NOTE TO USERS

DATA MINING WITH RELATIONAL DATABASE MANAGEMENT SYSTEMS

by

Beibei Zou

School of Computer Science
McGill University, Montreal

11th January 2005

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Your file Votre référence
ISBN: 0-494-12571-3
Our file Notre référence
ISBN: 0-494-12571-3

Abstract

With the increasing demands of transforming raw data into information and knowledge, data mining becomes an important field to the discovery of useful information and hidden patterns in huge datasets. Both machine learning and database research have made major contributions to the field of data mining. However, there is still little effort made to improve the scalability of algorithms applied in data mining tasks. Scalability is crucial for data mining algorithms, since they have to handle large datasets quite often. In this thesis we take a step in this direction by extending a popular machine learning software, Weka3.4, to handle large datasets that can not fit into main memory by relying on relational database technology. Weka3.4-DB is implemented to store the data into and access the data from DB2 with a loose coupling approach in general. Additionally, a semi-tight coupling is applied to optimize the data manipulation methods by implementing core functionalities within the database. Based on the DB2 storage implementation, Weka3.4-DB achieves better scalability, but still provides a general interface for developers to implement new algorithms without the need of database or SQL knowledge.

Résumé

La demande croissante de transformer des données brutes en une source de connaissances utiles, fait de l'exploration de données un outil indispensable à la découverte d'information substancielle, dissimulée à l'intérieur d'immenses ensembles de données. L'apprentissage automatique et la recherche en bases de données ont tous deux grandement contribué à l'avancement de l'exploration de données. Toutefois, les effort déployés pour améliorer l'extensibilité des algorithmes appliqués en exploration de données restent limités. L'extensibilité de ces algorithmes est primordiale, puisque ceux-ci doivent souvent manipuler dénormes quantités de données. Dans cette thèse, nous faisons un pas dans cette direction en élargissant les fonctionnalités d'un logiciel d'apprentissage automatique populaire, Weka3.4, afin qu'il puisse manipuler des ensembles de données plus grands que la mémoire principale, au moyen de la technologie qu'offre les bases de données relationnelles. Weka3.4-DB est implémenté pour stocker et accéder les données via DB2 avec une approche en géénéral à couplage faible. De plus, un couplage semi-fort est appliqué pour optimiser les méthodes de manipulation de données en implémentant des fonctionnalités noyau à l'intérieur de la base de données. Basé sur l'implémentation de stockage de DB2, Weka3.4-DB atteint un plus haut niveau d'extensibilité tout en fournissant une interface générale aux développeurs pour implémenter de nouveaux algorithmes, sans la nécessité de connaître les bases de données ou le SQL.

Acknowledgments

This thesis could not have been accomplished without many people's support. It is my pleasure to thank those people who have made this thesis possible. First of all, I would like to express my special and sincere gratitude to my supervisors, Dr. Doina Precup and Dr. Bettina Kemme. They have guided me and encouraged me with their enthusiasm, inspiration and great efforts. During the work on my thesis, they have provided sound advice, good teaching and a lot of ideas. I would have been lost without them.

I would also like to thank my other supervisor, Glen Newton, from the Canada Institute for Scientific and Technical Information (CISTI), of the National Research Council Canada (NRC), who has helped to establish the collaboration between the research work at the School of Computer Science of McGill University and the research group at CISTI, NRC. He has contributed to the research work with his expertise on relational database management systems and Java programming. Special thanks goes to Greg Kresko from CISTI who has helped to set up the DB2 server, and the rest of the research group at CISTI.

I am grateful to the research environment provided by the Distributed Information Systems Group at the School of Computer Science, which is a stimulating and fun place to learn and grow. I wish to thank Huigu Wu, Yi Lin, Chenliang Sun, Emory Merryman, Shuqing Wu, Brian Gabor, Qifang Zheng, and Xueli Li for their support and inspiration. I also appreciated Jean-Sebastien Légare's help on translating the abstract to the French version, and the system staff Andrew Bogecho for setting up the DB2 server for the early phase of the research.

I would like to thank my family members, my parents and my sister, who have supported me all the time. This thesis is a special dedication to my husband, Feng Qian, who has shared my happiness and sadness every day.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data mining applies computational and statistical technologies to discover useful information and hidden patterns in large datasets. It has been developed by both the *Machine Learning* and *Database* communities since 1990. Each of theses communities offers different approaches since they work from different perspectives. Machine learning researchers provide a solid theoretical framework and develop machine learning algorithms that are suitable for major data mining tasks. Database experts facilitate the data mining process by providing sophisticated and advanced data storage management technology. In Chapter 2, a short overview of data mining is given, and contributions of the machine learning and database communities are reviewed.

Classification is an important problem addressed in typical data mining tasks, such as analyzing scientific experiments, medical diagnosis, fraud detection, credit approval and target marketing. Many algorithms for classification have been developed in the machine learning community, including e.g. logistic regression, decision trees and naive Bayes. In Chapter 3, an overview of classification is presented, and typical algorithms are described. In order to improve the effectiveness of the data mining process, data preprocessing is necessary. Chapter 3 also provides a brief discussion of data preprocessing techniques that are typically applied in classification methods.

Most commercial database management systems (DBMS) are based on the relational

model introduced in the 1970s. They provide efficient data storage, fast access structures and a wide variety of indexing methods to speed up data retrieval. SQL is the standard query language that is supported by most relational DBMSs. It provides sophisticated query functionality like nested queries and aggregate functions. SQL can be embedded in a host language. Hence, it is possible to access relational DBMS through application programs. In Chapter 4, relational DBMS technology and SQL are discussed in more detail.

While scalability is an important issue for all algorithms, it is especially critical for machine learning algorithms that are applied to data mining problems, because handling huge amounts of data becomes inevitable for real data mining tasks. Some general strategies have been developed to deal with large datasets, such as sampling and data squashing. A potential problem of these strategies is that they introduce incredible overhead and sometimes even decrease the accuracy of algorithms. Other approaches focus on making specific algorithms, especially decision trees, more scalable. The alternative that has been adopted in this thesis is to explore commonly used and well-developed relational database systems as data storage and retrieval, which can be easily applied to all algorithms without sacrificing the accuracy. In Chapter 5, existing approaches to handle the scalability issue in data mining are discussed.

Weka3.4 is an open source machine learning software package, which has implemented many state-of-the-art machine learning algorithms. Since it is implemented using memory-based data structures, Weka3.4 can only be used on datasets that can fit into memory. In Chapter 6, the system architecture and data structures of Weka3.4 are discussed in detail. The scalability limitations of Weka3.4 make it a perfect target for exploring solutions that can improve the scalability of existing algorithms.

The goal of this thesis is to extend Weka3.4 to Weka3.4-DB. Weka3.4-DB is able to handle large datasets by storing them in and accessing them through data resource management systems, especially relational database systems. The ultimate goal is to enhance Weka3.4 to provide scalability for all algorithms implemented in the package. In order to achieve this goal, this thesis presents a new storage interface, put between

the data mining algorithms implemented in Weka3.4 and the storage system that represents the data. Furthermore, an implementation of this interface using the relational DBMS DB2 is presented. With this, all algorithms implemented in Weka3.4 can run in Weka3.4-DB without changes. That is, the algorithms use a DBMS when accessing the data but do not need to be aware of this. Also, new algorithms can be implemented without developers being required to know SQL. In principle, the move to Weka3.4-DB allows them to run on larger datasets than possible in Weka3.4. However, some algorithms use internally large data structures, limiting their scalability. We analyzed the logistic regression algorithm in more detail, and came to the conclusion that we have to additionally provide an abstraction for typical main memory data structures, like arrays, that are then implemented on DB2. We have adjusted the logistic regression algorithm to use the new data structures in order to further increase scalability. This leads to a higher integration with the database system, however it is transparent to the algorithm implementation.

Chapter 7 describes the interface, the DB2 implementation of the interface, and different version of the logistic regression algorithm.

Weka3.4-DB has been evaluated on both synthetic data and real data using the logistic regression algorithm. The experiment results show significant improvement in regard to scalability with reasonable execution time. The performance evaluation shows that our approach of using a relational database system while providing developers with a further data structure interface is a practical solution to provide scalability for data mining algorithms without the need to know SQL. In Chapter 8, the experiment design and results are discussed in more detail.

In Chapter 9, conclusions are drawn and future work is discussed.

Chapter 2

Data Mining

2.1 Overview

The task of data mining is to extract useful information from huge datasets. With technological advances in data storage and data management, scientists, business and medical researchers are able to gather, store and manage previous unimaginable quantities of data. The need of transforming raw data into information and knowledge has been increasing dramatically. Modern data mining is motivated by this change in data collection and the need for data analysis. Since the early 1990's, research in data mining has largely focused on computational and algorithmic issues rather than the traditional statistical aspects of data analysis. Normally, data mining involves an integration of techniques from multiple disciplines, such as machine learning, database technology, data pre-processing and data visualization.

For the past ten years, *Machine Learning* and *Database* research have been playing major roles in the field of data mining.

2.2 Data Mining and Machine Learning

Machine learning involves the study of how machines and humans can learn from data and has been an important component of research in artificial intelligence (AI). It aims to simulate human learning by programming machines to learn tasks by experience [25].

Early work in this field was strongly connected to theories in cognitive science, trying to build algorithms and machines that could adapt to data in a manner thought to be similar to human learning [35]. In recent years, much of the research in machine learning has shifted from modeling how humans learn to the pragmatic aim of constructing algorithms that can learn and perform well on specific tasks. This leads to a much greater overlap with applied statistics by adding a computational flavor.

According to Mitchell [25], most often, machine learning problems are formed in terms of a task, such as playing chess, a performance measure, such as % of games won against opponents, and experience, such as playing practice games. A machine learning algorithm developed for a certain task is based on the type of training experience available and the target function to be learned. The target function can be represented by the learning algorithm in terms of a tree, or a collection of rules, or a polynomial function, etc.

Machine learning algorithms prove to be valuable in the following application domains:

- data mining: hidden information or knowledge are needed to be discovered from large datasets automatically in domains, such as financial analysis and medicine diagnosis.
- not well-known domains: domain knowledge is not fully understood, such as speech recognition and computer vision.
- frequently changing domains: the desired functions change frequently, such as robotics and computer games.

Within machine learning (and supervised learning in particular), decision trees [31], artificial neural networks [5, 4], nearest-neighbor [3], naive Bayesian networks[13], and support-vector machines [38] are well known algorithmic approaches.

The significance of machine learning to data mining lies in the fact that many of the algorithms being used in data mining have solid foundations in machine learning.

2.3 Data Mining and Database Systems

Another strand of data mining research emerged in the 1990's within the database research community independently and in parallel with developments in machine learning.

The introduction of relational database concepts [10] and high-level data models [9] proved to be major conceptual breakthroughs in the database field, which provided general and principled frameworks for data modeling and access. Issues [15] such as updating the database in a systematic manner, answering structured queries about the data and controlling access and security in the context of multiple users, became the foundations of modern database management.

By early 1990's, relational database technology was successfully established in the commercial sector. But those relational database systems were never designed to support data analysis tasks. Instead, they are primarily designed for the purpose of storing and querying data, and to offer transaction support.

When interest in data warehousing began to grow in the early 1990's [22], database researchers quickly realized that not only did their customers want to store, manage, and access their data in a systematic fashion, but now they also wished to be able to analyze it. Developing data analysis algorithms that can operate directly on relational databases forms the main component of modern database-oriented research in data mining.

The work by Agrawal et.al [2] on association rule mining is probably the very first example that demonstrates how simple association rules can be mined from a relational database in an efficient manner. An example of an association rule is "if a person buys beers, then he is more likely to also buy chips with probability 0.8". Most work proposed by the database community emphasizes having very efficient data structures and algorithms for operating on data that does not fit into main memory, and searching in datasets for simple local patterns such as association rules. For example, Gehrke et.al [16] describe substantial computational and memory optimizations in their implementation of CART [7] by using special-purpose data structures, and they apply their algorithm to datasets involving millions of points. Bradley et.al [6] describe a heuristic algorithm for an implementation of the Expectation-Maximization (EM) algorithm applied to Gaussian mixture modeling on massive datasets, which seeks to minimize the number of passes through the dataset.

The influence of databases on data mining has led to an emphasis on the data access aspects of analyzing large datasets.

Chapter 3

Classification

3.1 Overview

Classification is an important problem in machine learning that has been addressed by many algorithms. The task is to predict categorical class labels based on several attributes or features. For instance, given the customer information described by *age*, *credit_rating* and *student* (if he or she is a student), if the class label is whether the customer is going to buy a laptop, then the classification task is to classify the customer information with binary values *Yes* and *No*. In order to label the new customer information with the category, the classification algorithm, in the following called classifier, has to be trained from labeled customer records. The given customer information is the training experience and the classifier is the target function as introduced in the previous chapter. More precisely, a set of labeled data records used for training the classifier is called the *training dataset*. The data records in a training dataset are called *training data* or *training data records*. Each training record consists of the same number of attribute/value pairs. Attributes can be, e.g., *age* and *credit_rating*. They describe the training data. Among those attributes, the one used as the target of a classification task is called *class attribute* or *class label*. The basic types of attributes are *numeric* and *nominal*. Numeric attributes

can have real or integer values. For instance, age is a numeric attribute and it can contain real numbers. Nominal attributes require discrete values. Usually, they have a list of possible values. For instance, the credit_rating is a nominal attribute and the possible values are *good* and *bad*. Nominal attributes are also referred to as *categorical* attributes. When algorithms need to weigh training data, additional *weight* attributes are used. Weight attributes have a numeric type. In general, a training dataset is described by its attributes and the number of data records it has. A set of labeled data records used for testing the classifier is called the *testing dataset*. The data records and attributes in a testing dataset are similar to those in the training dataset. A testing dataset is also described by its attributes and the number of data records it has.

The basic idea of classification is to train a classifier using labeled training data and then use the classifier to classify new data. Classification can be defined as a two-step process.

- Train a classifier, such as decision trees, classification rules or mathematical functions, based on labeled training data.
- Estimate accuracy of the classifier with testing data independent of training data. The accuracy rate is the percentage of the testing data that are classified correctly.

The major classification algorithms include decision trees, Bayesian networks, logistic regression, neural networks, support vector machines and k-nearest neighbor. Classifiers are evaluated by the following criteria [18]:

- predictive accuracy: the ability of the classifier to correctly predict class labels of new data
- speed: the time needed to train the classifier and the time needed to use the classifier
- robustness: the ability of the classifier to handle noise and missing values

- scalability: the efficiency in constructing the classifier given a large training dataset
- interpretability: the level of understanding and insight provided by the classifier

3.2 Regression

3.2.1 Overview

Regression [19] is a well-understood statistical technique for analyzing data. Most regression techniques are used to predict continuous labels, such as linear and non-linear regression. Using generalized linear regression, categorical data can also be analyzed. The regression classifier is formed as a statistical function, which models the relation between the class label and the attribute values of data records. In a regression function, the class attribute is called *response* variable, and the remaining attributes are called *predictor* variables.

The simplest form of regression is linear regression. In linear regression, the response variable Y is modeled as a linear function of one or more predictor variables. Linear regression with one predictor variable X_1 is called bivariate linear regression, such as $Y = a + bX_1$. Y is assumed to be normally distributed. For instance, suppose the class attribute, the number of credits that a customer can earn, is the response variable Y, and the attribute, the number of products the customer purchased, is the predictor variable X_1. A linear function can be built by solving the coefficients a and b based on the given customer data. Using this linear function, the value of the response variable for new customers can be computed based on values of the predictor variable. Linear regression with more than one predictor variable is called multiple regression, such as $Y = a + bX_1 + cX_2$. Y is modeled as a linear function of a multidimensional feature vector (X_1, X_2). The coefficients in linear regression can be solved by least squares estimation, which minimizes the error between the true value and the estimation of the function. In other words, the linear function that results

from least squares estimation represents the best estimate of the true value.

A more complicated model is nonlinear regression where the response variable and predictor variables are not linearly dependent. Instead, the model uses a polynomial function. Most often, however, the nonlinear model can be transformed into the linear one and solved by least squares estimation.

Generalized linear models are a broad set of models designed to generalize the linear model to target response variables of different types [23, 21]. The differences between the generalized linear model and the linear model are the following:

- the response variable Y does not need to be continuous and its distribution does not need to be normal.
- the response variable Y is a linear function of the predictor variables via a link function, which is determined by the distribution of Y.

The coefficients in the generalized linear model are solved by maximum likelihood estimation, which requires iterative computational procedures.

3.2.2 Logistic Regression

Logistic regression [21] is a member of the class of generalized linear models. The response variable of logistic regression is a binary variable Y that can take the value 1 as success with probability p and the value 0 as failure with probability $1 - p$. The probability that $Y = 1$, given the value of X, is denoted by $p(Y)$. In other words, the binary variable Y has a binomial distribution with parameter p. Logistic regression estimates the parameter p as a function of the predictor variables X (the vector of predictor variables). More precisely, assume that $p(Y)$ is given as:

$$p(Y) = \frac{e^{\beta X}}{1+e^{\beta X}}$$

and β is a coefficient vector, then the logistic regression function is a logit transformation of p:

$$logit(p(Y)) = \frac{p(Y)}{1-p(Y)} = e^{\beta X}$$

The goal of logistic regression is to find the best model to describe the relationship between the binary response variable and predictor variables. In other words, its goal is to find a good estimator $\hat{\beta}$ (estimated coefficient vector) of the coefficients β.

Logistic regression becomes unstable in some situations, e.g., if the number of predictor variables is close to the size of the training dataset, or predictor variables are highly correlated. Such cases lead to overfitting, which can make the logistic regression model appear perfect on the training dataset, while it performs badly on the testing dataset.

Logistic ridge regression is an extension of logistic regression, providing more stable functions. Ridge regression shrinks the regression coefficients by imposing a penalty on their size. The key idea behind ridge regression is to avoid overfitting by imposing a penalty on large fluctuations of the estimated coefficients. A complexity (regularization) parameter λ ($\lambda \geq 0$), called ridge parameter, controls the amount of shrinkage of the norm of β. The regularization put forward by Hoerl and Kennard [20] is the sum of squares of the regression coefficients. It is first introduced in the context of least squares regression by Hoerl and Kennard [20] and is adapted to logistic regression by Le Cessie and Van Houweligen [8].

Two similar approaches of deriving a ridge estimator (the estimator of regression coefficients) are discussed in [8]. One that was first introduced by [14] attempts to maximize the log-likelihood function with a penalty on the norm of β: $l^{\lambda}(\beta) = l(\beta) - \lambda\|\beta\|_2$, where $l(\beta) = \sum(Ylog(p(Y)) + (1-Y)log(1-p(Y)))$ and $\|\beta\| = (\sum(\beta_j)^2)^{\frac{1}{2}}$ is the norm of the coefficient vector β. Maximization of $l^{\lambda}(\beta)$ leads to $\hat{\beta}$. The ridge parameter λ controls how much the norm of β shrinks. When $\lambda = 0$, the solution is the ordinary maximum likelihood estimate (MLE). When $\lambda \to \infty$, the β_j all tend to 0. Therefore, the estimate $\hat{\beta}^{\lambda}$ is expected to be closer to the real value of β than the unrestricted maximum likelihood estimate (MLE). Another method introduced by [37] is to obtain $\hat{\beta}^{\lambda}$ by the Newton-Raphson maximization procedure. For more detail information, please refer to [37].

The way of choosing the ridge parameter λ discussed in [14] is based on minimizing an estimate of the prediction error of the model using one of three error measures: classification or counting error (MCE), squared error (MSE) and minus log-likelihood error (MML). With a Cross-Validation (CV) or Akaike Information Criterion (AIC), the predictive value of the logistic model is compared for various values of λ and an optimal λ is chosen such that the mean error rate is minimal.

For instance, given the customer information in the previous example, each data record is denoted as (x_i, y_i), where x_i is a vector representing the values of attributes *age, credit_rating, student* of the ith record and y_i is the class label *buy_laptop* of the ith record. In the simplest case in which the ridge parameter is given, the ridge algorithm would iterate through all customer records a number of times in order to derive the ridge estimator that can satisfy the unrestricted maximum likelihood function.

3.3 Data Preprocessing

Data preprocessing [30] is an important step in the data mining process. Incomplete, noisy and inconsistent data can affect the accuracy and efficiency of data mining methods, and sometimes even prevent data mining methods from being applied. Therefore, a number of techniques have been developed to improve the quality of data and consequent mining results. For classification, a number of general data preprocessing techniques have been proposed.

- Data cleaning: data cleaning attempts to fill missing values, where some data records don't have values for some of the attributes. Missing values are filled with the mean calculated based on all available attribute values, or the most probable value based on statistics. Furthermore, data cleaning attempts to remove or reduce noisy data. Data cleaning is often required when using regression methods.

- Relevance analysis: whenever irrelevant attributes or redundant attributes appear in the data, relevance analysis can be performed to remove those attributes from the learning process. It is also called feature selection. Relevance analysis tends to reduce the dimension of the feature space, which results in shorter learning time. Therefore, it improves the efficiency and scalability of the learning process. The general idea behind relevance analysis is to compute some measure that can be used to quantify the relevance of an attribute. For example, a simple measure used by some regression methods is based on the number of distinct values of one attribute. The fewer distinct values an attribute has, the less likely it is to be relevant.

- Data transformation: one type of data transformation is called generalization, which transforms data from lower-level concepts to higher-level concepts. The typical application of generalization is to transform continuous-valued attributes to nominal/categorical attributes (e.g., instead of exact age, age ranges are given), or transform nominal attributes to binary attributes (a binary attribute is a special nominal attribute, which only has two discrete values). Another type of data transformation is called normalization, which involves scaling all values of some attributes such that the normalized values fall within a small specified range, e.g., 0.0-1.0. Normalization can prevent one attribute with a large range from over-weighting another with a small range.

In order to perform data preprocessing, the data records have to be scanned usually one or more times.

Chapter 4

Relational Database Management Systems

4.1 Overview

A relational database management system (RDBMS) [33, 40] is a system that stores and manages data that follows the relational data model. An RDBMS provides functions to define data structures, integrity constraints, and to share and retrieve data.

The general idea behind relational databases is the *Entity-Relationship (ER)* model. The basic structure of representing data in the relational database is a *relation*, which is not the same as the *relationship* in the ER model. A relation represents data that belongs to one application dependent entity type. For example, all information about customers could be stored in one relation. A relation consists of a *relation schema* and *relation instance*. The relation instance is a set of tuples, each tuple describing one entity (e.g. one customer). The schema provides the meaning behind the tuples in the relation. It provides the name (e.g. Customers), the attributes which describe the entity type (e.g. name, age, salary) and the domain of each attribute (e.g. string, integer). An example of a relation schema is :

Customers(id:string, name:string, age: integer, rank: integer)

where each attribute name is followed by a domain name. For instance, the attribute `name` has domain `string`, which defines that the value associated with the attribute `name` must be a character `string`. An instance of a relation is a set of tuples, where each tuple has the same number of attributes with attribute values according to the attribute domains. Often, a relation instance is referred to as a table, tuples are the rows in the table, and attributes are the columns (attribute names are the column headers). Tuples are also called records, and attributes are called fields. As such, a relation is an ideal form to represent a dataset used for data mining as introduced in Chapter 3. Attributes have the same meaning, and data records are the tuples in the relation.

A relational database is a structured way of storing information, and RDBMS prevent entering incorrect data by allowing the definition of integrity constraints. An integrity constraint (IC) is a condition over a database schema that restricts the data to be stored in a table of the database. Among many kinds of integrity constraints, the most important one is the key constraint. A key constraint defines a certain minimal subset of attributes of a relation, called the *primary key*, that is a unique identifier for a tuple. No two tuples can have the same values in the primary key attribute. For instance, the primary key of `Customers` relation is `id`, which means no two customers have the same id. Therefore, tuples or records in a table can be easily accessed by referring to the primary key of the table. Another key constraint is called *foreign key*. A foreign key in a relation A refers to a primary key in relation B to enforce referential integrity among the tables. The foreign key in the referencing relation must match the primary key of the referenced relation. For instance, in addition to `Customers`, a second relation is:

GoldenMemberCard(cid:string, credit:int, id:string)

The primary key of the `GoldenMemberCard` relation is `cid`. Additionally, the relation has the foreign key `id`, referring to relation `Customers`, and the `id` is primary key in `Customers`. The foreign key constraint ensures that only customers that exist in the `Customers` relation (there is a tuple in `Customer` with this id) can have golden member

cards. That is, any value that appears in the `id` of a tuple of the `GoldenMemberCard` relation must also appear in the `id` of some tuple in the `Customers` relation.

4.2 SQL

Data in a relational database is accessed via a database query. A database query is an operation that either retrieves data from the database (again in form of a relation) or modifies the data. A query language is a specialized language for writing queries.

Structured Query Language (SQL) [33, 40] is the most widely used commercial relational database language. It was originally developed at IBM in the SEQUEL-XRM and System-R projects in the early 1970s. The Sequel language has evolved since then, and its name has changed to SQL. SQL statements can be divided into two categories: *data definition language(DDL)* and *data manipulation language(DML)*. DDL statements are used to build and modify the structure of tables and other objects, e.g. *CREATE, DROP* statements. DML statements are used to work with the data in tables, e.g. *INSERT, SELECT, DELETE, UPDATE* statements.

4.2.1 Basic Structure

The basic structure of an SQL SELECT statement consists of three clauses: *select*, *from* and *where*.

- `select` clause: it contains a list of attribute names of tables appearing in the `from` clause
- `from` clause: it contains a list of table names that are needed to be evaluated in the query
- `where` clause: it contains a Boolean combination of conditions on attributes of tables appearing in the `from` clause

An example of a basic SQL query using the Customers relation is

```
select id, name
from Customers
where age > 30
```

It selects the values of the `id` and `name` attributes of all tuples in the relation `Customers` where age values are over 30. The result is in the form of a table with two attributes: `id` and `name`.

SQL provides a special attribute value *null* to represent any unknown or inapplicable attribute value. The comparison operator to test whether an attribute value is *null* is `IS NULL`. The opposite comparison operator is `IS NOT NULL`. For instance, the above SQL query can be modified as :

```
select id, name
from Customers
where age > 30 and rank IS NOT NULL
```

The query has an extra condition on attribute `rank`. It will select values of attributes `id` and `name` of tuples from `Customers` table where age values are over 30 and rank values are not null.

SQL allows to specify the order in which result tuples are displayed. The `order by` clause makes the tuples in the result of a query to be sorted in some order. For instance,

```
select id, name
from Customers
where age > 30 and rank IS NOT NULL
order by name
```

It returns the same tuples as the previous query but tuples are displayed in ascending order on the values of attribute `name` (i.e. alphabetically in this example). The default

sort order is ascending, but descending (`order by name desc`) is also possible.

4.2.2 Aggregate Functions

SQL supports a class of built-in aggregate functions for computing aggregate values such as MIN and SUM. There are five aggregate functions: `avg` for average, `min` for minimum, `max` for maximum, `sum` for total, and `count`. For instance,

```
select min(age), count(name)
from Customers
where rank > 2
```

The query returns the minimum age of customers and the number of customers whose rank is over 2.

Often, aggregate functions are applied to groups. SQL provides a `group by` clause to group tuples of a table based on certain attribute values and `having` clause to specify the group qualification. The attribute names that appear in the `select` clause must also appear in the `group by` clause or must be aggregated. The group qualification in the `having` clause is applied to all groups before generating the final result groups. For instance, consider the query:

```
select rank, count(rank), avg(age)
from Customers
group by rank
having count(*) > 10
```

The query returns a list of attribute `rank` values, the number of tuples that have this rank value, and the average age for customers with this rank. Only if there are more than 10 tuples in the `Customers` relation that have this rank, a result tuple is returned.

The `group by` clause can be used as an independent clause and without including any aggregate function. An example is:

```
select rank
from Customers
where rank > 2
group by rank
```

It is a normal select query with one extra `group by` clause. The query returns the list of existing rank values that are over 2.

4.2.3 Join Operations

Join operations take two relations and return as a result another relation. Join operations consist of a *join type* and a *join condition*. The join condition defines which tuples in the two relations match, and what attributes are present in the result of the join. The join type defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated. The default type is *inner join*. The inner join only returns the matched tuples from both relations and ignores those that tuples in each relation do not match any tuple in the other relation. Another type is *outer join*. Unlike the inner join, the outer join keeps both matched tuples and those tuples that do not match any tuple in the other relation. A simple example of inner join is:

```
select name
from Customers inner join GoldenMemberCard
on id
```

This query only returns those customers who have the golden member cards, since only those tuples in the `Customers` relation that can match tuples in the `GoldenMemberCard` with same `id` values, will be returned. In this case, all the tuples in the `GoldenMemberCard`

relation will be returned, since all of them have matched tuples in the `Customers` relation due to the foreign key constraint.

4.2.4 Nested Queries

One powerful feature of SQL is nested queries. A nested query is a query that has another query embedded inside it. The embedded query is called a subquery. Whenever a query needs to express a condition that refers to the result of another query, the subquery is used to compute the subsidiary result table and appears as part of the main query. A subquery typically appears in the `where` clause of a query. The most common use of subqueries is to perform tests for set membership. SQL allows testing tuples for membership in a relation with the key word *in* and testing the absence of set membership with the key word *not in*. For example, the following nested query finds all customers who have the golden member cards.

```
select name
from Customers
where id IN (select id
             from GoldenMemberCard)
```

4.3 Indexing

When queries are executed over large relation instances, execution time can be very slow, since the entire relation must be scanned and for each tuple the condition in the `where` clause must be evaluated. Indexing [33, 40] is used to speed up the query processing time.

Let's first have a look at equality in queries. The query below

```
select *
from Customers
```

```
where id = 100
```

selects exactly one tuple, namely the customer with `id` = 100. Equality queries can also retrieve several tuples, namely if the attribute in the `where` clause is not the primary key or unique. Equality queries typically (not necessarily) pick up few tuples of the relation. However, if no index is defined, the entire relation has to be scanned to find the matching tuples. An index is defined over one or more attributes, called the search key of the index. It is an additional data structure, such that, given values for the search key attributes (e.g. 100) it can efficiently determine the physical location of the matching tuples. There exist many different indexing methods (e.g. B+-Tree, hashing). Some of them can also be used for range queries. For example, the range query

```
select rank
from Customers
where age<20
```

selects the ranks of customers younger than 20. The index now determines the physical location of the corresponding tuples starting with the youngest customer. If the number of matching tuples is small, retrieving the matching tuples one by one will still be faster than scanning the entire relation. However, if there are many matching tuples, which might be spread all over the physical storage of the relation, the direct access to these tuples one by one might be slower than simply scanning the entire relation and performing the attribute test (< 20) on each tuple. There is the possibility to create one clustered index per relation. A clustered index is, in principle, the same as an unclustered index (defined over one or more search key attributes), but the tuples in the physical storage of the relation are actually sorted according to their order of the search key attributes. For instance, if we create a clustered index over the `age` attribute, all tuples of the `Customers` relation will be physically ordered according to their age on the physical storage. Hence, retrieving all customers younger than 20 will start with the first customer, and then sequentially retrieve all following

customers until the first customer with age = 20 is found. This is faster than scanning and testing all tuples.

4.4 JDBC

SQL is a powerful declarative query language, but it does not provide the full expressive power of a general-purpose language. Java Database Connectivity (JDBC) [33, 40] enables the integration of SQL into a general-purpose programming language through an application programming interface (API). It allows application programs to access different DBMSs without recompilation. Any direct interaction with a specific DBMS is through a DBMS specific driver, which can translate the JDBC calls into DBMS-specific calls. Drivers are loaded dynamically on demand, since only at run-time it is known which DBMSs the application is going to access. All the existing drivers are registered with a driver manager.

There are four main components of the architecture of JDBC:

- application: the application (Java program using JDBC API) initiates and terminates the connection with the data source. It submits SQL statements and retrieves the results through a well-defined interface as specified by JDBC API.
- driver manager: the driver manager is used to load JDBC drivers and to pass JDBC function calls from the application to the correct driver. It handles JDBC initialization and information calls from the application and logs all function calls. In addition, it may perform some error checking.
- drivers: the driver establishes the connection with the data source. It submits requests and returns results by translating data, error formats, and error codes from a form that is used by the data source into the JDBC standard.
- DBMS: the DBMS processes commands from the driver and returns the results.

JDBC is a collection of Java classes and interfaces that enables database access from programs written in the Java programming language. The classes and interface are part of the `java.sql` package. The major steps of using JDBC is to load the JDBC driver, connect to a data source, and execute SQL queries. An example of a small application program is

```
import java.sql.*;

public class JDBCexample{

  public static void main(String[] args) throws SQLException{

    //Load the DB2 JDBC driver
    DriverManager.registerDriver(new com.ibm.db2.jcc.DB2Driver);
    //build a connection
    Connection conn = DriverManager.getConnection(url, uid, passward);
    //create statement
    Statement stmt = conn.createStatement();
    Resultset rs = stmt.executeQuery
                  (''select age from customers where rank = 5'');
    while(rs.next()){
      int age = rs.getInt(1);
      System.out.println(''rank 5, age: ''+age);
    }
    rs.close();

    //create preparedStatement
    PreparedStatement pstmt = conn.prepareStatement(
                              ''select avg(age) from Customers ''+
                              ''where rank = ?'');
    rs = pstmt.executeQuery();
```

```
    for(int i=1; i<=10; i++){
      pstmt.setInt(i);
      rs = pstmt.executeQuery();
      while(rs.next()){
        int age = rs.getInt(1);
        System.out.println(''rank ''+i+'', age: ''+age);
      }
    }
    rs.close();
    pstmt.close();
    stmt.close();
    conn.close();
}
```

In JDBC, DBMS drivers are managed by the `DriverManager` class. As it is shown in the example, the static method `registerDriver` is called to register the DB2 driver. A connection with the DBMS is started through the creation of a `Connection` object. In the example, the `getConnection` method of the `DriverManager` class returns a `Connection` object, which represents a communication channel to the DB2 database. `getConnection` has to indicate the location of the DB2 database in form of a url. Furthermore, a user name and password must be given since only authorized users can access the database.

The SQL query is submitted to the database through the `Statement` object. In the example, the `createStatement` method of the `Connection` object returns a `Statement` object, and the `executeQuery` method of the `Statement` class submits the query to DB2 and lets it get executed in the database. Once the SQL query is executed, a resulting relation is returned through the `ResultSet` class. Since the `ResultSet` object of a select query represents a set of tuples in the resulting relation, but the application program can only handle one tuple or even one attribute at a time, the `ResultSet` class provides methods to iterate through the returned tuples. After a query is executed, the `ResultSet` is positioned right before the first tuple. The method `next`

fetches the next tuple and enables reading of the attribute values through `gettype` methods, where type is the type of the field. When the `ResultSet` reaches the last tuple, the `next` method can't satisfy the condition of the `while` loop any more, since calling the `next` method makes the `ResultSet` to position right after the last tuple. Potentially, a `ResultSet` might contain millions of tuples. They might not fit all in the address space of the client JDBC program. Hence, the `ResultSet` is usually stored as a temporary relation at the database, and tuples are transferred from DB2 to the program space one by one when `next()` is called. Depending on the JDBC driver, a small set of tuples might be cached at the client side. At end of the program, the `Connection`, `Statement` and `PreparedStatement` objects are closed.

Besides the `Statement`, JDBC allows the creation of another kind of statement, called `PreparedStatement`. `PreparedStatement` can refer to variables in the application program and is good for repeating one query many times once it is compiled. JDBC replaces each parameter with a `?` and sets values for each parameter at run-time through `settype` methods, where `type` is the type of the parameter. In the example, the `prepareStatement` method of the `Connection` class returns a `PreparedStatement` object. At this time point, the query exists in compiled form at the database. It can be called and executed without compilation with different values for its input parameter `rank`. Once the query is compiled by the `PreparedStatement` object, the value of the parameter in the `PreparedStatement` object is set by the `setInt` method in a loop and the query is executed repeatedly.

Chapter 5

Scalability

5.1 General Challenges Imposed by Massive Data Sets

One of the main challenges in dealing with massive datasets is the scaling effects that often occur as datasets grow in size. For a dataset with p attributes and N data records, the time complexity [11] of a data mining algorithm is typically expressed as the worst running time as a function of N and p, e.g. $O(Np)$. Algorithms whose time complexity scales poorly as a function of N are often unacceptable for large datasets. Therefore, data mining researchers interested in massive dataset applications often focus on algorithms that scale in the "near-linear" range for N and usually no worse than p^2 for p.

The other relevant aspect of data analysis for large datasets concerns the physical storage location of the data relative to CPU. The primary memory consists of RAM (random-access memory) and has the benefit of allowing relatively fast random access of any byte on the order of 10^{-7} to 10^{-8} seconds with current technology. This is how long it takes the system to bring the data from memory to CPU to do a computation. Secondary memory consists of disk storage. The access time here is on the order of 10^{-2} seconds. Even though the storage technology is consistently changing (currently

allowing storage on the order of Gigabytes/a RAM and Terabytes/a hard disk), the relative difference in access time between primary and secondary memory still remains on the order of 10^4 to 10^5.

Thus, the time complexity mentioned earlier will be affected dramatically by the physical location of the data. If the algorithm requires one computation per data record, and each data record is accessed randomly, then the time taken by the algorithm will be proportional to cN, where N is the number of data records and c is the time it takes to access the data record. It simply indicates that algorithms that frequently access the disk will be much slower than algorithms that operate on data entirely in main memory. If the data is organized so that it can be accessed sequentially from the disk, then the cost of disk access decreases, since sequential scanning can be carried out much more efficiently than random access of the same amount of data. But many data mining algorithms either repeatedly access different subsets of the data in an unpredictable way, e.g. classification trees, or require multiple passes through the entire dataset. Even if such algorithms scale reasonably in N and p, while they may run in reasonable time on data in main memory, they will be infeasible for large datasets that exceed main memory capacity.

5.2 General Approaches to Achieve Scalability

There are a number of general approaches for developing scalable data mining algorithms.

- Running a random sample of the whole dataset is often used in practice, especially for data mining tasks involving iterative and interactive phases of model-building. But generating a fairly random sample from a large database stored on disk may itself be a time-consuming task from a computational point of view.
- Du Mouchel et.al. [27] proposed a statistically-motivated method for data-squashing, which creates a set of M weighted pseudo data points, where M is

much smaller than the original number N, and the pseudo data points are automatically chosen by the algorithm to mimic as closely as possible the statistical structure of the original large dataset. The method is empirically demonstrated to provide one to two orders of magnitude reduction in prediction error on a logistic regression problem compared to simple random sampling of a dataset. Similar ideas by Moore and Lee [26], Bradley et.al. [6] and Pavlov et.al. [28] propose to generate a smaller approximate representation of the original large dataset that matches the statistical characteristics of the original dataset as closely as possible. One advantage of this general approach is that once the reduced set is created, the original dataset can be thrown away and computationally intensive processes, such as visualization and model-building, can take place entirely on the reduced dataset in main memory.

- Pipelining is a quite effective online recursive approach. It processes the data through the analysis system as it arrives and recursively updates model parameters in an online adaptive manner. Cortes and Pregibon [12] describe an impressive system at AT&T, which adaptively updates estimates on whether a telephone line is a business or a residence, for about 350 million customers per night, based on about 300 million records of daily phone calls. Logistic regression models are trained offline and the probability of a number being a business is modeled by a logistic regression model with input variables based on characteristics of calls, such as time of day, length of calls, etc.

- Provost and Kolluri [29] describe a variety of other techniques for scaling up to massive datasets. They categorize those techniques into three main approaches. The straightforward approach is to build fast algorithms by restricting the space of models to be searched or developing powerful search heuristics. The idea of the second approach is to partition the data into subsets to process them in parallel, and to compute the final result as a function of the results retrieved from the parallel computations. The last approach is to use a relational representation. A typical way of using relational data directly is to integrate data mining algorithms with database management system (DBMSs). This is the

approach that we will take in this thesis.

More specialized approaches have been developed for particular algorithms, e.g. decision tress [24, 39, 17].

5.3 Achieving Scalability by using Relational Database Systems

Since more and more massive datasets are stored in database systems, developing an effective architecture for a data mining system on top of a database system becomes an interesting implementation issue. Database systems are designed to provide the flexibility and efficiency of sorting, organizing, accessing and processing data. Data in database systems tends to be well organized, indexed, cleaned, and integrated, which makes many tasks, such as finding relevant data, much easier than for data in flat files. Once data mining systems take advantage of database systems, more scalable algorithms and data structures can be explored.

Han and Kamber [18] propose a number of architectural alternatives that could be developed.

- No Coupling: A data mining system will not use any functionality of the DBMS. It fetches the data from some file system, processes the data using some data mining algorithms, and then stores the mining results in another file.
- Loose Coupling: A data mining system will use the basic functions of the DBMS to store and fetch data. But, it does not explore data structures and query optimization methods provided by the DBMS. In this case, it is hard to achieve high scalability and good performance for large datasets.
- Semitight Coupling: Beside the basic functionality of DBMSs, a few essential data mining primitives are implemented within the DBMS. The primitives can

include sorting, indexing, aggregation, histogram analysis, multi-way join, and precomputation of some essential statistical measures, such as sum, count, max, min, standard deviation, etc. Even some frequently used intermediate mining results can be precomputed and stored in the DBMS.

- Tight Coupling: A data mining system is completely integrated with the DBMS. The data mining system is one functional component of the integrated system. Data mining queries and functions are optimized based on mining query analysis, data structures, indexing schemes, and query processing methods of the DBMS.

The very early work conducted by Agrawal et.al. [32] compares the performance of loose coupling and tight coupling alternatives for developing the well-known data mining algorithm Apriori et.al. [2] on a relational database system. The loosely-coupled Apriori algorithm is developed using a standard application program with embedded SQL statements (similar to Java with JDBC) where the application program runs on a different machine or at least in different access space. The tightly-coupled Apriori algorithm is developed with user-defined functions, which are defined by the application programs, but executed within the database engine. They claim the tight-coupling gives more than two fold performance advantage over loose-coupling based on experiments on six real-life customer datasets.

Agrawal et.al. [36] have further worked on integrating association rule mining with relational database systems. They attempt to understand the implications of various architectural alternatives for coupling data mining with relational database systems. The most important one is called *Cache-Mine*. The basic idea of Cache-Mine is that after reading the data once from the DBMS, the data mining algorithm temporarily caches the relevant data in a look-side buffer on a local disk. The cached data could be transformed to a format that enables efficient future accesses. The cached data is discarded when the execution completes. The advantage of Cache-Mine is great programming flexibility. The disadvantage is that it requires additional disk space for caching. Cache-Mine is reported to perform better than other alternatives.

The work done in this thesis is inspired by all the previous work related to using relational databases to scale up existing machine learning algorithms. It aims to scale up a very popular open source package of machine learning algorithms, *Weka3.4*, by taking advantage of the efficient storage and retrieval of relational representation. Since one of the goals is to not change (or minimally change) the algorithms implemented in the software package, we employ generally a loose coupling approach. However, for some special functions we provide a semitight coupling. Furthermore, we looked at the logistic regression algorithm in more detail and slightly adjust it to the new architecture to increase scalability even further. This adjustment can still be considered as loose coupling. Overall, we extend algorithms in the package to adjust to the new storage system without modifying the results of algorithms.

Chapter 6

Weka3.4

6.1 Introduction

Weka3.4 [42] is a popular, open source, machine learning software package, which has been developed at the Department of Computer Science, University of Waikato, New Zealand. Weka3.4 implements many state-of-the-art machine learning algorithms, and is widely used in teaching and research by the machine learning community, as well as outside users. Weka3.4 contains tools for data preprocessing, regression, mining association rules, classification, clustering and visualization.

6.2 System Architecture and Data Structures

This thesis extends the storage system of Weka3.4. Hence it is crucial to understand Weka3.4's system architecture and data structures.

The architecture of Weka3.4 consists of a GUI user interface, machine learning algorithms and one well-defined data structure interface, `core`. As shown in Figure 7.1, all the algorithms in Weka3.4 are using data structures and methods that can manipulate data through `core`.

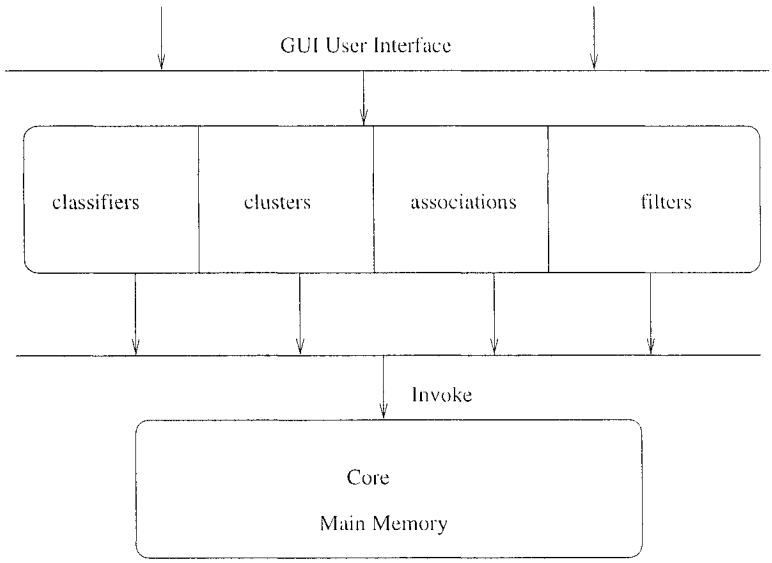

Figure 6.1: Architecture of Weka3.4

All the data structures defined in `core` are memory-based data structures. The most important are the `Instances` and `Instance` classes. `Instances` is implemented to store the dataset information, providing functionality to access attribute information and to manipulate data records inside the dataset, such as delete and sort the data records. `Instance` is implemented to store the information of any single data record, providing functionality to access weight, attribute and class values. Each data record is stored as one `Instance` object, and all `Instance` objects are stored in a vector of `Instance` objects, which is maintained in the `Instances` class. The `Instances` class provides methods to allow algorithms to enumerate all the data records or access one specific data record by its position in the dataset. Important methods of `Instances` are summarized in Table 6.1.

Normally, at the start of a data mining algorithm, the training data is loaded into main memory and stored as an `Instances` object. During the computation, more `Instances` objects are instantiated by creating a copy of an existing `Instances` object. If the data records of none of the `Instances` objects are modified, they can be

Method	*Description*
Instances(Instances dataset)	Constructor copying all instances and references to the header information from the given set of instances
Instances(Instances dataset, int capacity)	Constructor creating an empty set of instances
Instances(Instances source, int first, int toCopy)	Constructor creating a new set of instances by copying a subset of another set
Instances(Reader reader)	Constructor reading an ARFF file from a header and assigning a weight (1.0) to each instance
Instances(Reader reader, int capacity)	Constructor reading the header of an ARFF file from a reader and reserving the space for the given number of instances
Instances(String name, FastVector attinfo, int capacity)	Constructor creating an empty set of instances
void add(Instance instance)	adds one instance to the end of the dataset
void delete(int index)	removes an instance at a given position of the dataset
void deleteWithMissing(Attribute att)	removes all instances with missing values for a particular attribute from the dataset
Enumeration enumerateInstances()	returns an enumeration of all instances in the dataset
Instance firstInstance()	returns the first instance in the dataset
Instance instance(int index)	returns the instance at the given position
Instance lastInstance()	returns the last instance in the dataset
double meanOrMode(Attribute att)	returns the mean/mode for the numeric/nominal attribute as a floating-point value

Method	*Description*
int numDistinctValues(Attribute att)	returns the number of distinct values of a given attribute
int numInstances()	returns the number of instances in the dataset
void randomize(Random random)	shuffles the instances in the dataset so that they are ordered randomly
void renameAttribute(Attribute att, String name)	renames an attribute
void renameAttributeValue(Attribute att, String name, String val)	renames a value of a nominal attribute value
Instances resample(Random random)	creates a new dataset with the same size using random sampling with replacement
void sort(Attribute att)	sorts the instances based on an attribute
void stratify(int numFolds)	dynamically groups a set of instances according to its class value if the class attribute is nominal
double sumOfWeights()	computes the sum of all instances' weights
double variance(Attribute att)	computes the variance of a numeric attribute

Table 6.1: Methods of `Instances` class

shared, i.e., a lazy update is deployed. When a new `Instances` object is created, it shares the same vector of `Instance` objects with the `Instances` object from which it was copied. Once it wants to change an `Instance` object in the vector (i.e. a data record), it creates its own copy of the vector of `Instance` object. Then it creates a copy of each `Instance` object it wants to update. Figure 6.2. shows in detail how three `Instances` objects share the same data records or have their own copies. For example, `Instances` object 1 and `Instances` object 2 share the same vector of `Instance` objects, but `Instances` object 3 has its own vector of `Instance` objects. Both vectors share some `Instance` objects. But, for data record A, both vectors have different copies.

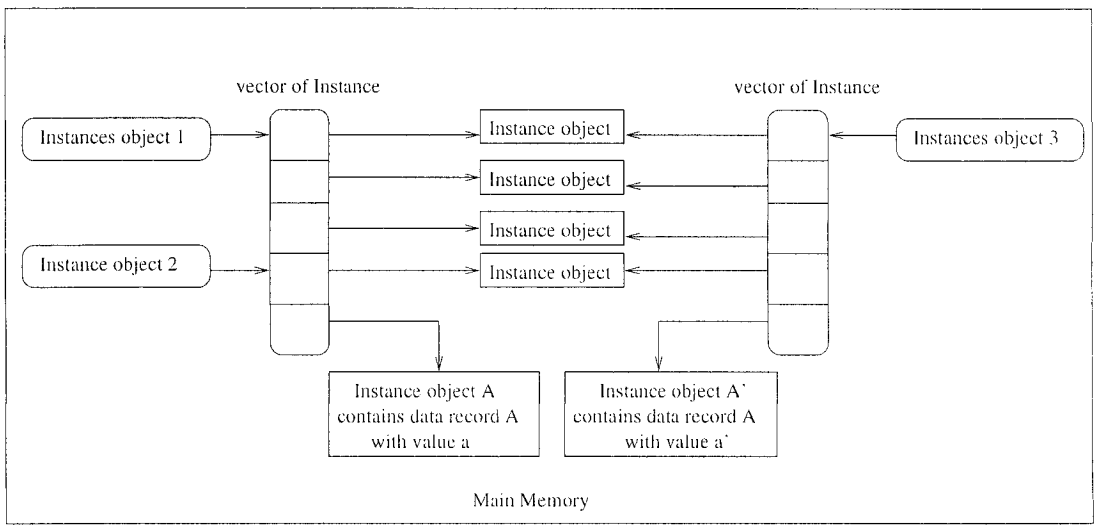

Figure 6.2: Instances and Instance

6.3 Data Flow

In this section, we will shortly discuss the important methods in Table 6.1. They are grouped into the following categories:

1. Storing Data

   When an algorithm needs to load data into main memory, it creates an `Instances` object to store all the attribute name and type information, convert each data record into one `Instance` object, and call the `add(Instance instance)` method to insert the `Instance` objects into the vector of `Instance` objects, which is maintained in the `Instances` object. Afterwords, `Instances` objects are typically created by copying an existing `Instances` object. The new object shares the same vector of `Instance` objects with the original object or creates its own vector but shares some `Instance` objects. Both cases are shown in Figure 6.2. When an object of `Instances` calls its own methods, executing those methods can only affect its own copy of data, more precisely, the vector of `Instance` objects.

2. Data Access

   When an algorithm needs to access all the training data, it uses the corresponding methods of the `Instances` class. It can either call `enumerateInstance()` to get an enumeration of `Instance` objects or call `instance(int position)` in a loop to iterate over the `Instance` objects. The `enumerateInstance` method implements the `Enumeration` interface in the Java library. It accesses the `Instance` object from the vector of `Instance` objects in the `Instances` object. The `instance(int position)` method can access a specific data record by locating the corresponding `Instance` object in the vector based on the given position. `firstInstance()` and `lastInstance()` are two other methods to access data records, which are implemented by invoking the `Instance(int index)` method with the corresponding indices. Information about the dataset can be returned by methods like `numInstance()` and `numDistinctValues(Attribute att)`.

3. Data Manipulation

   An algorithm can manipulate data through methods, such as `delete(int position)`

and `sort()`. The `delete(int position)` method deletes data records by removing the corresponding `Instance` object from the vector of `Instance` objects (note that this does not delete the `Instance` object itself, because some other `Instances` may still point to it). The `sort()` method sorts the vector of `Instance` objects based on one attribute value, by applying the `quicksort` algorithm on the vector.

4. Computing Statistics

   An algorithm can compute statistics about the data records through methods, such as `sumofWeights` and `meanOrMode`. The `sumofWeights` method sums the weights of data records by looping through the vector of `Instance` objects and accessing the weight value of each `Instance` object. The `meanOrMood` method computes the mean for a numerical attribute and mood for a nominal attribute over the `Instance` objects in the vector. Similar to the `sumofWeights` method, it calculates the sum of weights and the sum of products of the weight with the corresponding value at the given attribute by looping through the vector of `Instance` objects.

6.4 Performance Limitation

Weka3.4 is a memory-based package and all the algorithms implemented in Weka3.4 are typical machine learning algorithms, which do not address the scalability issue. Weka3.4 can achieve a good performance in terms of execution time, but it is hard to scale up with large datasets. Since it is implemented in Java, in order to save memory, Weka3.4 takes full advantage of the object-oriented language by sharing objects as much as possible. Except for a few incremental machine learning algorithms, most algorithms, especially the classification algorithms implemented in Weka3.4, need to train using all the training data, which means the training dataset has to be in memory most of the time. Therefore, the size of the training data becomes critical, and eventually, most algorithms will run out of memory with large training datasets.

This is especially true if they create several `Instances` objects that maintain their own `Instance` objects. But, unfortunately, dealing with large datasets is quite common in most data mining tasks. Therefore, this kind of implementation prevents Weka3.4 from being applied to many general data mining tasks of interest for applications.

Chapter 7

Weka 3.4-DB

7.1 Intuition and Goal

As we have discussed in Chapter 5, the scalability of algorithms is an important performance issue. So far, either novel algorithms have been developed or special strategies have been deployed to preprocess data in order to make it fit into main memory. But the first approach is limited to a small number of algorithms and can't resolve the problem in general. Although the second approach does apply to all the algorithms, it can add incredible overhead when preprocessing a huge amount of data and sometimes will even reduce the accuracy of algorithms. In this thesis, we follow a different approach to allow data mining algorithms to access arbitrarily large datasets. We replace the limited main memory as a main storage medium with the potentially unlimited secondary storage. However, secondary storage access is complex and very time consuming. Hence, we take advantages of relational database systems, which can provide sophisticated data storage and retrieval offering a powerful data access API.

Despite all the previous work related to using relational databases to scale up existing machine learning algorithms, there is still a lack of general approaches that can work

with all kinds of learning algorithms. Most existing work focuses on specific kinds of algorithms, such as decision tree algorithms or association rule algorithms. The most systematic work that has been done is the one by Agrawal et.al. [36], but it is restricted to association rule mining. A further restriction of the approach is that the developers of the association rule mining algorithm must be very familiar with database technology, either implementing stored procedures within the database or accessing the database through SQL. However, SQL is quite difficult to understand for machine learning researchers not familiar with database technology. Hence, our idea is to give these researchers their familiar interface, in particular, the `core` interface of Weka3.4 as described in the previous chapter. In contrast to the current implementation of `core`, however, data is not necessarily stored in main memory, but loaded from and stored into a relational database as needed. Ideally, this approach allows all existing algorithms in Weka3.4 to run on datasets that do not fit into main memory without any change to the algorithms themselves. In principle, the developer of the algorithm does not even have to be aware of any main memory restriction. Hence, the first goal of the thesis was a re-implementation of the `core` package of Weka3.4 without any change to its interface. The goal is to have no memory restriction on objects managed within `core`. The advanced functions of DB2 should be explored whenever possible within the new implementation.

However, this general approach might restrict performance in two ways:

- The data mining algorithms might create their own large main memory structures by not being aware of main memory limitations. In this case, a simple `core` is not enough to allow unlimited dataset sizes.
- The data mining algorithms, not being aware of the frequent secondary storage access, might access the `core` data structure in a very inefficient way leading to very large execution time.

As such, the second task of this thesis was to determine whether the existing algorithms can be made more efficient and scalable by (i) simple changes to the `core`,

and/or by (ii) providing additional interface methods that can help increase the efficiency and scalability of the algorithms.

7.2 Data Structures

The basic idea of extending Weka3.4 to Weka3.4-DB is to create a general data structure interface that defines data structures and methods for manipulating data used by all algorithms. Such an interface is implemented transparently to the algorithms, and different implementations can co-exist. One implementation is the existing Weka3.4 main memory implementation, which is efficient but not scalable. The second implementation, developed in this thesis, is based on the widely used relational database management system DB2 from IBM, and called Weka3.4-DB. It uses DB2 as data storage, which can improve the scalability of all algorithms in principle. In order to provide the flexibility of choosing different levels of scalability and efficiency, Weka3.4-DB keeps the main memory data storage implementation as an option.

The redesigned system architecture is shown in Figure 7.1. Based on the `core` interface from Weka3.4, a general data structure interface has been defined, and any data source that implements this interface can plug into Weka3.4 as a data storage implementation. The basic idea of our DB2 storage implementation is to store the training data in a DB2 database, and only load the training data from the DB2 database to main memory when it is needed. Additionally, methods that require computation on all the data records or a subset of the data records, e.g aggregate methods like `sumOfWeights()`, are implemented using DB2 functionality if possible, without the need to load all the data into memory. This is done by using SQL aggregate functions as explained in Chapter 4. A fundamental strategy employed in the entire DB2 storage implementation is that any method that needs to do computation on all the data will be implemented in a way that avoids loading all the data into main memory at the same time.

Data is now split between a small amount of main memory data structures and a

DB2 database that the major part resides in. We will now look at both parts in more detail.

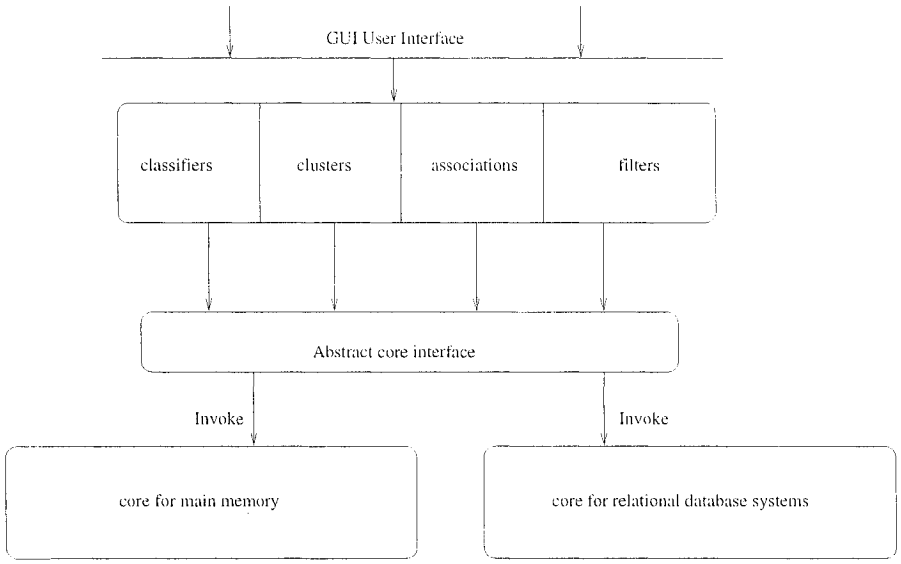

Figure 7.1: Architecture of Weka3.4-DB

7.2.1 Main Memory Data Structure

Among all the classes defined in the `core` package, `Instances` and `Instance` are the two most important ones, since they are defined to represent the training dataset and each individual data record. All other classes are defined either based on `Instances` and `Instance` data structures or are completely independent.

Figure 7.2 shows specifically how `Instances` and `Instance` classes are extended in Weka3.4-DB. The basic idea is to design a system that allows for easy storage of data records in DB2 and fast retrieval of the data records from DB2. Whenever an `Instances` object needs to read data, it knows where to access the corresponding data records in DB2. Therefore, the important issue is how to maintain the training data effectively so that any `Instances` object can access its own copy of the data.

At the beginning of the algorithm, the training data is represented as an `Instances` object in main memory, which contains the attribute information and an index vector. However, the vector does not point to `Instance` objects representing data records. Instead, the data records are stored in the database. The vector contains enough information to retrieve the data records from the database. We will see later what this means. As in the main memory implementation, we only want to create new data records (maintained in the DB2) if different vectors of `Instances` objects require it. However, now `Instance` objects do not all reside in main memory any more. Instead, an `Instance` object is only created (and the data record is loaded to main memory), when it is accessed by the data mining algorithms. For this, we had to slightly redesign the internals of the class hierarchy.

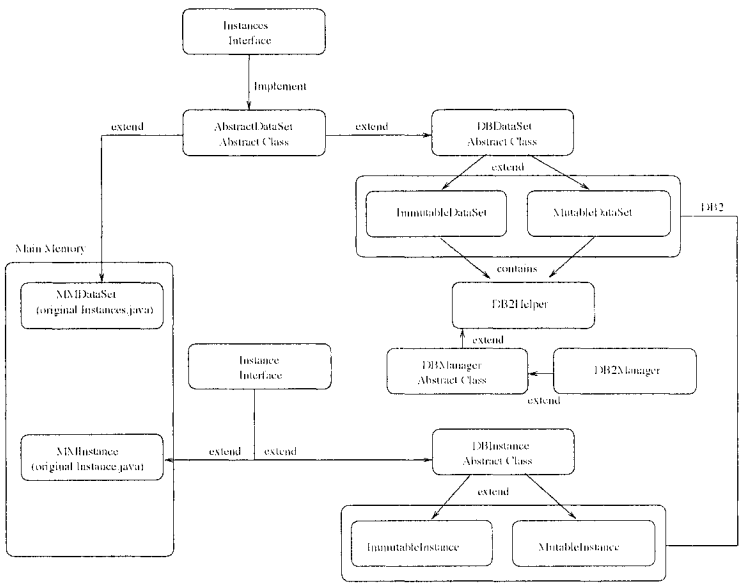

Figure 7.2: Weka3.4-DB: Instances and Instance

At the very top is the `Instances` interface with the same methods described in Chapter 6. The abstract class `AbstractDataSet` implements commonly used variables and methods for all the data storage implementations. `MMDataSet` is the original `Instances` implementation in Weka3.4, and `DBDataSet` is the abstract class for our relational database implementation. It implements commonly used variables and

methods of the two classes: `ImmutableDataSet` and `MutableDataSet`. The difference between `ImmutableDataSet` class and `MutableDataSet` class is that `MutableDataSet` supports all the functions that allow algorithms to change attribute (except weight attribute) values at any point of the computation, while `ImmutableDataSet` does not support those functions (only weight values are allowed to change). The same strategy is applied to the `Instance` class. `MMInstance` is the main memory implementation and is compilable with `MMDataSet`. For the DB2 implementation, there are again an abstract class `DBInstance`, and then an `ImmutableInstance` and a `MutableInstance`. The reason to have two different classes is that any class that extends `DBDataset` has to read data records from the database. If no data record can be modified, then all objects of the class can share the same data records, which means those data records can be stored in a read-only table. Using read-only tables will simplify any kind of read operations and reduce the overhead of loading data records from database. So, whenever the algorithms do not need to change initial attribute values, they can use `ImmutableDataSet` and `ImmutableInstance` to refer to datasets and data records. If they want to change attribute values, they have to use `MutableDataSet` and `MutableInstance`. Furthermore, which type of `Instances` and `Instance` class will be used in the data mining algorithms is defined at runtime, as it is shown in Chapter 8.

`DBManager` is the abstract class that defines basic variables and functions that are used to interact with relational database systems, such as DB2, Oracle and MySQL. `DB2Manager` and `DB2Helper` are two classes that implement all the functions, which can interact with DB2.

7.2.2 Database Implementation

DB2 contains all data records. Tables created in DB2 have to ensure that `Instances` objects can access their own copies of the training data efficiently. There are a number of alternatives for implementing those tables.

- Naive approach: The training data and their weights are first stored in one dataset table, which is the copy of the data for the initial `Instances` object. Whenever a new `Instances` object is instantiated, a new table is created and the training data and their weights are copied to the new table, which is the copy of the data for the new object.

- Duplication approach: The training data is first stored in one dataset table. Each record in the table is marked with the same identifier. indicating that all records belong to the same initial `Instances` object. Whenever a new `Instances` object is instantiated, each of the original records in the dataset table is duplicated, and also stored in the dataset table. All duplicated records receive one common new identifier indicating that they belong to the newly instantiated `Instances` object.

- Lazy approach: The training data is first stored in a dataset table and a common identifier is used to mark those records as belonging to the initial `Instances` object. Whenever a new `Instances` object is instantiated, it shares the data records with the original object by sharing the same identifier in the dataset table. If the new object needs to change a record for computation purposes, a copy of the original record is inserted in the dataset table with a new identifier. This new record will be updated. If all the training data needs to be preprocessed, a table similar to the dataset table is created containing all modified records with a new identifier. Since the attribute values of the training data are not updated by many algorithms, this seems to be the most efficient approach. However, most algorithms do change the weight attributes associated with the data records. Hence, in order for this approach to be more efficient than the duplication approach, we generate two tables. One table contains all standard attributes, one contains the weight attribute. The corresponding records in both tables are correlated by the unique identifiers in both tables. Therefore, whenever a new `Instances` object is initiated, it shares the data records in the dataset table with other `Instances` objects, but it receives a copy of the weights in the weight table for each data record in the dataset table.

The naive approach is very costly, since it creates many new tables, which is one of the most expensive operations of DB2 and should be avoided as much as possible. Even though the duplication approach does not have to create new tables, it still needs to copy the whole training data every time a new `Instance` object is initiated, which will hurt the performance when the training dataset is large. The lazy approach captures the important feature of how most algorithms in Weka3.4 manipulate data. Most algorithms are more likely to change weights but not the attribute values of training data. By default, the training data is not modified at all, and making a separate copy of the weights is enough to distinguish the data records associated with different `Instances` objects. Once the object has made any change on the attribute values of some data records, the updates are flashed back to the dataset table by creating new records with a new identifier, and the object can track those changes by those identifiers assigned to the data records. An extra table is introduced only when dramatic changes have been made on the whole training data, e.g., through preprocessing. Figure 7.3 shows how the tables are implemented in DB2:

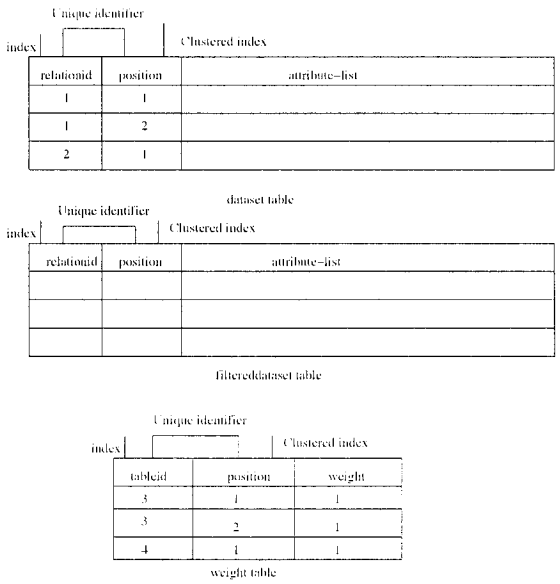

Figure 7.3: Weka3.4-DB tables

- `dataset` table: The `dataset` table stores the training data, i.e the data records/`Instance` objects. Each attribute of the training dataset corresponds to one column in the `dataset` table. The `relationid` column contains identifiers assigned to each data record in the training dataset. The value of the `relationid` column determines which copy of the training data belongs to which `Instances` object. The `position` column contains positions of data records in the training dataset. This is basically the identification of each data record in the training dataset. It is equivalent to the position of an `Instance` object in the vector of `Instance` objects contained in the `Instances` class. Combining `relationid` and `position` can specifically locate a data record associated with a specific vector of `Instance` objects. Therefore, the `relationid` column and `position` column are combined as an unique identifier of the `dataset` table. The reason for having an unique identifier implicitly not a primary key explicitly is that a key constraint would slow down the process of loading the data into DB2, since the database has to check the key constraint on each data record before inserting it into the table.

  The `position` is created for each data record when it is first loaded into the `dataset` table, and corresponds to the position of the data records in the training dataset. The `relationid` is associated with one or more than one `Instances` objects (since `Instances` objects can share training data). Using the `ImmutableDataSet` class, there is only one `relationid`, which is created when the training data is loaded into the database. All objects share the same `relationid` since the training data is shared by all objects and never changed. In the `MutableDataSet` class, the first `relationid` is created when the training data is loaded into the database, and objects share the same `relationid` until some data records are changed. Then the `Instances` object that has changed a record will receive a new `relationid`. This new `relationid` is used for changed records. Each of these records will be inserted into the dataset table, having the same position values but the new `relationid`. The `Instances` object can track both unchanged and changed data records by keeping its own

relationids. That is, an Instances object can have data records with different relationids. Changed records have the new relationid, unchanged records have the same relationids as determined by the Instances object (of which the new object is a copy).

The dataset table has three indices. One is on the unique identifier, one is an unclustered index on relationid, and one is a clustered index on position. The reason for having those indices is that most often relationid and position are the search conditions defined in the where clause of SQL queries.

- filtereddataset table: The filtereddataset table is created only when the data preprocessing has made dramatic changes on the training data. It stores attribute values of the training dataset after the data records in the dataset have been preprocessed. The filtereddataset table in principle has the same schema as the dataset except that it may have different attribute columns if the attributes are altered by the data preprocessing. The filtereddataset table has the same indices as the dataset table.

- weight table: The weight table stores the weights of data records of Instances objects. The tableid column and position column correspond to the relationid column and position column in dataset table. Combining tableid and position can specifically locate weights of training data records associated with a specific Instances object. Therefore, the tableid column and position column are combined as the unique identifier of the weight table. The tableid is associated with one or more than one Instances object. Joining dataset table and weight table on tableid/relationid and position can match the weights with the corresponding training data records. The weight table also has three indexes. One is on the unique identifier, one is an unclustered index on tableid, and one is a clustered index on position. The reason for having those indices is that most often tableid and position are the search conditions defined in the where clause of SQL queries.

7.3 Basic Interaction between `core` and DB2

As it has been discussed in the previous chapters, the interaction between the application program and DB2 is performed through JDBC, and there are a number of implementation alternatives proposed by [36]. The alternative adopted by the `core` of Weka3.4-DB is a loose coupling approach, which has been discussed in Chapter 5. The basic idea is to access the data records in the database directly by executing SQL queries, where the DB2 server is running in a different address space from Weka3.4-DB. More precisely, through the JDBC API, a connection to the DB2 server is built by `core`, SQL queries are constructed in `core`, sent to the database and executed, and the resulting relation is returned through `ResultSet` objects. We have taken this approach for two reasons. First, it is a simple approach with great flexibility. Second, since we want to provide a generalized interface into which new data mining algorithms can be plugged in by non-DB2 experts, this seems to be the most feasible approach.

Here, we will outline the interaction between the `core` data structures and its methods, and the database system. As summarized in Chapter 7, the typical methods of `Instances` in Weka3.4 are responsible for storing data, data access, data manipulation, and calculating statistics of data. Both the `ImmutableDataSet` class and the `MutableDataSet` class are DB2 data storage implementations. Since the `ImmutableDataSet` objects share the same copy of the training data, it makes the underlying implementation of the `ImmutableDataSet` class simpler than the one of the `MutableDataSet` class. Because most of the algorithms in Weka3.4 only access data records without modifying them and the training data will not be changed by any object created during the computation, the `ImmutableDataSet` class becomes more suitable for most algorithms. In the following, we give an overview of how storage, data access, and data manipulation are implemented in the `ImmutableDataSet` of Weka3.4-DB.

7.3.1 Storing Data

When an algorithm needs to load training data from an input file into main memory, our `core` implementation creates some main memory information and stores the training data from the file into the database. First, a main memory `ImmutableDataSet` object is created to store all the attribute type information. The `ImmutableDataSet` object contains a vector of the size of the training dataset. However, in contrast to Weka3.4, this vector does not contain entire data records. Instead the vector entries are all integers representing the positions of the data records. We refer to this vector as an index vector. The data records themselves are stored in the `dataset` table using the generated `relationid` and the appropriate positions. Also weight records with the newly generated `tableid` and positions are added to the `weight` table. All database operations are performed using a special `load` interface.

The index vector is the only memory-based data structure used in the `ImmutableDataSet` class that could grow linearly with the size of the training dataset. The reason for having this index vector is that, in general, `ImmutableDataSet` objects created in the algorithms can reorder the data records during the computation, for instance by sort and re-sample. Having the index vector in memory can make such operations less expensive, since only the positions of data records in the index vector are adjusted and there is no need to reorder the underlying data records in the database physically. And most importantly, it is an efficient way to keep each `ImmutableDataSet` object having its own copy of the training data in order to guarantee the correctness of the algorithms. Therefore, the index vector represents the training data in the correct order for a specific `ImmutableDataSet` object. Most often, any `ImmutableDataSet` object created afterward either shares the same copy of training data with the original object or has its own copy. In the first case, it shares the same index vector, `relationid` and `tableid` (share all data records and weights). In the latter, it creates its own index vector, calls the `add` method to add positions of the corresponding data records into the index vector, and inserts the corresponding weights and positions into the `weight` table with a new `tableid`. But it still shares the same `relationid`

to refer to the training data, since the training data itself never changes. The new copy of the training data is represented by the new index vector, indicating which data records in which order are associated with the `ImmutableInstance` object.

7.3.2 Basic Data Access

When an algorithm needs to access all the training data, it can either call `enumerateInstance()` to get an enumeration of `ImmutableInstance` objects or call `instance(int index)` in a loop to iterate over all `ImmutableInstance` objects. The way of accessing data records from DB2 using the data access methods of `ImmutableDataSet` is to retrieve the position from the index vector and access the corresponding data records in the `dataset` table or `weight` table by specifying the `relationid/tableid` and the position. The following example shows the basic idea of how `instance(int index)` is implemented in `ImmutableDataSet` class.

```
public final Instance instance(int index)
throws SQLException{
 //get position from the index vector
 int position = (Integer)(m_Positions.elementAt(index)).intValue();
 //process the weight and attribute names to the Column array
 Column cols = new Column[numAttributes()+1];
 cols[0] = WEIGHT;
 System.arraycopy(processAtt(m_Attributes),0,cols,1,numAttributes());
 String colnames = makeColumnNameSequence(cols);
 String query =  "SELECT "+colnames+" from weight, dataset"
                 + " WHERE tableid =  "+tid+ " AND relationid =  "+rid
                 + " AND dataset.position =  "+position
                 + " AND weight.position =  "+position;
 ResultSet rs = null;
 double[] values = new double[numAttributes];
 double weight;
```

```
 rs = stmt.executeQuery(query);
 while(rs.next()){
   weight = rs.getDouble(0);
   for(int i=0, n=numAttributes(); i<n; i++){
     rs.getDouble(i+1);
     if (!rs.wasNull()){
       values[i] = rs.getDouble(i+1);
     }else{
       values[i] = Double.Nan;
     }
   }
 }
 rs.close();
 ImmutableInstance instance = new Instance(weight, position, values);
 instance.setDataSet(this);
 return instance;
}
```

In this example, only one data record is in the resulting relation, since the position value can uniquely identify the data record in one copy of the training dataset.

The `enumerateInstance()` method first generates an `enumerate` object. This allows us, in principle, to execute the SQL statement just one time, when this object is generated. A possible statement is:

```
SELECT *
FROM dataset OUTER JOIN weight
ON weight.position = dataset.position
WHERE weight.tid = DS
ORDER BY position
```

which retrieves all records in the order in which they were entered into the database. Since the records are physically ordered this way, this should be a fast scan through the `dataset` and `weight` tables. However, since the records have to be obtained through the `enumerate` interface, they have to be returned according to the positions stored in the position vector (which might be arbitrary). Hence, we must use a scrollable JDBC `ResultSet` with which we can jump arbitrarily to any position. When the next entry of the vector indicates a position p, we can call `ResultSet.absolute(p)` to retrieve the correct record (in order to retrieve the correct position with `absolute` we have to use an outer join. This guarantees that even if not all records are present in the position vector and the `weight` table, they will nevertheless be selected. Hence `absolute(p)` guarantees to provide the data record with position p.). Alternatively, we can use for `enumerateInstance()` the same mechanism as for `instance(index)`, submitting one SQL statement for each record to be retrieved. Only the second alternative is implemented in the `ImmutableDataSet` class.

It is not immediately clear which of the two alternatives is faster: submitting one SQL statement for each record to be accessed or having a single SQL that retrieves all tuples which are then accessed in an arbitrary way through `ResultSet` primitives. So far, we tested the case in which the order in the position vector is the same as the order in which the records are stored in the `ResultSet`. In this case, one SQL statement for the entire data set outperforms by far individual SQL statements for each record. We are currently performing tests for the following two cases. (i) The position vector performed some sorting and hence does not follow anymore the position order in the database. In this case, arbitrary access of records in the `ResultSet` might be very slow. (ii) The size of the position vector of the specific `ImmutableDataset` object is much smaller than the size of the entire data set. This can occur, e.g., in decision trees, where in later iterations of the decision tree construction only subsets of the entire data set are analyzed. In this case, the number of retrieved records might actually be small, and hence the execution of the few SQL statements could be faster. The further investigation is underway.

7.3.3 Basic Data Manipulation

An algorithm can manipulate data through methods such as `add(Instance instance)` and `delete(int index)`. Since the training data has been stored in the database by the `load()` method and all the `ImmutableDataSet` objects share the same copy, the `add(Instance instance)` methods adds references to data records by adding the corresponding position values to the index vector and inserting the weight records into the `weight` table. No data record is added into the `dataset` table. Similar to the `add(Instance instance)` method, `delete(int index)` method deletes the data records by removing the corresponding position values from the index vector and deleting the weight records from the `weight` table. No data record in the `dataset` table is deleted. Updating any change on the index vector and the `weight` table can guarantee the index vector represents the data records associated with the `ImmutableDataSet` object in the correct order and the `weight` table contains all the data records associated with the `ImmutableDataSet` instance. When the `ImmutableDataSet` object calls its own methods, executing those methods can only affect its own copy of the training data, more precisely, the index vector and the `weight` table. Note that the data records in the `weight` table may not be ordered as in the index vectors. However, this does not play any role since either they can be ordered upon retrieving them from DB2 or the order is not important.

7.4 Moving Functionalities into DB2

Whenever the training data is accessed, it must be loaded record by record into main memory. This removes any memory constraint but increases response time tremendously. Hence, avoiding to load data records to memory whenever possible is highly desirable. One way to address this is to push some functionalities into DB2. This leads to a tighter integration with the underlying database, which improves performance.

Depending on how tightly a strategy couples Weka3.4-DB to the relational DBMS,

different strategies result in different efficiencies of the implementation of Weka3.4-DB. We first look at some basic methods of the `Instances` interface.

Strategy I

This is the strategy described so far. The training data is loaded one record at a time. All the attributes of one data record are retrieved from DB2 and the data record is represented as an `ImmutableInstance` object. The idea is to store the data record as an `ImmutableInstance` object in main memory to complete the current step of the computation, then let the Java garbage collector collect the unused `ImmutableInstance` object, and continue to load the next data record into main memory for the next step of the computation. Since only a few `ImmutableInstance` objects remain in the main memory at any time (the garbage collector collects the unused objects periodically), strategy I can provide high scalability. But the computation may be slow, since it has to retrieve the data records one by one.

Strategy II

Strategy II employs the same approach as strategy I, but only retrieves the attribute values that are involved in the computation. This saves space and leads to less communication overhead between Weka3.4-DB and the database. Using this approach, however, the developer of the data mining algorithms has to indicate which attributes are needed. Hence, this strategy requires an extension of the current `Instances` interface, which is not done in the current `core` implementation.

Strategy III

Strategy III provides the best optimization. It employs the same approach as the semitight coupling proposed by Hand and and Kamber [18]. The idea is to use some standard functions within the database system, and to provide an interface to the data mining algorithms to call them. Compared to the other strategies, this provides the maximum scalability and efficiency for Weka3.4-DB, and should be applied as much as possible.

Most often, the methods that calculate some statistics of the training data can be

optimized by applying this strategy. The typical easy example is sum. When the main memory implementation needs to sum weights of the training data, it iterates over the whole vector of Instance objects in the Instances class. If we used the same implementation for ImmutableDataSet class, we would have to retrieve each data record from DB2, load it in memory to retrieve the weight for summation. Instead, we achieve the sum computation by applying the aggregate function sum of SQL on DB2. For instance, the sumOfWeights method in ImmutableDataSet class is:

```
public final double sumOfWeights()
throws SQLException{
 String query = "SELECT sum(weight) FROM weight where tableid = "+tid;
 ResultSet rs = null;
 rs = stmt.executeQuery(query);
 double sum=0;
 while(rs.next()){
  sum=rs.getDouble(1);
 }
 rs.close();
 return sum;
}
```

A more complicated method is meanOrMode(Attribute att). Similar to the sum, instead of loading the data records from DB2, we achieve the meanOrMode(Attribute att) method by using the following SQL statement to calculate the mean of a numeric attribute.

```
public final double meanOrMode(Attribute att)
throws SQLException{
  if (att.isNumeric()){
    double result = 0;
    String query = "SELECT sum ( weight * "+att.name()
```

```
                    +") FROM weight, dataset"
                    +" WHERE tableid = "+tid
                    +" AND relationid = "+rid
                    +" AND weight.position = dataset.position"
                    +" AND "+att.name()+" IS NOT NULL"
    ResultSet rs = null;
    rs = stmt.executeQuery(query);
    while(rs.next()){
       result = rs.getDouble(1);
    }
    ..........
  }else if (att.isNominal()){
    ..........
  }
}
```

Another typical function is `sort`. Whenever the method needs to order the training data based on some attribute value, instead of applying any main memory sorting algorithm, which requires to retrieve records possibly multiple times from DB2, we implement `sort` by using a SQL statement, selecting the position and using the `order by` clause on DB2. For instance, the `sort` method in `ImmutableDataSet` class is shown in the following.

```
public final void sort(Attribute att)
throws SQLException{
 String query =  "SELECT weight.position FROM weight INNER JOIN "
                +"dataset ON weight.position = dataset.position "
                +"WHERE tableid = "+tid
                +" AND relationid = "+rid
                +" ORDER BY "+att.name();
 ResultSet rs = null;
```

```
  rs = stmt.executeQuery(query);
  FastVector result = new FastVector;
  while(rs.next()){
    result.addElement(new Integer((Int)rs.getDouble(1)));
  }
  m_Positions = result;
  rs.close();
}
```

Note that only the weight table represents the data records associated with one particular `ImmutableInstance` object, so it needs to join the `weight` table with the `dataset` table in order to get the correct order.

These optimizations, compared to strategy II, do not need a change on the `core` interface, as long as they implement the methods of the `Instances` interface. We have implemented them in the `ImmutableDataSet` class.

7.5 Optimizations outside the `core`

So far, we have only described how we have re-implemented the `core`. The implementation is transparent to the existing algorithms. All data mining algorithms and filters, as described in Figure 6.2 build on top of `core`. Hence, they can now take advantage of the new implementation that is unrestricted in size. However, theses algorithms themselves can be implemented in a smarter way if they are aware of memory limitations. We will discuss our optimization of the filter algorithms and the logistic regression algorithm.

7.5.1 Data Preprocessing: Filters

The general idea of data preprocessing has been discussed in Chapter 3. In Weka3.4, the data preprocessing classes, called *filters*, are implemented in the `filters` interface, which is independent from the `core` interface. Those filters can be used independently to preprocess the training data before running the algorithms or can be invoked inside the algorithms. We have looked at three filters in particular.

1. *ReplaceMissingValues*: replace all missing values of nominal and numeric attributes with the modes and means of the training data.

2. *NominalToBinary*: convert all nominal attributes into binary numeric attributes, such that an attribute with k values is transformed into k binary attributes, and binary attributes are left binary.

3. *RemoveUseless*: remove attributes that do not vary at all or that vary too much, which applies to two kinds of attributes: constant attributes that do not vary at all; attributes that exceed the maximum percentage of the variance parameter.

In Weka3.4, filters are implemented to filter the training data records one by one. A filter stores all the filtered training data in a queue. Applying the same implementation in Weka3.4-DB would load all the training data into main memory and write the new values back to DB2. This adds considerable overhead to the algorithm. And, more importantly, it introduces a main memory constraint by using the queue data structure. In order to make the data processing step feasible and efficient for large datasets, extra methods are created to allow those filters to be run without loading any data records into main memory. In other words, the strategy III for database-oriented implementation is employed to move the functionalities of those filters into the database. The basic ideas of how those filters are implemented are described in the following:

- *ReplaceMissingValuesFilter*: precompute modes of nominal attributes and means

of numeric attributes with SQL aggregate functions and use *update* SQL statements to replace missing values with modes and means.

- *NominalToBinaryFilter*: adding attributes means changing the table schema by adding columns. Therefore, the heuristic is that if the number of new columns does not exceed the threshold, new columns are added to the `dataset` table and the corresponding values for new columns are updated; otherwise, the `filtereddataset` is created with unchanged columns and new columns, unchanged values are copied from `dataset` to `filtereddataset`, and the corresponding values for new columns are inserted. The heuristic helps minimize the cost of the operation, since creating a table is a very expensive operation in the database, and adding many columns to a table also costs a lot.

- *RemoveUseless*: removing attributes can be done on the attribute information stored in the `ImmutableDataSet` class without touching the database, since the SQL statements used to load data records from DB2 look at the attribute information in the `ImmutableDataSet` class to determine which attribute to retrieve. The criterion used to determine which attributes are useless is the number of attribute values. For nominal attributes, the number of values is calculated using the attribute information stored in main memory. For numeric attributes, the number of values is calculated by counting the distinct values in the database. For those that either have a single attribute value or too many attribute values that exceeds the threshold, they will be removed from the attribute information stored in main memory.

As a result, none of the filters requires to read data records into main memory, improving scalability and efficiency.

7.5.2 Logistic Regression

The general idea of logistic regression has been discussed in Chapter 3. The logistic regression algorithm implemented in Weka3.4 is a penalized logistic regression with

a default ridge parameter $1.0E-8$. It is based on the paper of le Cassie and van Houwelingen [8], but without estimating the ridge parameter. The ridge parameter can be specified as user input. If no user input is given, the default value is taken for the computation. Since there is no need to apply any method to choose the ridge parameter, the only step left is to compute the coefficients for the attributes. Therefore, the logistic regression algorithm simply goes through the training data and builds the logistic regression model by calculating the coefficient for each attribute at each step.

Suppose there are k classes for n data records with m attributes and the coefficient matrix B is an $m*(k-1)$ matrix. The log-likelihood function applied in the logistic regression algorithm is:

$$L = -\sum_{i=1..n}\sum_{j=1..(k-1)}(Y_{ij} * ln(P_j(X_i))) + (1 - (\sum_{j=1..(k-1)} Y_{ij})) * ln(1 - \sum_{j=1..(k-1)} P_j(X_i))$$

$$+ridge * (B^2)$$

where the probability of class j except the last class is

$$P_j(X_i) = exp(X_i B_j)/((\sum_{[j=1..(k-1)]} exp(X_i * B_j)) + 1)$$

and the last class has probability

$$1 - (\sum_{[j=1..(k-1)]} P_j(X_i)) = 1/((\sum_{[j=1..(k-1)]} exp(X_i * B_j)) + 1)$$

The goal of the algorithm is to find the matrix B for which L is minimized. A Quasi-Newton method is used to search for the optimized values of the $m*(k-1)$ variables. Before the optimization procedure is used, the matrix B is squeezed into a $m*(k-1)$ vector. Once the matrix B is computed, the probability of any data record that belongs to a certain class can be computed by the probability functions as above. Although the original logistic regression does not deal with weights for the attributes, the implementation is adjusted to handle the weights.

The basic steps of logistic regression implemented in Weka3.4 are shown in the following:

```
Logistic (training dataset, testing dataset)
  Store training dataset into main memory as an Instances object
  filter the training dataset using ReplaceMissingValues
  filter the training dataset using RemoveUseless
  filter the training dataset using NominalToBinary

  Normalize the training dateset and store the
  normalized training data in a 2-dimensional array
  Compute the coefficients by minimizing the
  log-likelihood function based on the 2-dimensional array

  Evaluate the logistic regression model on the filtered
  training dataset
  Evaluate the logistic regression model on the filtered
  testing dataset incrementally
```

The filters that are invoked inside the algorithm have been re-implemented as discussed in the previous section, and hence do not impose any scalability restriction any more.

However, the logistic regression algorithm creates a 2-dimensional main memory structure that is, in fact, as large as the entire dataset. Even though logistic regression can achieve good scalability using the optimized filters, it is limited by the 2-dimensional array constraint. This shows that developers must be aware of space limitations. However, we can help them in developing scalable implementations by providing adequate support. In the above example, normalizing data and the computation of the normalized data seem to be a standard approach usable in various algorithms. Therefore, we offer an extra interface that allows normalizing data within the database. It is implemented using DB2. That is, the normalization uses SQL queries and the

result is stored in DB2. The developer can use it without knowing that a database implementation is used.

The two alternatives lead to two variations of the implementation of computing the logistic regression model. They differ in how they are implemented in the algorithm.

- variation 1: load the training data from DB2 with strategy I, normalize the training data and store the normalized training data in the 2-dimensional array (existing algorithm).
- variation 2: create extra normalization methods in the `ImmutableDataSet` class, strategy III calls these methods from the logistic regression algorithm; load the normalized training data from DB2 with strategy I.

7.6 Strategies for Performance Optimized JDBC Application

There exist several ways to generally speed up JDBC applications. Some of these strategies have been applied in the `DB2Manager` and `DB2Helper` classes, and are described in the following.

- Connection: Connection management is important for application performance. Creating a connection to a database server is expensive and it is even more expensive if the server is a remote server. A simple and easy strategy is to open one connection and share it in a serial fashion among multiple statement objects. Hence, in the `core` implementation of Weka3.4-DB, the connection is opened by a `DB2Manager` object, which is a static field in `DBDataSet`. All the objects of either `ImmutableDataSet` or `MutableDataSet` share the same `DB2Manager` object, and hence share the same connection. Since this strategy only works with the single-user mode, this DB2 storage implementation only

supports single-user mode. If multiple-user mode is requested, the DB2Manager object should maintain a connection pool.

- Transaction Atomicity: In general, a transaction represents one logical unit of work or piece of code that either executes entirely and commits, or it does not execute at all, aborting all the work done so far. Initializing a transaction and terminating it (commit or abort) can be quite time consuming. Using JDBC default, each SQL statement executes as a single transaction using `autocommit on`. In `autocommit off`, the program decides which statements belong to a transaction by setting explicit commit statements when a transaction should terminate. Using `autocommit on` gives poor performance when multiple statements are to be executed one after another, because commit is issued after each statement by default. This reduces performance by issuing unnecessary commits. Therefore, in our `core` implementation, the autocommit is set to false and the `commit()` method is called explicitly after a set of related statements.

- Transaction Isolation Level: The isolation level represents how a database maintains data integrity against problems like dirty reads, phantom reads and non-repeatable reads that can occur due to concurrent transactions. Different isolation levels have different impacts on the performance. A stricter isolation level has worse performance in terms of execution time. This is true because the database uses locks to prevent different transactions to access the same data records. The stricter the isolation level, the more locks must be requested, hence the more overhead occurs due to locking. The default setting is `read_committed`, which means a transaction can only read the data from the database when the data has been committed by other transactions. Any isolation level that is lower than the default is likely to be faster, and the opposite will probably be slower. In our `core` implementation, the isolation level can be lowered to `read_uncommitted` level, since there are no concurrent transactions and execution is sequential. Hence, there is never uncommitted data. That is, although we set the isolation level to uncommitted, leading to extremely low locking overhead, we achieve the same effect as `read_committed`.

- Statement: There are three types of statement interfaces in JDBC to represent a SQL query and execute that query: `Statement`, `PreparedStatement` and `CallableStatement`. `Statement` is used for static SQL statements with no input and output parameters. `PreparedStatement` is used for dynamic SQL statements with input parameters and `CallableStatement` is used for dynamic SQL statements with both input and output parameters. `PreparedStatement` gives usually better performance compared to `Statement` because it is preparsed and pre-compiled by the database once for the first time and then it reuses the parsed and compiled statement afterward. Because of this feature, it can significantly improve the performance when a statement executes repeatedly, since it reduces the overload incurred by parsing and compiling. `PreparedStatement` has been applied in our `core` implementation. For instance, the insert query used for inserting the data records from the training dataset is executed as many times as the size of the training dataset. A `PreparedStatement` created for such a query helps save the overhead of parsing the same statement multiple times. Hence, in the `core` implementation, a `PreparedStatement` object with the batch update feature is created for inserting the data records into the database. The attribute values of each data record are added to the batch of the `PreparedStatement` object. Whenever the attribute values of a predefined number of data records have been added into the batch, the `PreparedStatement` is executed and all the attribute values are inserted into the database.

  There exist further possibilities to use the `PreparedStatement`. For instance, a function that offers to select certain attributes of all data records can be implemented with the `PreparedStatement` in the following way. A `PreparedStatement` object used for retrieving the attributes of data records from the database is created before the loop. During the loop the `PreparedStatement` object is executed repeatedly to retrieve the attribute value of each data record step by step.

Chapter 8

Performance Evaluation

8.1 Experiment Design

8.1.1 Goal and Setup

The major goal of the experiment section is to show that applying a DB2 storage implementation can improve the scalability of existing algorithms significantly without modifying the results of the algorithms. We use logistic regression as a demonstration. Furthermore, we compare the performance of different versions of logistic regression to show the effectiveness of different strategies for improving the scalability.

The experiments are conducted on a Linux machine with dual CPUs at CISTI, NRC. The kernel version of the Linux machine is 2.4.18-26.8.0, the CPU model is Intel (R) XEON (TM) MP CPU 1.50GHz, the CPU frequency is 1492.183 and the total memory is 3098684 KB. DB2 8.1.0 server, set up by Greg Kresko from CISTI, is running on the same machine with a fixed configuration. Appendix A shows how the DB2 server is configured.

8.1.2 Datasets

ARFF

The input dataset files are required to handle *ARFF* (Attribute-Relation Format File) format [41]. An ARFF consists of a header section and a data section. The header section contains a relation name declared by token *@relation* and attribute information declared by token *@attribute*. The attribute types supported by Weka3.4-DB are numeric and nominal. The numeric attributes can be defined as real or integer numbers and the nominal attributes have a list of possible nominal values. The data section contains all the data declared by token *@data*. Each data record resides on one line of the file. Attribute values of each data record are separated by comma and missing values are represented by question marks. Appendix B shows a sample ARFF.

Synthetic Datasets

The first experiment is based on synthetic datasets, which are generated by a data generator for classification tasks. The data generator is originally from IBM Almaden Research Center [1]. We slightly modified it to generate ARFF format dataset files. The data generator only generates numeric type attributes and nominal type class attributes. The numeric values are randomly generated integer values. The nominal class values are binary numbers with user-defined percentages. In the experiment, ten training datasets between 10,000 to 100,000 data records and one testing dataset with 5000 data records are used. Each dataset has 50 attributes and 1 class attribute without missing values.

Real Datasets

The second experiment is based on real datasets, which are derived from one AVIRIS (Airborne Visible/Infrared Imaging Spectrometer) dataset. It is originally from JPL (Jet Propulsion Laboratory, California Institute Technology) and extensively corrected by CCRS (Canadian Center for Remote Sensing, Natural Resources Canada). The AVIRIS data is hyperspectral data that was captured by NASA/JPL AVIRIS

sensor over Cuprite, Nevada on June 12, 1996 (19:31UT). Paul Budkewitsch from CCRS released the AVIRIS dataset with 300,0000 data records and 170 attributes. For more background information about the dataset, please refer to [34].

In the experiment, four different AVIRIS datasets are generated by randomly sampling the original AVIRIS. This sampling was done by Glen Newton from CISTI (Canada Institute for Science and Technical Information) of NRC (National Research Council of Canada). Each attribute value represents a reflectance at an interval of 0.12nm wavelength in the range of 0.428 to 2.5 without 1.4 and 1.9nm. The class attribute value represents if a certain mineral is present or not. There are three target minerals associated with the original AVIRIS dataset, which are alunite (AL), kaolinite (KA) and buddingtonite (BU). In our experiment, we will look at kaolinite. The class labels of the generated four training datasets show if kaolinite (KA) is present or not with a threshold of 25%. In the experiment, four training datasets contain 12669, 19712, 35055 and 78592 data records respectively, and one testing dataset has 3224 data records. Each dataset has 168 numeric attributes and 1 nominal class attribute without missing values.

8.1.3 Logistic Regression

Weka3.4-DB runs under the original main memory `core` implementation and our DB2-based `core` implementation on the same machine as the DB2 server resides on, which is lightly loaded. In order to easily refer to different implementations, we call the first one the memory storage implementation and the second one the DB2 storage implementation. The two implementation are started with the following commands.

```
java -Xms64M -Xmx64M -Dds=mm weka.classifiers.functions.Logistic
     -t data/synthetic10k.arff -T data/synthetic.arff
java -Xms64M -Xmx64M -Dds=dbi weka.classifiers.functions.Logistic
     -t data/synthetic10k.arff -T data/synthetic.arff
```

In order to make a reasonable comparison between the two implementations without having an extremely long running time, the memory size that can be used by both implementations is constrained to 64MB. The `-Dds` option is used to set up the storage implementation: `mm` means memory storage implementation and `dbi` means DB2 storage implementation with `ImmutableDataSet` (since logistic regression will not change any attribute values after the filter operations). The *-t* option is used to set up the training dataset and the *-T* option is used to set up the testing dataset. All experiments use the default values for the options for logistic regression.

Besides the implementation of logistic regression based on the memory storage, logistic regression based on the DB2 storage has been implemented with three different versions for the empirical study. Each version has adopted different strategies discussed in the last chapter to achieve a database-oriented implementation.

- version 1:
    1. no modification on the implementation of logistic regression, i.e. only use the basic interface of `core` to access and manipulate data, (variation 1 of logistic regression described in Section 7.5.2)
    2. filters are not optimized and only use the basic `core`.
- version 2:
    1. no modification on the implementation of logistic regression, (variation 1 of logistic regression)
    2. use the modified filters
- version 3:
    1. use the enhanced interface, and adjust logistic regression (variation 2 of logistic regression described in Section 7.5.2),
    2. apply the filters

Version 1 is a naive approach. It only uses the basic interface of `core` and does not optimize the algorithm itself to improve scalability and efficiency. In contrast to the other two versions, version 1 is supposed to be the worst.

Version 2 and version 3 move some of the computation of the algorithm into the database, which makes them more efficient. Although, version 2 has a memory constraint, its use of an array makes the data access faster than loading data records from the database.

Since version 3 removes all the memory constraints, it can achieve the highest scalability among all the versions. However, the performance of version 3 may suffer from the overhead caused by loading the data records into main memory.

The different versions of logistic regression in Weka3.4-DB are maintained by the CVS repository and will run under DB2 storage implementation with the same command. In the experiment, we call the implementation of logistic regression based on the memory storage as main memory version of logistic regression, and the implementations of three versions of logistic regression based on DB2 storage as version 1 of logistic regression, version 2 of logistic regression and version 3 of logistic regression.

8.2 Experimental Results

8.2.1 Experimental Results for Synthetic Datasets

Results for version 1 were very bad, with long execution times even for small datasets. Hence, we do not discuss it further. The experimental results of version 1 are not shown in the following figures.

The results for synthetic datasets of running the main memory version of logistic regression and version 2 of logistic regression are shown in Figure 8.1.

Both implementations produce the same classification results on all the datasets. The figure shows that the main memory version of logistic regression runs out of memory

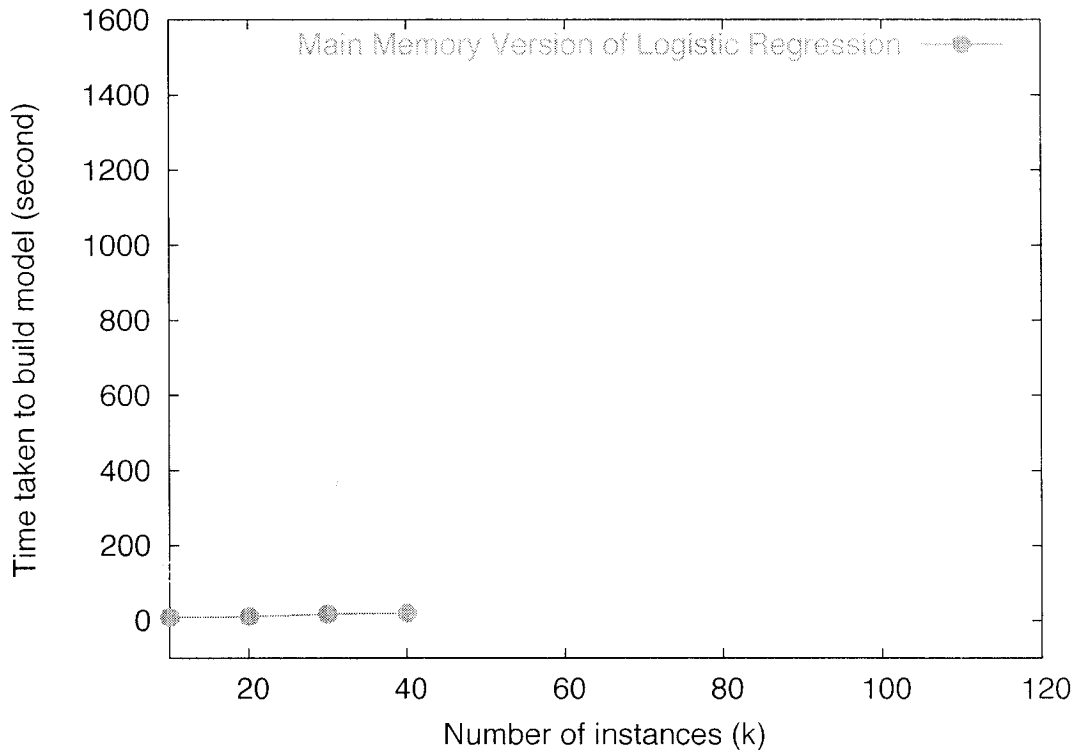

Figure 8.1: Synthetic Datasets: Main Memory vs V2

on the training dataset with 50,000 data records, and the version 2 can run with the training datasets up to 100,000 data records.

Version 2 can handle a bit more than twice the number of data records as the main memory version. It can't increase any further since the 2-dimensional main memory data structure of logistic regression hinders further scalability. The execution time of version 2 grows linearly with the size of the training dataset. It also increases linearly in the main memory version but with a small coefficient, which is nearly negligible. The reason is that version 2 retrieves each data record individually from the database while the main memory version loads all the data records in main memory and hence has very fast access.

However, version 2 runs out of memory for a dataset with 110,000 data records. The

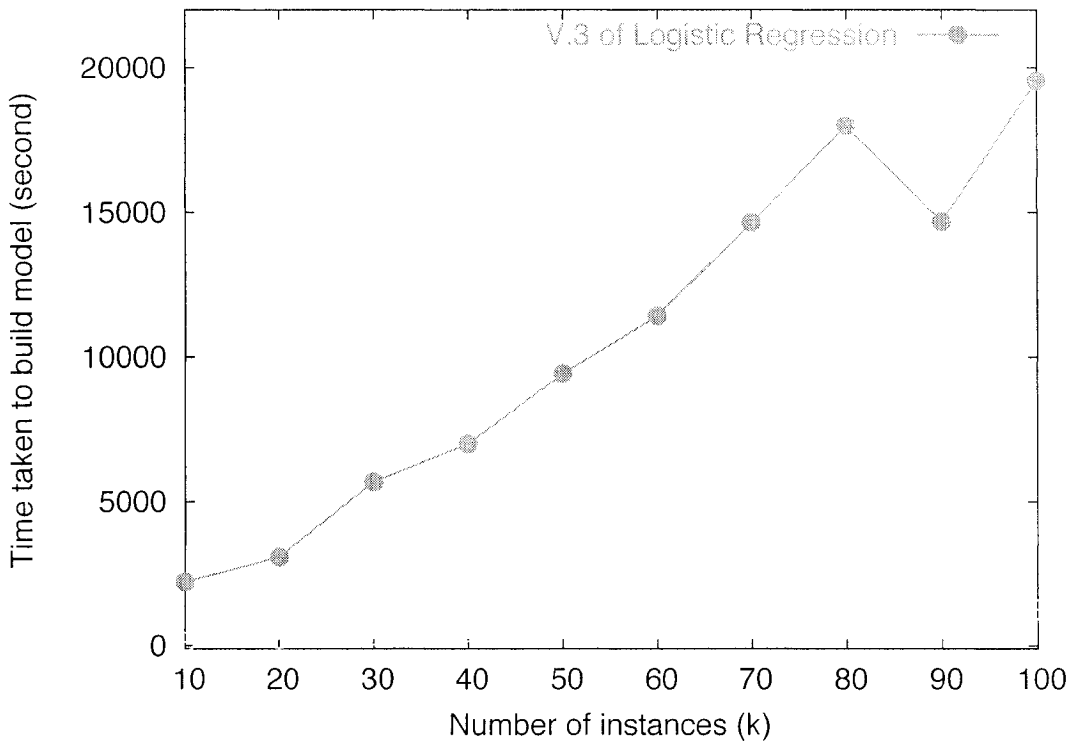

Figure 8.2: Synthetic Datasets: V2 vs V3

experimental results of running version 3 of logistic regression are shown in Figures 8.2 and 8.3 Version 3 is running slower than version 2, because version 2 uses the 2-dimensional array to store the normalized training data, while version 3 has to load the normalized training data from DB2. However, scalability is much improved. We ran the experiment up to a training dataset with 600,000 data records with version 3 without memory problems. The response time increased with the size of the training dataset for the entire experiment. Version 3 runs out of memory on the training dataset with 700,000 data records, because the index vector becomes a memory constraint, since it grows linearly with the size of the training dataset.

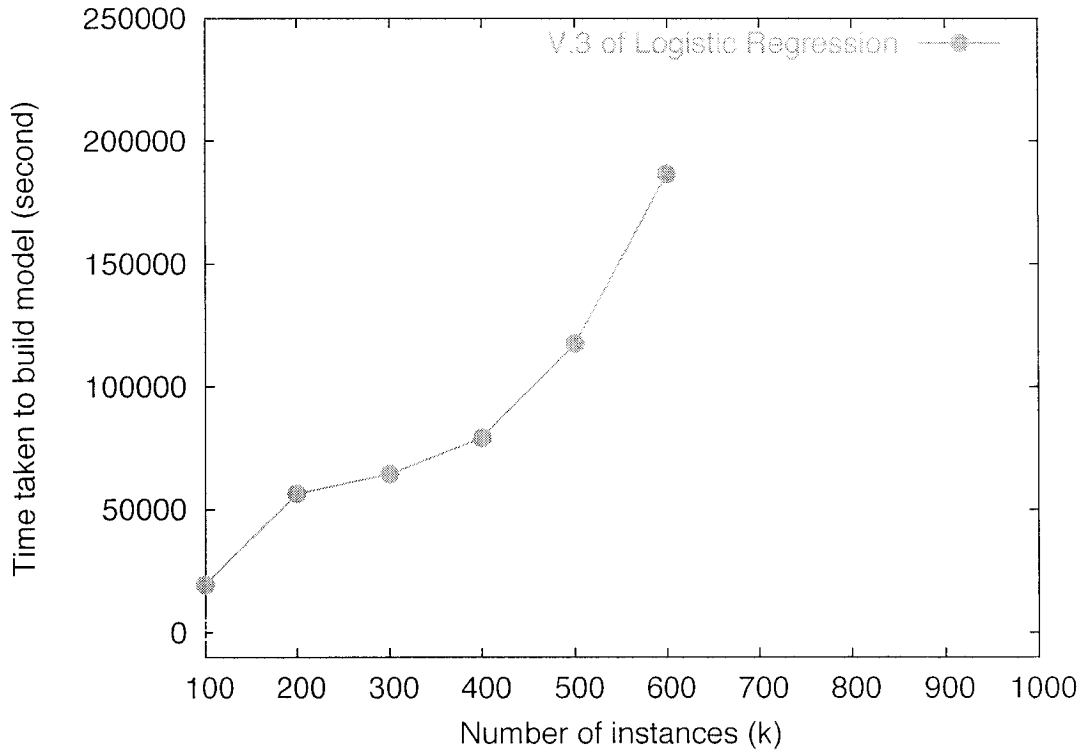

Figure 8.3: Synthetic Datasets: V2 vs V3

8.2.2 Experimental Results for Real Datasets

The experimental results of running the main memory version of logistic regression and version 2 of logistic regression for the AVIRIS datasets are shown in Figure 8.4.

Both implementations produce the same classification results on all the datasets. The figure shows that the main memory version of logistic regression runs out of memory on the training dataset with 19712 data records, and version 2 of logistic regression runs out of memory on the training dataset with 78592 data records. Since the dataset has more attributes than the synthetic dataset, more memory is needed to store each dataset. It shows that the scalability of version 2 of logistic regression does suffer from the memory constraint caused by the 2-dimensional array. However it can handle 4 times as many data records than the main memory version. Response

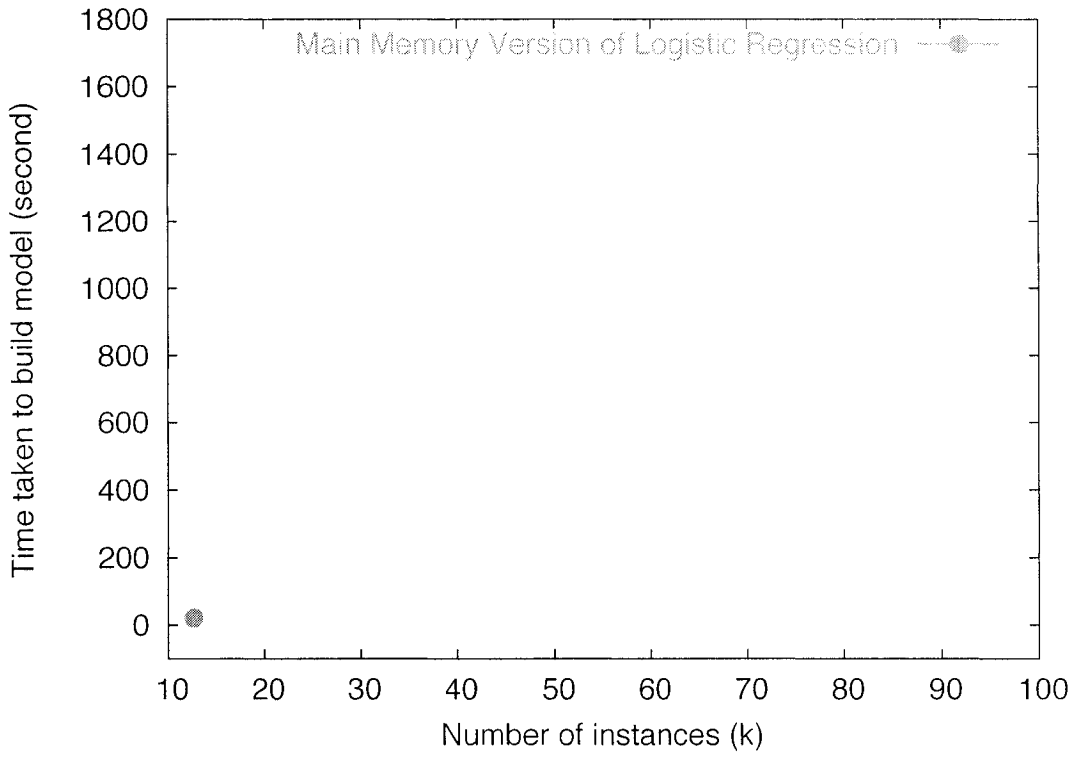

Figure 8.4: AVIRIS Datasets with 169 attributes: Main Memory vs V2

times are higher for both versions compared to the synthetic dataset due to the higher number of attributes, which requires more computation.

The experimental results of running version 3 of logistic regression are shown in Figure 8.5. Version 3 of logistic regression can scale up to larger datasets than version 2 as expected, because version 3 has removed almost all the memory constraints from its implementation. Note that the experiment stops only because we did not have access to datasets with a larger number of data records.

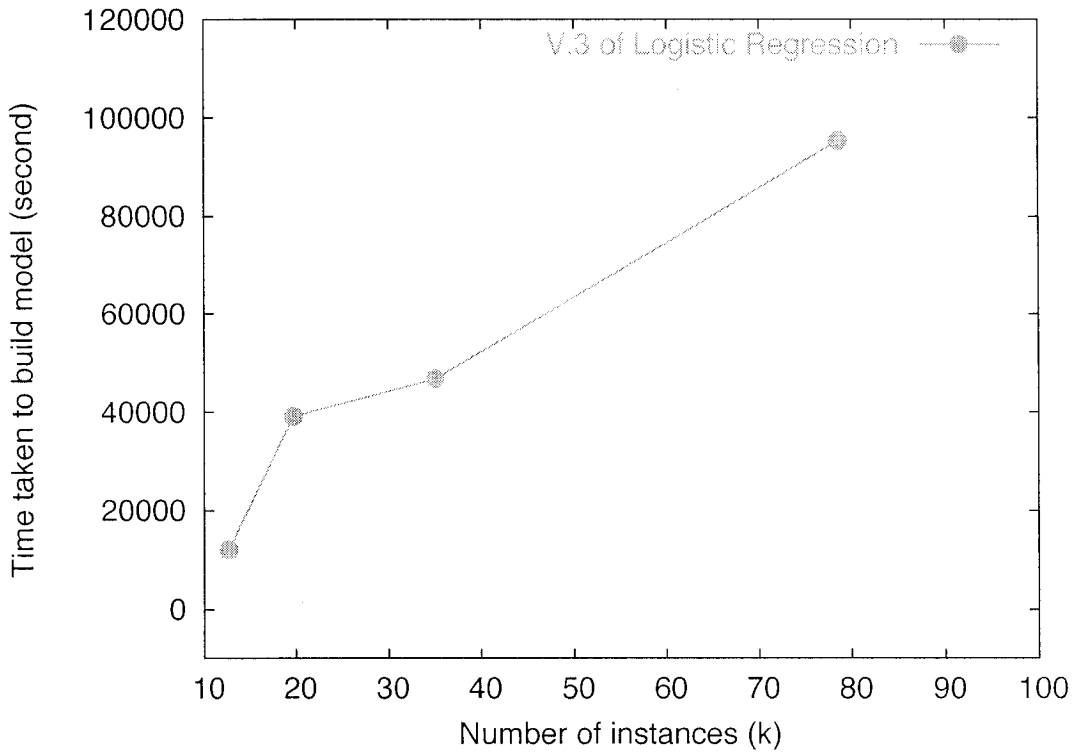

Figure 8.5: AVIRIS Datasets with 169 attributes: V2 vs V3

8.2.3 Analysis

Even though there is lack of clear experimental results for version 1 of logistic regression, our preliminary tests show that a simplistic DB2 storage implementation is not enough and has even less scalability than the original main memory implementation. In version 2, filters have been re-implemented by moving functionality into the database. A similar approach is applied when improving version 2 to version 3, where extra interfaces have been introduced to provide methods that can get rid of the 2-dimensional array. Therefore, introducing extra interfaces that are implemented within the database does improve the scalability of the algorithm. If more functionality is implemented within the database, more scalability can be achieved. The results show clearly that version 3 can achieve higher scalability than version 2

because the normalization function is implemented within the database and there is no need to apply it on the training data in main memory; therefore the 2-dimensional array holding the normalized training data is removed. Since the strategy that is used to improve scalability is similar to the semitight coupling proposed by Han [18], the performance results further prove that a tighter coupling approach can achieve better scalability.

Since improving the scalability is the major goal of this study, the efficiency is not an important concern when running the logistic regression on DB2 storage implementation. It is obvious that the logistic regression on memory storage implementation always has the best execution time. But, even though version 3 of logistic regression on DB2 storage implementation is slower than all the other implementations, it still has a reasonable response time with respect to the size of the training dataset.

There are a couple of special factors in this experiment that may affect the execution time of all the implementations when the experiment setting is different.

- datasets: training datasets do not have missing values and nominal attributes, therefore, the effect of filters dose not count much into the execution time. Execution time will increase when filters really do some serious work.
- network: Weka3.4-DB is running on the same machine as the DB2 server. Therefore, the effect of network traffic does not count into the execution time. Execution time will increase when Weka3.4-DB is running on a different machine.

Chapter 9

Conclusion

In this thesis, we extended Weka3.4 successfully to handle large datasets that can't fit into main memory. Weka3.4-DB is implemented to store the data into and access the data from DB2 with a loose coupling approach in general. Additionally, a semi-tight coupling is applied to optimize the data manipulation methods by implementing core functionalities within the database. Based on the DB2 storage implementation, Weka3.4-DB achieves higher scalability, but still provides a general interface for developers to implement new algorithms without the need of database or SQL knowledge.

The experiment on logistic regression demonstrates that Weka3.4 can be extended to handle large datasets that do not fit into memory with a reasonable execution time. This proves that using relational database systems is a strategic and practical solution for solving the problem of handling large datasets in data mining tasks.

However, there are still a number of issues that need to be addressed in future work before achieving the final goal, that is, to claim that Weka3.4-DB is a memory-constraint free package that can handle arbitrary large datasets.

- The index vector can become a constraint when the size of the training dataset is extremely large. The optimal solution would be to resolve the problem without using memory data structures but still achieve similar performance.

- For more complicated algorithms, such as decision trees (which split the training dataset recursively), further specialized interfaces have to be provided to achieve good performance.
- Other implementation alternatives could be applied in order to find the best strategy to optimize the performance. For instance, a local disk can be used to cache frequently used data records.
- Furthermore, the interaction between Weka3.4-DB and the database can be adjusted, starting from further JDBC related optimizations to using the stored procedures technology of the database.

Appendix A

DB2 Server Configuration

The DB2 server is tuned with following parameters:

1. Application heap size (applheapsz) that defines the number of private memory pages available to be used by the database manager on behalf of a specific agent or subagent: 10,000 pages (4KB)

2. Query heap size (query_heap_sz) that defines the maximum amount of memory that can be allocated for the query heap: 10,000 pages (4KB)

3. Application support layer heap size (aslheapsz) that defines the maximum amount of memory that can be allocated for the communication buffer between the local application and its associated agent: 1000 (4KB)

4. Transaction log file size (logfilsiz) that defines the size of each primary and secondary log file: 50,000(4KB)

5. Number of transaction files primary (logprimary) that defines the number of primary log files that can be used for recovery: 100

6. Number of transaction files secondary (logsecond) that defines the number of secondary log files that can be used for recovery: 100

Appendix B

ARFF Example From Weka3.4

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature real
@attribute humidity real
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,85,85,FALSE,no
sunny,80,90,?,no
overcast,?,86,FALSE,yes
rainy,70,96,FALSE,yes
rainy,68,80,?,yes
rainy,65,70,TRUE,no
overcast,?,65,TRUE,yes
sunny,72,95,FALSE,no
?,69,70,FALSE,yes
rainy,75,80,FALSE,yes
```

```
sunny,?,70,TRUE,yes
overcast,72,90,TRUE,yes
overcast,81,?,FALSE,yes
rainy,?,91,TRUE,no
```

Bibliography

[1] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engieering*, 5(6):914–925, 1993.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining associations between sets of items in massive databases. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 207–216, 1993.

[3] C. G. Atkeson, S. Schaal, and A. W. Moore. Locally weighted learning. *Artificial Intelligence Review*, 11(1-5):11–73, 1997.

[4] D. H. Ballard. *An Introduction to Natural Computation.* MIT Press, 1997.

[5] C. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, 1995.

[6] P. Bradley, U. M. Fayyad, and C. Reina. Scaling EM to large databases. Technical report, Microsoft Research, MSR-TR-98-35, 1998.

[7] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth Statistical Press, 1984.

[8] S. Le Cessie and J. C. Van Houweligen. Ridge estimators in logistic regression. *Applied Statistic*, 41(1):191–201, 1992.

[9] P. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1975.

[10] E. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1971.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[12] C. Cortes and D. Pregibon. Giga-mining. *Proceedings of the International Conference on Machine Learning*, pages 174–178, 1998.

[13] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[14] D. E. Duffy and T. J. Santner. On the small sample properties of norm-restricted maximum likelihood estimators for logistic regression models. *Communs Statist. Theory Meth.*, 18:959–980, 1989.

[15] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

[16] J. Gehrke, V. Ganti, R. Ramakrishnan, and W-Y. Loh. BOAT-Optimistic decision tree construction. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 169–180, 1999.

[17] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest: A framework for fast decision tree construction of large datasets. *Proceedings of the International Conference on Very Large Data Bases*, pages 416–427, 1998.

[18] J. Han and M. Kamber. *Data Mining: concepts and techniques*. Morgan Kaufmann Publishers, 2001.

[19] T. Hastie. *The Elements of Statistical Learning: data mining, inference, and prediction*. Springer, 2001.

[20] A. E. Hoerl and R. W. Kennard. Ridge regression: biased estimates for nonorthogonal problems. *Technometrics*, 12:55–67, 1970.

[21] D. W. Hosmer and S. Lemeshow. *Applied Logistic Regression.* Wiley, 2000.

[22] W. Inmon. *Building the Data Warehouse.* Wiley, 1996.

[23] P. McCullagh and J. A. Nelder. *Generalized Linear Models.* Chapman and Hall, 1989.

[24] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. *Proceedings of the International Conference on Extending Database Technology*, pages 18–32, 1996.

[25] T. M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[26] A. W. Moore and M. Lee. Cached sufficient statistics for efficient machine learning with large data sets. *Journal of Artificial Intelligence Research*, 8:67–91, 1998.

[27] W. Du Mouchel, C. Volinsky, T. Johson, C. Cortes, and D. Pregibon. Squashing flat files flatter. *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, pages 6–15, 1999.

[28] D. Pavlov, H. Mannila, and P. Smyth. Prediction with local patterns using cross-entropy. *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, pages 357–361, 1999.

[29] F. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Journal of Data Mining and Knowledge Discovery*, 3(2):131–169, 1999.

[30] D. Pyle. *Data Preparation for Data Mining.* Morgan Kaufmann Publishers, 1999.

[31] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publisher, 1993.

[32] Agrawal R. and Shim. K. Developing tightly-coupled data mining applications on a relational database system. *Proceedings of the International Conference on Knowledge Discovery in Databases and Data Mining*, 1996.

[33] R. Ramakrishnan and J. Gehrke. *Database Management System.* McGraw-Hill, 2003.

[34] B. J. Ross, A. G. Gualtieri, F. Fueten, and P. Budkewitsch. Hyperspectral image analysis using genetic programmming. *The Genetic and Evolutionary Computation Conference*, pages 1196–1203, 2002.

[35] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[36] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 343–354, 1998.

[37] R. L. Schaefer, L. D. Roi, and R. A. Wolfe. A ridge logistic estimate. *Communs Statist. Theory Meth.*, 13:99–113, 1984.

[38] C. Scholkopf, J. C. Burges, and A. J. Smola. *Advances in Kernel Methods.* MIT Press, 1999.

[39] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. *Proceedings of the International Conference on Very Large Data Bases*, pages 544–555, 1996.

[40] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, 2002.

[41] I. H. Witten and E. Frank. ARFF: attribute relation file format. http://www.cs.waikato.ac.nz/ ml/weka/arff.html.

[42] I. H. Witten and E. Frank. Data mining software in Java. http://www.cs.waikato.ac.nz/ml/weka/.