: clustering (k-means, sota, fuzzy c-means), decision tree, (fuzzy) rule induction, regression, subgroup and association rule mining.
- machine learning: neural networks (RBF and MLP), support vector machines*, bayes networks and bayes classifier*
- statistics: via integrated R*
- visualization: scatter plot, histogram, parallel coordinates, multidimensional scaling, rule plotters, line and pie charts*
- misc: scripting nodes.

(*: via external libraries or tools).

IV. Extending Knime

Knime already includes new plug-ins to incorporate existing data analysis tools, such as Weka [2], the statistical toolkit R [3], and JFreeChart [4]. It is usually straightforward to create wrappers for external tools without having to modify these executables themselves. Adding new nodes to Knime, also for native new operations, is easy. For this, one needs to extend three abstract classes:
- `NodeModel`: this class is responsible for the main computations. It requires to overwrite three main methods: `configure()`, `execute()`, and `reset()`. The first takes the meta information of the input tables and creates the definition of the output specification. The `execute`-function performs the actual creation of the output data or models, and `reset` discards all intermediate results.
- `NodeDialog`: this class is used to specify the dialog that enables the user to adjust individual settings

that affect the node's execution. A standardized set of `DefaultDialogComponent` objects allows to very quickly create dialogs where only a few standard settings are needed.

- `NodeView`: this class can be overwritten multiple times to allow for different views onto the underlying model. Each view is automatically registered with a `HiLiteHandler` which sends events when other views have hilited points and allows to launch events in case inside this view points have been hilit.

In addition to the three model, dialog, and view classes the programmer also needs to provide a `NodeFactory`, creating new instances. The factory also provides names and other details such as the number of available views or a flag indicating absence or presence of a dialog.

A wizard integrated in the Eclipse-based development environment allows to quickly generate all required class bodies for a new node.

V. Work in Progress

Knime is continuously extended. A few extensions currently being actively under development are described below:

A. Meta Nodes

The ability to wrap a certain sub workflow into an encapsulating node will enable us to assign dedicated servers to this subflow, export it to other users as a predefined module and allow us to create wrappers for repeated execution as needed in cases such as, e.g. cross-validation, bagging and boosting, ensemble learning etc. First prototypes work nicely and we expect to release this feature in the near future. The ability to handle nested workflows will also enable the user to visually design much larger, more complex workflows.

B. Distributed Processing

Due to the modular architecture it is easy to designate specific nodes to be run on separate machines. But to accommodate the increasing availability of multi-core machines, also the support for shared memory parallelism becomes increasingly important. Knime will offer a unified framework to parallelize data-parallel operations as well as the distribution of operations on a cluster or a GRID.

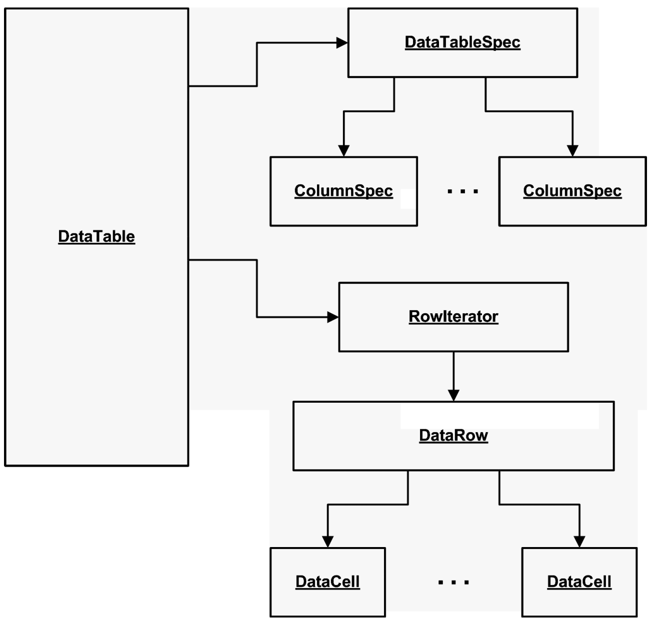

Fig. 3. A UML diagram of the data structure and the main classes it relies on.

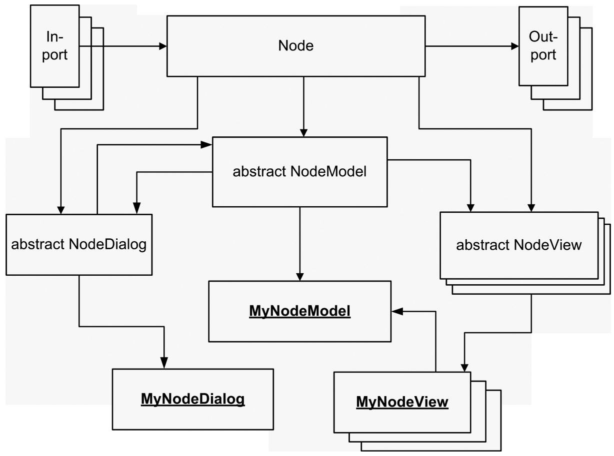

Fig. 4. A UML diagram of the Node and the main classes it relies on.

C. Chem- and Bioinformatics

A number of current projects focus on applications in the Life Sciences. Nodes to process gene expression data and high throughput, high content cell assay images are under development.

D. Webservices

Experimental nodes to access webservices via SOAP have been devised to call computation of chemical properties. Knime itself can also be seen as a potential server for a webservice itself, allowing external users to run predefined workflows.

VI. Acknowledgements

We would like to thank numerous students of Konstanz University for continuous feedback and bug reporting. We thank, in particular, Kilian Thiel and Simona Pintilie for their work on Sota resp. the Parallel Coordinates display.

References

[1] Eclipse Foundation, *Eclipse 3.1 Documentation*, http://www.eclipse.org.
[2] Ian H. Witten and Eibe Frank, *Data Mining: Practical machine learning tools and techniques*, Morgan Kaufmann, San Francisco, 2005, http://www.cs.waikato.ac.nz/ ml/weka/index.html.
[3] R Development Core Team, *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, 2005, http://www.R-project.org.
[4] David Gilbert, *JFreeChart Developer Guide*, Object Refinery Limited, Berkeley, California, 2005, http://www.jfree.org/jfreechart.