EXTERNAL TOOL INTEGRATION WITH PROXY FILTERS IN A DATA MINING APPLICATION FRAMEWORK

Lauri Tuovinen, Perttu Laurinen and Juha Röning
Department of Electrical and Information Engineering, University of Oulu, P.O. Box 4500, FIN-90014, Finland

Keywords: Data mining, Application framework, Pipes and filters, Tool integration, Proxy pattern.

Abstract: An important phase in the development of a data mining application is algorithm selection: for any given data mining task there is likely to be a range of different model types available, as well as a number of different methods for constructing the models. Choosing the one that best accomplishes the task is not trivial and generally involves trials and comparisons of different configurations. It is often convenient to perform the trials on a platform other than the ultimate implementation technology of the application; for example, the application may be implemented in a general-purpose programming language such as C++ while model prototyping is carried out in a scientific computing environment such as MATLAB. For this we propose a data mining application framework that allows MATLAB and other external tools to be integrated into applications via proxies known as gateway filters. To the framework the gateway filters appear no different from algorithms implemented on the framework platform, so it is possible to build a full application prototype early on and then, once the algorithms to be used have been selected, to turn the prototype into a deployable application simply by replacing proxies with natively implemented filters. Thus the framework comprehensively supports the various steps of application programming, from algorithm design and prototype building to final implementation.

1 INTRODUCTION

Various software packages have been created to support the development of data mining applications. Different solutions support the development process in different ways and at different stages. For instance, a graphical tool for building pipe-and-filter style data flows may be useful for rapid prototyping but not for implementing the final product, assuming that a standalone, fully tailored application is desired. A library of data mining algorithms, on the other hand, does not provide a graphical interface but can be included in the final product to speed up the programming phase.

Smart Archive (SA) (Laurinen et al., 2005; Tuovinen et al., 2008) is a software framework for data mining applications implemented in Java and C++. As a traditional application framework it differs in a significant way from both algorithm libraries and graphical application builders. Like a library, it speeds up programming, but instead of providing a vast collection of reusable algorithms it provides a generic kernel into which the algorithms required by a specific application are inserted.

SA can also be used to speed up prototype building, but instead of offering a graphical interface for coupling algorithms it helps the development of the algorithms themselves. A new extension to the framework allows applications to include programs implemented with specialized tools that are more appropriate for the task of algorithm design than general-purpose programming languages like Java. SA thus makes it possible to write an algorithm in, for example, MATLAB, and then to test it immediately in the context of a fully functional application prototype.

Processing algorithms in Smart Archive are known as filters. In a deployed application these are usually written entirely in the implementation language of the framework (Java or C++). However, when an algorithm is implemented using a different tool, it is represented in SA by a proxy instead. The proxy sends the input data to the external tool and receives the results from it while presenting to SA the same interface that all other filters implement. The framework therefore sees no difference between proxies, known as gateway filters, and regular filters.

Since all filters appear similar to the framework, developers can put together a complete application at a relatively early stage—complete in the sense that all of the filters to be included in the application are present as regular filters, proxies or simple dummies.

Tuovinen L., Röning J. and Laurinen P. (2009).
EXTERNAL TOOL INTEGRATION WITH PROXY FILTERS IN A DATA MINING APPLICATION FRAMEWORK.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 249-255
DOI: 10.5220/0002256102490255

The developers can then replace the proxies and dummies with regular filters once the algorithms they represent have been designed, tested and coded in the language of the framework. Some proxies may even be left in the final application, for example if a gateway filter has been used to build an interface to a third-party algorithm library.

This paper proposes gateway filters as a means of supporting algorithm design and prototype implementation in data mining software development. Gateways provide an encapsulated and open-ended interface that allows functionality provided by other tools to be included in SA-based applications. Encapsulation makes using the interface convenient by allowing regular filters to be replaced with gateways and vice versa without modifying other parts of the application. Openness ensures that the range of supported tools is not limited to a few or just one. A positive side effect of openness is that gateway filters have several potential applications besides the principal one.

Section 2 discusses related work, most importantly other data mining frameworks. The main contribution is given in Section 3, which introduces the concept of gateway filters, and Section 4, which presents a gateway filter implementation and documents some tests performed on it. Section 5 discusses the findings, and Section 6 concludes the paper.

2 RELATED WORK

Weka (Witten and Frank, 2005) is a library of machine learning algorithms developed at the University of Waikato, New Zealand. The library provides a collection of algorithms for regression, classification, clustering and association rule mining as well as I/O interfaces and preprocessing, evaluation and visualization methods. The algorithms can be invoked from Java code or operated using a set of graphical user interfaces bundled with the library.

D2K (Automated Learning Group, 2003) is a data mining environment developed by the Automated Learning Group of the University of Illinois. It provides a graphical toolkit for arranging application components, called modules, into processing sequences, called itineraries, but developers also have the option of using the core components of D2K without the toolkit. In addition to a selection of reusable modules for various purposes (I/O, data preparation, computation, visualization) there is an API for programming new ones.

KNIME (Berthold et al., 2006), developed at the University of Konstanz, Germany, is another data mining environment with a graphical user interface for manipulating components, called nodes. Although KNIME and D2K are superficially different, there is a close analogy between the nodes and workflows of KNIME and the modules and itineraries of D2K. Based on the Eclipse integrated development environment, KNIME bears more resemblance to a programming tool and allows arbitrary code snippets to be inserted into workflows.

RapidMiner, formerly known as YALE (Mierswa et al., 2006), was originally developed at the University of Dortmund, Germany. Like D2K and KNIME, it is a graphical environment for building data mining applications from components, called operators. However, unlike D2K and KNIME, RapidMiner models the operator sequence as a tree rather than an arbitrary directed graph. The operator tree can be manipulated either graphically or by editing the XML representation, which is always available in its own tab in the user interface.

Smart Archive shares with Weka, D2K, KNIME and RapidMiner the principle of composing applications from components, with the output of one component becoming input for the next one. The sequence of components, analogous to a D2K itinerary, is called the execution graph. Another layer of functionality built upon the execution graph, called the mining kernel, is responsible for gathering data from data sources (via interface objects known as input receivers) and feeding it into the execution graph.

A general difference between Smart Archive and the other frameworks discussed above is the greater emphasis that SA puts on openness and flexibility. In practice this means that SA offers greater freedom to work directly with application code and to use selected parts of the framework to build a precisely tailored application. Another significant difference is found in the approach to using databases in applications: Smart Archive, with its data sinks embedded in components, provides a tighter form of integration than the other frameworks, where database inputs and outputs are just types of components among others.

From a software engineering perspective, the integration of SA with external tools falls into the scope of software interoperability, on which there is a considerable body of literature. Since SA is a component-based framework, research on component-based systems is of particular interest. Broadly speaking, SA belongs to the category of open systems (Oberndorf, 1997) and this is the feature that allows external tools to be connected to it. The concrete tool integration approach of SA is similar to that of the generic architecture presented in (Grundy et al., 1998): the external tools are wrapped in component-based interfaces.

Although the idea of gateway filters is not new,

applying it to tool integration does not appear to have spread to data mining frameworks. The other frameworks do provide components that could conceivably be used to implement something like the `RDBGateway` filter described in Section 4, but the extra effort required would be considerable and the result would be awkward compared to the case where all the details of exchanging data and instructions with the external tool are taken care of by a single component. In SA applications this connection can be established simply by instantiating a proxy and setting its parameters.

3 THE GATEWAY FILTER CONCEPT

Filters are the core elements of any Smart Archive application. Each filter implements a transformation of data, usually reducing its volume and increasing its degree of abstraction. The desired knowledge is extracted from raw data through a progression of transformations. Filters are packaged in components, which may also contain a data sink for storing the transformed data produced by the filter.

Each filter has one or more named input channels through which it receives data to be processed. An input channel, in turn, has one or more input variables, which contain the actual data values. Similarly, each filter has one or more output channels. A channel may be defined as a cohesive variable group that plays a distinct role in the operation carried out by the filter. When a filter is joined with a pipe to another application element, the pipe is always attached to a particular input or output channel.

Although a filter conceptually represents a data transformation, it is not necessary for it to implement the transformation. Instead, the task of executing the transformation may be delegated to another application by employing a gateway filter. Gateway filters are derived from an abstract class that combines two Gang-of-Four design patterns (Gamma et al., 1994): proxy and template method. The proxy pattern separates the implementation of the filter algorithm from the filter object so that the implementation may reside in a different environment while to the framework it appears as if the proxy itself were implementing the algorithm. Within the proxy, the template method defines the steps of relaying inputs to the implementation and receiving outputs from it.

Smart Archive filter classes are usually derived from the class `AbstractFilter`. This class declares an abstract method, `executeTransformation`, which constitutes the core of the filter, transforming inputs into outputs. Regular filters override this method with a concrete implementation of the filter algorithm. With gateway filters, however, an additional layer of abstraction is added.

The base class of all gateway filters is `ExternalToolGateway`, a direct subclass of `AbstractFilter`. `ExternalToolGateway` overrides `executeTransformation` with a template method, the essential parts of which are shown in Figure 1. In the body of the template method there are calls to four subroutines, each of which is an abstract method in `ExternalToolGateway`. Thus, in order to create a gateway filter, a programmer must create a subclass of `ExternalToolGateway` that overrides these four methods with concrete implementations of the corresponding steps of the template algorithm.

Each of the four abstract methods returns a `boolean` value where `true` indicates that the operation was successfully completed, in which case the algorithm proceeds to the next step. In the case of a `false` return value the algorithm terminates instantly and `executeTransformation` returns a `null` output object. In any event a fifth method, `cleanUp`, will be called before returning, allowing the filter to reset any states that the other four might have modified.

The gateway filter can be thought of as a client that requests services from a server residing on the concrete implementation platform. The first subroutine call in the template method, `sendInputToExternal`, delivers a batch of input data to the server. The second call, `triggerExternal`, causes the server to forward the data to an implementation the filter algorithm. The third call, `waitForResponse`, blocks the client until the server announces either that it is ready to send output or that the operation failed. In the former case, the fourth call, `getOutputFromExternal`, pulls in the output data generated by the server. This process is illustrated in Figure 2.

It is worth noting that the template method assumes that it is not possible for the client to simply make a single function call to the server, with the input data as argument and the output data as return value. This choice was made because it places fewer restrictions on the implementation of the connection between client and server. The next section presents a gateway filter implementation where asynchronous client-server communication is the only option.

4 IMPLEMENTATION AND TESTING

There are many different ways to implement a gateway filter, distinguished mainly by the means of client-server communication. Perhaps the most

straightforward option would be to use remote method invocation, but this option is viable only with server platforms capable of running a Java virtual machine. Instead, a more open-ended approach was adopted in the implementation of the first gateway filter: indirect communication via a repository.

Because SA powerfully supports interaction with relational databases (RDBs), they were chosen as the repository technology to be used with the gateway filter. Thus, a subclass of `ExternalToolGateway` was written that writes the filter inputs into a set of RDB tables and reads the outputs from another set of tables. The new class was named `RDBGateway`.

Figure 3 shows the passage of signals and data when `RDBGateway` is used. The means of passing signals is a status table, which both the client and the server can access. There are four states that occur in sequence when the filter is executed: 'idle' when the filter is inactive, 'input' when the client is passing input to the server, 'process' when the server may begin processing the data, and 'output' when the server has finished and the client may read the output. Alternatively, the server may set the state to 'abort' in the case of an error that prohibits it from generating output. In either case the filter will then go back to the idle state and remain there until it is invoked again.

The server, while not executing an algorithm, stays in a loop where it polls the status table for the trigger signal (state change to 'process'). Besides the state, the status table contains information on where to find the input data, which algorithm to invoke and where to write the outputs. While the server is processing, the client waits in a loop, polling the status table for the ready signal (state change to 'output'). When the signal is received, the outputs are available in tables specified to the server via the status table.

An implementation of the server has been written for the scientific computing environment MATLAB.

```
function executeTransformation(inputs) {

   if (sendInputToExternal(inputs)) {
      if (triggerExternal()) {
         success = waitForResponse();
         if (success) {
            outputs = getOutputFromExternal();
         } else {
            outputs = null;
         }
      }
   }
   cleanUp();

   return outputs;

}
```

Figure 1: The template method that implements `executeTransformation` in the `ExternalToolGateway` class.

To test the connection to MATLAB, an application that first trains and then applies a k nearest neighbors (kNN) classifier was devised. The data used was synthetic and randomly generated. The classifier was implemented as a MATLAB function and an `RDBGateway` was used to represent it in the application. The same algorithm was also implemented in Java as a regular filter so that the performance overhead incurred by the gateway could be estimated.

The classifier was tested with datasets of different sizes. The results of the tests are documented in Tables 1 and 2. These figures compare poorly to the performance of the Java implementation, for which the training times were consistently zero within the accuracy of the timer and the classification task was completed in less than 100 milliseconds even in the $N_T = N_C = 500$ case. However, the performance of the MATLAB implementation comprises several elements, not all of which depend on the gateway. Its effect on classification performance can be estimated by looking at the training performance figures.

Since training a kNN classifier involves simply storing the training set, the training times give a fair idea of how long it takes to transfer a dataset between the framework and an external tool. The figures imply that in the tests performed, the time to transfer inputs to the classifier and the results back to the gateway was a small fraction of the time it took to process the inputs. In an application required to respond rapidly to new data this overhead could be a problem, but in an application that takes minutes to process a single dataset, an overhead of seconds should be acceptable.

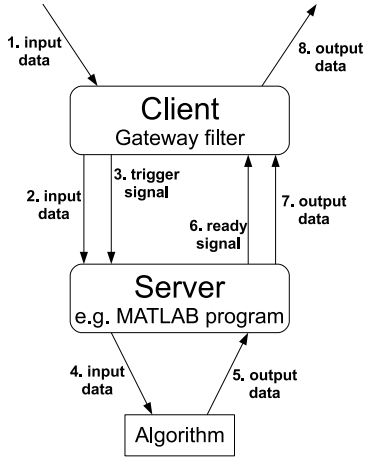

Figure 2: The operating principle of gateway filters. The filter receives a batch of data (1), which it forwards to the external tool (2). The filter then triggers the external tool (3), which invokes the appropriate algorithm (4). Once the algorithm has returned its output (5), the external tool sends a ready signal (6) to the filter, which accepts the output (7) and forwards it to the framework (8).

It should be noted that this example is not intended as a realistic demonstration of how `RDBGateway` might be used. A library implementation would be a more likely choice if one were to use kNN classification in an SA application. Also, the number of variables in the data used for the tests was considerably smaller than what one would normally encounter in a data mining task, so the measured processing times do not accurately reflect the performance of the tested implementations in a real-world situation.

Table 1: Training performance of the kNN classifier, in milliseconds. N_T denotes the size of the training dataset. The times are averaged over three test runs.

N_T		
100	**300**	**500**
280 ms	590 ms	1060 ms

5 DISCUSSION

The chief purpose of the gateway filter concept is to support connections to external computing tools, primarily for building functional application prototypes. It is assumed here that developing the application involves designing new algorithms, not simply finding suitable parameters for off-the-shelf ones. This is more laborious than using library algorithms, but on the other hand it gives developers more freedom and leverage, which is characteristic of the SA approach.

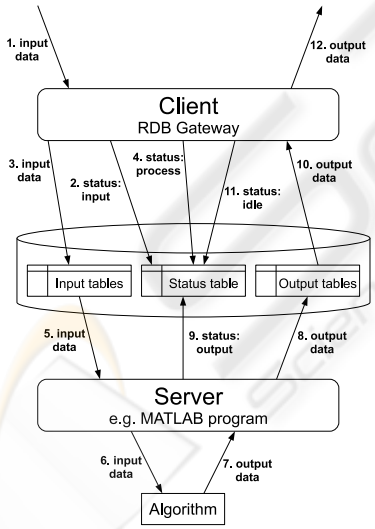

Figure 3: The operating principle of the RDB gateway filter. Upon receiving a batch of data (1), the filter first sets the status in the status table to 'input' (2) and then writes the data into the designated input tables (3). The status is then changed to 'process' (4), which triggers the external tool to read the input data (5) and invoke the algorithm (6). When the algorithm returns (7), the tool writes its output into the designated output tables (8) and changes the status to 'output' (9), signalling to the filter that it may read the data (10). Finally, the filter resets the status to 'idle' (11) and forwards the output data to the framework (12).

Experience from various research projects suggests that algorithm developers, especially ones with a background in mathematics, prefer to work with scientific computing packages such as MATLAB or R rather than directly with the implementation language of the application to be created. Research is also the context where the need to design custom algorithms for a data mining application is most likely to arise. This makes SA particularly useful for creating new experimental results and transferring them into production use; this capacity of SA has already been demonstrated (Tuovinen et al., 2008).

The MATLAB test shows that the connection from SA to external tools through a proxy works and does not cause an unreasonable transmission overhead compared to the time it takes to process a moderately large dataset. It is therefore possible to develop an algorithm in MATLAB and include it in a functional prototype of an SA-based data mining application. Since the function implementing the algorithm is completely separated from the proxy representing it in SA and since the MATLAB programming language is interpreted, one can make changes to the algorithm without recompiling anything, even while the main Java program is running. This makes it convenient to try different variants and configurations of the algorithm at the developer's will.

As required, the interface between the proxy and the external tool is completely encapsulated by the gateway filter, which limits the scope of the changes required to remove the dependency on the external tool: the only differences between the two application configurations in the kNN example are in the code for setting up the filter for the classifier component. This code accounts for less than ten percent of the code lines in the main routine of the application. Since the Java implementation of the classifier takes the same inputs and gives the same outputs as the MATLAB implementation, they are fully interchangeable.

The other major requirement, openness, is also

Table 2: Classification performance of the kNN classifier, in seconds. N_T denotes the size of the training dataset and N_C the size of the classification dataset. Each time is the result of a single test run.

		N_T		
		100	**300**	**500**
N_C	**100**	2.3 s	5.7 s	9.3 s
	300	6.2 s	18.3 s	29.7 s
	500	11.0 s	29.8 s	49.2 s

fulfilled because `RDBGateway` can be used to interface with any computing tool from which it is possible to access a relational database. Since RBD technology is firmly established and widely employed, this is a fairly loose constraint: any tool designed to be used for processing structured data is likely to have the option of importing the data from a database, either as a built-in function or via some kind of add-on. The external tool could even be one of the other frameworks reviewed in Section 2, configured to run an application that reads its original inputs from the input tables of `RDBGateway` and writes its final outputs into the output tables. It would also be possible to use the MATLAB server with another framework, since it does not depend on SA; however, this would still be more awkward than using SA since a suitable client component would first have to be implemented.

If none of the available gateway classes are satisfactory, it is possible to derive a new one from `ExternalToolGateway`, which places no restrictions on the means of communication with the external tool. The solution is therefore genuinely open-ended, but openness also has its disadvantages. To keep the set of possible external tools as inclusive as possible, the implementation of the gateway concept makes deliberately few assumptions about the server side, but this also means more work for a developer implementing a new server because it limits what the framework can do for the developer. On the other hand, the absence of many restricting assumptions makes gateway filters potentially suitable for a variety of purposes that were never considered as primary design goals. We discuss some notable possibilities below.

Before an application is deployed, gateway filters used in the prototype will probably need to be replaced with regular filters: it is likely that some of the tools used by the algorithm developers are not available in the production environment, and in any event the overhead from communicating with them would be undesirable. However, a gateway could also be used as an interface to a free algorithm library, in which case the overhead would probably be acceptable and the library could be distributed with the application. For example, a gateway that can serve as a proxy for any Weka algorithm would significantly increase the effective number of reusable filters available. Developers could then focus on designing those filters that can not be implemented simply by picking the right library algorithm.

Another purpose that gateway filters could serve in the finished application is distributed computing. This would involve creating a gateway that can invoke a function residing on a remote server; unlike an external tool, the remote code could work directly with the data structures used by SA for internal communication. In an application that includes two or more computationally heavy components that can be executed in parallel, gateway filters could be used to distribute the workload of these components. Alternatively, one could write a gateway that divides the batch of input data into work units and sends each unit to a different remote machine to be processed. It is worth noting that SA has already been used to implement a distributed application, but in this case it was entire SA instances that were executed in parallel, not individual components (Tuovinen et al., 2008).

Finally, since it makes no difference to the framework how the external tool generates its results, a gateway could be used to employ a human expert in lieu of a filter algorithm. In this case the external tool would be an interactive application, not just a passive platform for algorithm execution. The role of the framework would be to prepare data for the application and possibly to process its output further. For example, one might want to use a tool like SignalPlayer (Siirtola et al., 2007) for exploratory analysis of time series data when the problem at hand is not understood well enough to allow a non-interactive solution.

The principal problem identified in the current `RDBGateway` implementation is that although inputs and outputs are automatically transferred between the proxy and the external tool, there is no similar mechanism for transferring algorithm parameters. Therefore, when working on an application containing an external algorithm, a developer has to switch from the main application to the external tool to change the parameters of the algorithm. This has proven somewhat inconvenient, so it would be a significant improvement if proxy filters could mediate parameter values to externally implemented algorithms.

6 CONCLUSIONS

In this paper we described a solution to the problem of including algorithms implemented with external tools in data mining applications built using a software framework. The framework, called Smart Archive, is an object-oriented application framework based on the pipes-and-filters architectural style. Smart Archive applications can interface with external tools through special filters known as gateways. A gateway filter serves as a proxy that represents an externally implemented algorithm to the framework and transfers data between the framework and the external tool.

A typical example of an external tool that one might want to use in conjunction with Smart Archive is the scientific computing environment MATLAB.

The proxy filter concept was demonstrated by establishing a gateway to MATLAB and using it to perform a classification task on synthetic data. The performance of the classifier was compared to that of the same classification algorithm implemented as a regular Smart Archive filter. The performance of the MATLAB version proved substantially worse, but the effect of data transfer overhead caused by the gateway was found to be comparably small.

To the framework, a gateway filter appears no different from any other kind of filter, so it can be conveniently replaced with a filter implemented in the language of the framework once the details of the algorithm have been settled. In the classification test, for instance, replacing the MATLAB gateway with a regular filter could be accomplished with minor and bounded modifications to the application code. Another desirable feature of the gateway mechanism is that it is highly open-ended: the gateway used to communicate with MATLAB could be used with many other tools as well, and developers are also free to create their own gateway implementations to suit their particular purposes.

Proxy filters were introduced into Smart Archive primarily to allow developers of data mining applications to build a functional application prototype from algorithms created with specialized tools that are better suited to such work than general-purpose programming languages. This ability was demonstrated by the MATLAB test. Other possible uses of proxies include interfacing with algorithm libraries and developing distributed applications. There could also be a gateway to an external tool operated interactively by a human expert, who would effectively assume the role of a filter by using the tool to produce the output to be sent back to the framework.

ACKNOWLEDGEMENTS

The authors would like to thank the Finnish Funding Agency for Technology and Innovation (http://www.tekes.fi), Rautaruukki (http://www.ruukki.com) and Polar Electro (http://www.polar.fi) for funding the research on Smart Archive in the SAMURAI project. L. Tuovinen wishes to thank the Graduate School in Electronics, Telecommunications and Automation (http://signal.hut.fi/geta/) and the Tauno Tönning Foundation (http://www.tonninginsaatio.fi) for funding his postgraduate work.

REFERENCES

Automated Learning Group (2003). *D2K Toolkit User Manual*. Technical manual, available at http://alg.ncsa.uiuc.edu.

Berthold, M. R., Cebron, N., Dill, F., di Fatta, G., Gabriel, T. R., Georg, F., Meinl, T., Ohl, P., Sieb, C., and Wiswedel, B. (2006). KNIME: The Konstanz information miner. In *Proceedings of the 4th Annual Industrial Simulation Conference, Workshop on Multi-Agent Systems and Simulation*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Grundy, J., Mugridge, R., Hosking, J., and Apperley, M. (1998). Tool integration, collaboration and user interaction issues in component-based software architectures. In *Proceedings of TOOLS 28: Technology of Object-Oriented Languages and Systems*, pages 299–312.

Laurinen, P., Tuovinen, L., and Röning, J. (2005). Smart Archive: a component-based data mining application framework. In *Proceedings of the Fifth International Conference on Intelligent Systems Design and Applications (ISDA 2005)*, pages 20–25.

Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., and Euler, T. (2006). YALE: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 935–940.

Oberndorf, P. A. (1997). Facilitating component-based software engineering: COTS and open systems. In *Proceedings of the Fifth International Symposium on Assessment of Software Tools and Techniques*, pages 143–148.

Siirtola, P., Laurinen, P., and Röning, J. (2007). SignalPlayer - a time series visualization system. In *Proceedings of the Finnish Signal Processing Symposium (FINSIG 2007)*.

Tuovinen, L., Laurinen, P., Juutilainen, I., and Röning, J. (2008). Data mining applications for diverse industrial application domains with Smart Archive. In *Proceedings of the IASTED International Conference on Software Engineering (SE 2008)*, pages 56–61.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*. San Francisco: Morgan Kaufmann, 2nd edition.