A Toolbox Approach to Flexible and Efficient Data Mining

Ole M. Nielsen*[1], Peter Christen[1], Markus Hegland[1], Tatiana Semenova[1], and Timothy Hancock[2]

[1] Australian National University, Canberra, ACT 0200, Australia
URL: `http://csl.anu.edu.au/ml/dm/`
[2] James Cook University, Townsville, QLD 4811, Australia

Abstract. This paper describes a flexible and efficient toolbox based on the scripting language Python, capable of handling common tasks in data mining. Using either a relational database or flat files the toolbox gives the user a uniform view of a data collection. Two core features of the toolbox are caching of database queries and parallelism within a collection of independent queries. Our toolbox provides a number of routines for basic data mining tasks on top of which the user can add more functions – mainly domain and data collection dependent – for complex and time consuming data mining tasks.

Keywords: Python, Relational Database, SQL, Caching, Health Data

1 Introduction

Due to the availability of cheap disk space and automatic data collection mechanisms huge amounts of data in the Terabyte range are becoming common in business and science [7]. Examples include the customer databases of health and car insurance companies, financial and business transactions, chemistry and bioinformatics databases, and remote sensing data sets. Besides being used to assist in daily transactions, such data may also contain a wealth of information which traditionally has been gathered independently at great expense. The aim of data mining is to extract useful information out of such large data collections [3].

There is much ongoing research in sophisticated algorithms for data mining purposes. Examples include predictive modelling, genetic algorithms, neural networks, decision trees, association rules, and many more. However, it is generally accepted that it is not possible to apply such algorithms without careful data understanding and preparation, which may often dominate the actual data mining activity [5, 11]. It is also rarely feasible to use off-the-shelf data mining software and expect useful results without a substantial amount of data insight. In addition, data miners working as consultants are often presented with data

* Corresponding author, E-Mail: Ole.Nielsen@anu.edu.au

sets from an unfamiliar domain and need to get a good feel for the data and the domain prior to any "real" data mining. The ease of initial data exploration and preprocessing may well hold the key to successful data mining results later in a project.

Using a portable, flexible, and easy to use toolbox can not only facilitate the data exploration phase of a data mining project, it can also help to unify data access through a middleware library and integrate different data mining applications through a common interface. Thus it forms the framework for the application of a suite of more sophisticated data mining algorithms. This paper describes the design, implementation, and application of such a toolbox in real-life data mining consultancies.

1.1 Requirements of a Data Mining Toolbox

It has been suggested that the size of databases in an average company doubles every 18 months [2] which is akin to the growth of hardware performance according to Moore's law. Yet, results from a data mining process should be readily available if one wants to use them to steer a business in a timely fashion. Consequently, *data mining software has to be able to handle large amounts of data efficiently and fast.*

On the other hand, data mining is as much an art as a science, and real-life data mining activities involve a great deal of experimentation and exploration of the data. Often one wants to "let the data speak for itself". In these cases one needs to conduct experiments where each outcome leads to new ideas and questions which in turn require more experiments. Therefore, it is mandatory that *data mining software facilitates easy querying of the data.*

Furthermore, data comes in many disguises and different formats. Examples are databases, variants of text files, compact but possibly non-portable binary formats, computed results, data downloaded from the Web and so forth. Data will usually change over time – both with respect to content and representation – as will the demands of the data miner. It is desirable to be able to access and combine all these variants uniformly. *Data mining software should therefore be as flexible as possible.*

Finally, data mining is often carried out by a group of collaborating researchers working on different aspects of the same dataset. A suitable software library providing shared facilities for access and execution of common operations leads to safer, more robust and more efficient code because the modules are tested first by the developer and then later by the group. A shared toolbox also tends to evolve towards efficiency because the best ideas and most useful routines will be chosen among all tools developed by the group.

This paper describes such a toolbox – called *DMtools* – developed by and aimed at a small data mining research group for *fast*, *easy*, and *flexible* access to large amounts of data.

The toolbox is currently under development and a predecessor has successfully been applied in health data mining projects under the ACSys CRC[1]. It assists our research group in all stages of data mining projects, starting from data preprocessing, analysis and simple summaries up to visualisation and report generation.

2 Related Work

Database and data mining research are two overlapping fields and there are many publications dealing with their intersection. An overview of database mining is given in [6]. According to the authors the efficient handling of data stored in relational databases is crucial because most available data is in a relational form. Scalable and efficient algorithms are one of the challenges, as is the development of high-level query languages and user interfaces. Another key requirement is interactivity.

A classification of frameworks for integrating data mining applications and database systems is presented in [4]. Three classes are presented: (1) Conventional – also called loosely coupled – where there is no integration between the database system and the data mining applications. Data is read tuple by tuple from a database, which is very time consuming. The advantage of this method is that any application previously running on data stored in a file system can easily be changed, but the disadvantage is that no database facilities like optimised data access or parallelism are used. (2) In the tightly coupled class data intensive and time-consuming operations are mapped to appropriate SQL queries and executed by the database management system. All applications that use SQL extensions or propose such extensions to improve data mining algorithms are within this class. (3) In the black box approach complete data mining algorithms are integrated into the database system. The main disadvantage of such an approach is its lack of flexibility. Following this classification, our *DMtools* belong to the tightly-coupled approach, as we generate simple SQL queries and retrieve the results for further processing in the toolbox. As the results are often aggregated data or statistical summaries, communication between the database and data mining contexts can be reduced significantly.

Several research papers address data mining based on SQL databases and propose extensions to the SQL standard to simplify data mining and make it more efficient. In [8] the authors propose a new SQL operator that enables efficient extraction of statistical information which is required for several classification algorithms. The problem of mining general association rules and sequential patterns with SQL queries is addressed in [13], where it is shown that it is possible to express complex mining computations using standard SQL. Our data mining toolbox is currently based on relational databases, but can also integrate

[1] ACSys CRC stands for 'Advanced Computational Systems Collaborative Research Centre' and the data mining consultancies were conducted at the Australian National University (ANU) in collaboration with the Commonwealth Scientific and Industrial Research Organisation (CSIRO).

flat files. No SQL extension is needed, instead we put a layer on top of SQL where most of the "intelligent" data processing is done. Database queries are cached to improve performance and re-usability.

Other toolbox approaches to data analysis include the IDEA (Interactive Data Exploration and Analysis) system [12], where the authors identify five general user requirements for data exploration: Querying (the selection of a subset of data according to the values of one or more attributes), segmenting (splitting the data into non-overlapping sub-sets), summary information (like counts or averages), integration of external tools and applications, and history mechanisms. The IDEA framework allows quick data analysis on a sampled sub-set of the data with the possibility to re-run the same analysis later on the complete data set. IDEA runs on a PC, with the user interacting on a graphical interface.

Yet another approach used in the Control [9] project is to trade quality and accuracy for interactive response times, in a way that the system quickly returns a rough approximation of a result that is refined continuously. The user can therefore get a glimpse at the final result very quickly and use this information to change the ongoing process. The Control system, among others, includes tools for interactive data aggregation, visualisation and data mining.

An object-oriented framework for data mining is presented in [14]. The described Data Miner's Arcade provides a collection of APIs for data access, plug'n'play type tool integration with graphical user interfaces, and for communication of results. Access to analysis tools is provided without requiring the user to become proficient with the different user interfaces. The framework is implemented in Java.

3 Choice of Software

The *DMtools* are based on the scripting language *Python* [1], an excellent tool for rapid code development that meets all of the requirements listed in Section 1.1 very well. Python handles large amounts of data efficiently, it is very easy to write scripts as well as general functions, it can be run interactively (interpretable) and it is flexible with regards to data types because it is based on general lists and dictionaries (associative arrays), of which the latter are implemented as very efficient hash-tables. Functions and routines can be used as templates which can be changed and extended as needed by the user to do more customised analysis tasks. Having a new data exploration idea in mind the data miner can implement a rapid prototype very easily by writing a script using the functions provided by our toolbox.

Databases using SQL are a standardised tool for storing and accessing transactional data in a safe and well-defined manner. The *DMtools* are accessing a relational database using the Python database API [2]. Currently, we are using MySQL [15] for the underlying database engine, but modules for other database servers are available as well. Both MySQL and Python are freely available, li-

[2] Available from the Python homepage at `http://www.python.org/topics/database/`

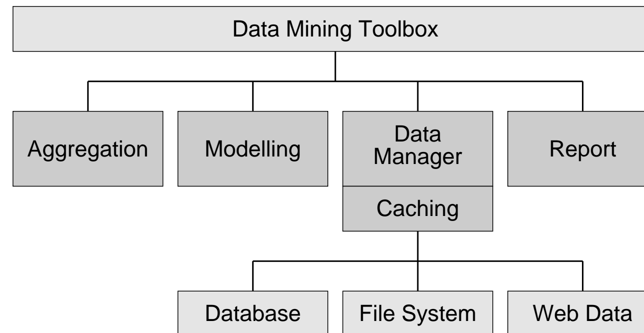

Fig. 1. Architecture of DMtools

censed as free software and enjoy very good support from a large user community. In addition, both products are very efficient and robust.

3.1 Toolbox Architecture

In our toolbox the ease of SQL queries and the safety of relational databases are combined with the efficiency of flat file access and the flexibility of object-oriented programming languages in an architecture as shown in Figure 1. Based on relational databases, flat files, the Web, or any other data source a *Data Manager* deals with retrieval, caching and storage of data. It provides routines to execute an arbitrary SQL query and to read and write binary and text files. The two important core components of this layer are its transparent caching mechanism and its parallel database interface which intercepts SQL queries and parallelises them on-the-fly. The *Aggregation* module implements a library of Python routines taking care of simple data exploration, statistical computations, and aggregation of raw data. The *Modelling* module contains functions for parallel predictive modelling, clustering, and generation of association rules. Finally, the *Report* module provides visualisation and allows facilities for simple automatic report generation.

Functions defined in the toolbox layer are designed to deal with issues specifically for a given data mining project, which means they use knowledge about a given database structure and return customised results and plots. This layer contains routines that are not available in standard data analysis or data mining packages.

Example 1. Dictionary of Mental Health Medications
A central object within the domain of health statistics is a *cohort*, defined here as a Python dictionary of *entities* like customers or patients fulfilling a given criterion. As one task in a data mining project might be the analysis of a group of entities (e.g. all patients taking certain medication), one can use the function `get_cohort` to extract such a cohort once and cache the resulting dictionary so it is readily available for subsequent invocations. Being interested in mental

health patients, one might define a dictionary like the one shown below and use it to get a cohort.

```
mental_drugs = {'Depression': [32654, 54306, 12005, 33421],
                'Anxiety': [10249, 66241],
                'Schizophrenia': [99641, 96044, 39561]}

depressed = get_cohort(mental_drugs['Depression'],1998)
```

Several kinds of analyses can be performed using a cohort as a starting point. For example the function `plot_age_gender(depressed)` provides barplots of the given cohort with respect to age groups and gender incorporating denominator data if available. Another function `list_drug_usage(depressed)` gives a list of all medication prescribed to patients in the given cohort. This list includes description of the drug, number of patients using it, the total number of prescriptions and the total cost of each drug. Routines from all modules can either be used interactively or added to other Python scripts to build more complex analysis tasks.

4 Caching

Caching of function results is a core technology used throughout *DMtools* in order to render the database approach feasible. We have developed a methodology for *supervised* caching of function results as opposed to the more common (and also very useful) *automatic* disk caching provided by most operating systems and Web browsers.

Like automatic disk caching, supervised caching trades space for time, but the approach we use is one where time consuming operations such as database queries or complex functions are intercepted, evaluated and the resulting objects are made persistent for rapid retrieval at a later stage. We have observed that many of these time consuming functions tend to be called repetitively with the same arguments. Thus, instead of computing them every time, the cached results are returned when available, leading to substantial time savings. The repetitiveness is even more pronounced when the toolbox cache is shared among many users, a feature we use extensively. This type of caching is particularly useful for computationally intensive functions with few frequently used combinations of input arguments. Supervised caching is invoked in the toolbox by explicitly applying it to chosen functions. Given a Python function of the form `T = func(arg1,...,argn)` caching in its simplest form is invoked by replacing the function call with `T = cache(func,(arg1,...,argn))`.

Example 2. Function Caching
Caching of a simple SQL query using the toolbox function `execquery` can be done as follows:

```
database = 'CustomerData'
query = 'select distinct CustomerID, count(*) from %s;' %database
customer_list = cache(execquery,(query))
```

Table 1. Function Caching Statistics

Function Name	Hits	Time (sec)		Gain(%)	Size (MB)
		Exec	Cache		
execquery	4,149	130	6	91.43	4.53
get_mbs_patients	172	1,281	76	93.92	48.53
get_selected_transactions	420	1,507	5	99.33	6.67
multiquery	46	133	0	99.69	0.76
simplequery	5	50	0	99.86	0.08
get_cohort	168	489	0	99.92	0.20
get_drug_usage	95	1,388	0	99.99	0.02

The user can take advantage of this caching technique by applying it to arbitrary Python functions. However, this technique has already been employed extensively in the *Data Manager* module so using the high level toolbox routines will utilise caching completely transparently with no user intervention – the caching supervision has been done in the toolbox design. For example, most of the SQL queries that are automatically generated by the toolbox are cached in this fashion. Generating queries automatically increases the chance of cache hits as opposed to queries written by the end user because of their inherent uniformity. In addition to this, caching can be used in code development for quick retrieval of precomputed results. For example if a result is obtained by automatically crawling the Web and parsing HTML or XML pages, caching will help in retrieving the same information later – even if the Web server is unserviceable at that point.

The function `get_cohort` used in a particular project required on the average 489 seconds worth of CPU time on a Sun Enterprise server and the result took up about 200 Kilobytes of memory. Subsequent loading takes 0.22 seconds – more than 2,000 times faster than the computing time. This particular function was hit 168 times in a fortnight saving four users a total of 23 hours of waiting. Table 1 shows some caching statistics from a real-life data mining consultancy in health services obtained from four users over two weeks. The table has one entry for each Python function that was cached. The second column shows how many times a particular instance of that function was hit, i.e. how many times results were retrieved from the cache rather than being computed. The third column shows the average CPU time which was required by instances of each function when they were originally executed, and the fourth column shows the average time it took to retrieve cached results for each function. The fifth column then shows the average percentile gain $((Exec - Cache)/Exec * 100)$ achieved by caching instances of each function, and the sixth column shows the average size of the cached results for each function. The table is sorted by average gain.

If the function definition changes after a result has been cached or if the result depends on other files wrong results may occur when using caching in it simplest form. The caching utility therefore supports specification of explicit dependencies in the form of a file list, which, if modified, triggers a recomputation.

Other options include forced recomputation of functions, statistics regarding time savings, sharing of cached results, clean-up routines and data compression. Note that if the inputs or outputs are very large, caching might not save time because disk access may dominate the execution time. This is due to overheads consisting mainly of input checks, hashing and comparisons of argument list, as well as writing and reading cache files. If caching does not lead to any time savings, a warning is given. Very large datasets are dealt with through blocking into manageable chunks and separate caching of these.

Example 3. Caching of XML Documents
Supervised caching is used extensively in the toolbox for database querying but is by no means restricted to this. Caching has proven to be useful in other aspects of the data mining toolbox as well. An example is a Web application built on top of the toolbox which allows managers to explore and rank branches according to one or more user-defined features such as *Annual revenue*, *Number of customers serviced relative to total population*, or *Average sales per customer*. The underlying data is historical sales transaction data which is updated monthly, so features need only be computed once for new data when it is added to the database. Because the data is static, cached features are never recomputed and the application can therefore make heavy use of the cached database queries. Moreover, no matter how complicated a feature is, it can be retrieved as quickly as any other feature once it has been cached. In addition, the Web application is configured through an XML document defining the data model and describing how to compute the features. The XML document must be read by the toolbox, parsed and converted into appropriate Python structures prior to any computations. Because response time is paramount in an interactive application, parsing and interpretation of XML is prohibitive, but by using the caching module, the resulting Python structures are cached and retrieved quickly enough for the interactive application. The caching function was made dependent on the XML file itself, so that all structures are recomputed whenever the XML file has been edited – for example to modify an existing feature-definition, add a new month, or change the data description. Below is a code snippet from the Web application. The XML configuration file is assumed to reside in `sales.xml`. The parser which builds Python structures from XML is called `parse_config` and it takes the XML filename as input. To cache this function, instead of the call `(feature_list, branch_list) = parse_config(filename)` we write:

```
filename = 'sales.xml'
(feature_list, branch_list) = cache(parse_config, (filename),
                                    dependencies = filename)
```

5 Database Access and Parallelism

The toolbox provides powerful and easy-to-use access to an SQL database using the Python database API. We are using MySQL but any SQL database known to Python will do. In its simplest form it allows execution of any valid SQL

query. If a *list* of queries is given, they are executed in parallel by the database server if a multiprocessor architecture is available and the results are returned in a list.

The achievable speedup of this procedure will naturally depend on factors such as the amount of *communication* and the *load balancing*. For large results communication time will dominate the execution time thus reducing the speedup. In addition, the total execution time of a parallel query is limited by the slowest query in the list, so if the queries are very different in their complexity, speedup will only be modest. However, a well balanced parallel query where results are of a reasonable size can make very good use of a parallel architecture. For example, executing a parallel query over five tables of size varying from 250 thousand to 13 million transactions took 2,280 seconds when run sequentially and 843 seconds when run in parallel. This translates into a speedup of 2.7 on five processors or an parallel efficiency of 0.54.

The database interface makes use of supervised caching technology and caches the results of queries as described in the previous section. This can be enabled or disabled through a keyword argument in the function `execquery`. The data_manager module also contains a number of functions to perform standard queries across several tables. One example is the function `standardquery` which takes as input two attribute names, A_1 and A_2, a list of (conforming) database tables, and an optional list of criteria to impose simple restrictions on the query. The function returns all occurrences of attribute A_2 for each distinct value of A_1 from all tables where all the given criteria are met (using conjunction). For example, the call

```
standardquery('Company', 'Customer', tables,
              [('Year', 1997), ('Qtr', 1)])
```

yields a count of customers for each company in the first quarter of 1997. Another example is the function `joinquery` which improves the performance of of normal SQL joins. It takes as arguments a list of fields, a list of tables, a list of joins of the form '`table1_name.field = table2_name.field`' and a list of conditions, and returns a dictionary of results.

6 Applications

To illustrate the application of the *DMtools* we give two examples designed and used for a health services data mining consultancy. The data collection we had to our disposal consisted of two tables, one containing medication prescriptions and the other containing doctor consultancies by patients. In addition, we had specialty information about doctors and geographical information that associated each post code with one of seven larger area codes (like capital, metropolitan or rural). All patient and doctor identifiers were coded to protect individual privacy. Finally, we had data describing different drugs and different treatments obtained from the Web.

Example 4. Doctors Prescription Behaviour
In this example, we describe how we analysed prescription patterns of specialist doctors. The aim was to find unusual behaviour such as over-prescribing. In particular, we wanted to know for each specialist how many patients he or she serviced and how many of these were prescribed a particular type of medication.

For this task, we linked medication prescription information with patient information for a user given doctor specialty. The domain specific function

```
get_doctor_behaviour(items, years, specialty_code)
```

takes as input a list of drug code items, a list of years (as we are interested in the temporal changes in prescription behaviour over a time period) and a doctor specialty code.

The toolbox is used as follows: First, the cohort of all patients taking medications in the given items list is extracted from the medication prescription database and the cohort of all patients seeing doctors with the particular specialty code is extracted from the doctor consultancy database. These lists are then matched resulting in the desired table (see example below). Sorting this table gives the highest ratio of prescriptions per patients which can lead to the detection procedure for over-prescribing.

The first run of `get_doctor_behaviour` with `items` being psycho tropic drugs and `specialty_code` being psychiatrists, over a five years period, required a run time of about two hours, extracting about 115 psychiatrists, almost 10,000 patients and more than half a million transactions to analyse. Subsequent studies with different medication groups were each processed in less than a minute thanks to caching.

Doctor Code		1995	1996	1997	1998
x42r19$	Total Patients:	424	450	241	199
	Mental Patients:	167	198	142	131
	Ratio:	39%	44%	59%	66%
7%w#t0q	Total Patients:	372	336	335	389
	Mental Patients:	101	115	121	156
	Ratio:	27%	34%	36%	40%

Example 5. Episode Extraction
Episodes are units of health care related to a particular type of treatment for a particular problem. An episode may last anywhere from one day to several years. Analysing temporal episodic data from a transactional database is a hard task not only because there are very many episodes within a large database but also because episodes are complex objects with different lengths and contents. To facilitate better understanding and manipulation, the *DMtools* contain routines to examine their basic characteristics, like length, number of transactions, average cost, etc. One example is the *timelines* diagram which displays all medical

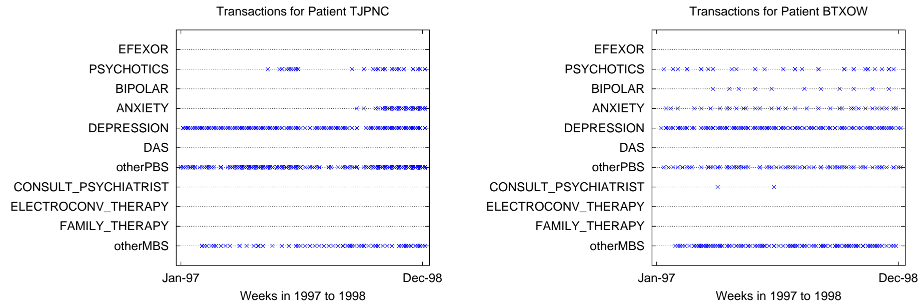

Fig. 2. Timelines: Medical services for two patients

transactions for a single patient split up for different groups of items as is shown in Figure 2. Detecting and extracting episodes from a transactional database is very time consuming and caching of such a functions is very helpful. It is feasible to cache several hundred thousand episodes – even if the resulting cache file has a size of hundred Megabytes – because the access time to get all these episodes is reduced from hours to minutes.

7 Outlook and Future Work

The *DMtools* is a project driven by the needs of a group of researches who are doing consultancies in health data mining. With this toolbox we try to improve and facilitate routine tasks in data analysis, with an emphasis on the exploration phase of a data mining project. It is important to have tools at hand that help to analyse and get a "feel" for the data interactively in the early stages of a data mining project, especially if the data is provided from external sources. This is in contrast to many data mining and knowledge discovery algorithms that aim at extracting information automatically from the data without any guidance from the user.

Ongoing work on the *DMtools* includes the extension of the toolbox with more analysis routines and the integration of algorithms like clustering, predictive modelling and association rules. As these processes are time consuming we are exploring methods to integrate external parallel applications (optimised C code using communication libraries like MPI [10]). Building graphical user interfaces (GUI) on top of our toolbox, Web enabling interfaces and exporting results via XML are on our wish list as is the publication of the *DMtools* as a package under a free software licence.

Acknowledgements

This research was partially supported by the Australian Advanced Computational Systems CRC (ACSys CRC). Peter Christen was funded by grants from

the *Swiss National Science Foundation* (SNF) and the *Novartis Stiftung*, Switzerland; and Tatiana Semenova is funded by a ARC SPIRT grant with the Australian Health Insurance Commission (HIC). The authors also like to thank Christopher Kelman (CDHAC) and the members of the CSIRO CMIS Enterprise Data Mining group for their contributions to this project.

References

1. D. Beazly, *Python Essential Reference*, New Riders, October 1999.
2. G. Bell and J. N. Gray, *The revolution yet to happen*, Beyond Calculation (P. J. Denning and R. M. Metcalfe, eds.), Springer Verlag, 1997.
3. M.J. A. Berry and G. Linoff, *Data Mining, Techniques for Marketing, Sales and Customer Support*. John Wiley & Sons, 1997.
4. E. Bezzaro, M. Mattoso and G. Xexeo, *An Analysis of the Integration between Data Mining Applications and Database Systems*, Proceedings of the Data Mining 2000 conference, Cambridge, 2000.
5. P. Chapman, R. Kerber, J. Clinton, T. Khabaza, T. Reinartz and R. Wirth, *The CRISP-DM Process Model*, Discussion Paper, March 1999. `www.crisp.org`
6. M.-S. Chen, J. Han and P.S. Yu, *Data Mining: An Overview from a Database Perspective*, IEEE Transactions on Knowledge Discovery and Data Engineering, Vol. 8, No. 6, December 1996.
7. D. Düllmann, *Petabyte databases*. Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-99), ACM Press, July 1999.
8. G. Graefe, U. Fayyad and S. Chaudhuri, *On the Efficient Gathering of Sufficient Statistics for Classification from Large SQL Databases*, Proceedings of KDD-98, 4th International Conference on Knowledge Discovery and Data Mining, AAAI Press, 1998.
9. J.M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman,T. Roth and P. Haas, *Interactive Data Analysis: The Control Project*, IEEE Computer, Vol. 32, August 1999.
10. W. Gropp, E. Lusk and A. Skjellum,*Using MPI – Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Massachusetts, 1994.
11. D. Pyle, *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, Inc., 1999.
12. P.G. Selfridge, D. Srivastava and L.O. Wilson, *IDEA: Interactive Data Exploration and Analysis*, Proceedings of the ACM SIGMOD International Conference on Management of Data, 1996.
13. S. Thomas and S. Sarawagi, *Mining Generalized Association Rules and Sequential Patterns Using SQL Queries*, Proceedings of KDD-98, 4th International Conference on Knowledge Discovery and Data Mining, AAAI Press, 1998.
14. G. Williams, I. Altas, S. Barkin, P. Christen, M. Hegland, A. Marquez, P. Milne, R. Nagappan and S. Roberts, *The Integrated Delivery of Large-Scale Data Mining: The ACSys Data Mining Project*, In Large-Scale Parallel Data Mining, M. J. Zaki and C.-T. Ho (Eds.), Springer Lecture Note in Artificial Intelligence 1759, 1999.
15. R.J. Yarger, G. Reese and T. King, *MySQL & mSQL*, O'Reilly, July 1999.