Data mining using Relational Database Management Systems[⋆]

Beibei Zou[1], Xuesong Ma[1], Bettina Kemme[1], Glen Newton[2], and Doina Precup[1]

[1] McGill University, Montreal, Canada
[2] National Research Council, Canada

Abstract. Software packages providing a whole set of data mining and machine learning algorithms are attractive because they allow experimentation with many kinds of algorithms in an easy setup. However, these packages are often based on main-memory data structures, limiting the amount of data they can handle. In this paper we use a relational database as secondary storage in order to eliminate this limitation. Unlike existing approaches, which often focus on optimizing a single algorithm to work with a database backend, we propose a general approach, which provides a database interface for several algorithms at once. We have taken a popular machine learning software package, Weka, and added a relational storage manager as back-tier to the system. The extension is transparent to the algorithms implemented in Weka, since it is hidden behind Weka's standard main-memory data structure interface. Furthermore, some general mining tasks are transfered into the database system to speed up execution. We tested the extended system, refered to as WekaDB, and our results show that it achieves a much higher scalability than Weka, while providing the same output and maintaining good computation time.

1 Introduction

Machine learning and mining algorithms face the critical issue of scalability in the presence of huge amounts of data . Typical approaches to address this problem are to select a subset of the data [4, 3], to adjust a particular algorithm to work incrementally (processing small batches of data at a time), or to change the algorithms such that they use data structures and access methods that are aware of the secondary storage. For instance, [8, 2] propose algorithms for decision tree construction that access special relational tables on secondary storage. Agarwal et al [7] develop database implementations of Apriori, a well-known algorithm for association rule mining, and show that some very specific implementation details can have a big impact on performance. Their approach achieves scalability by rearranging fundamental steps of the algorithm. Both of these pieces of work require the developer of the mining algorithm to be very familiar with database technology, implementing stored procedures, user defined functions, or choosing the best SQL statements. Machine learning researchers, however, are often not familiar enough with database technology to be aware of all optimization possibilities.

The goal of our research is to provide a general solution to scalability that can be applied to existing algorithms, ideally without modifying them, and that can be used by machine learning researchers to implement new algorithms without the need to be database experts. For that purpose, we have taken a very popular open-source package

⋆ Supported by NSERC, CFI, NRC

of machine learning algorithms, Weka [9], which can only be used on data sets that can fit into main memory, and extended it to be able to use a database as backend. In the extended system, WekaDB, a storage manager interface is defined with two implementations. One implementation is the original main-memory representation of data, the other uses a relational database management system (DBMS). All algorithms implemented in Weka can run in WekaDB without changes, and can use either of the two storage implementations depending on the data set size. Also, new algorithms can be added to the package without developers being required to know SQL.

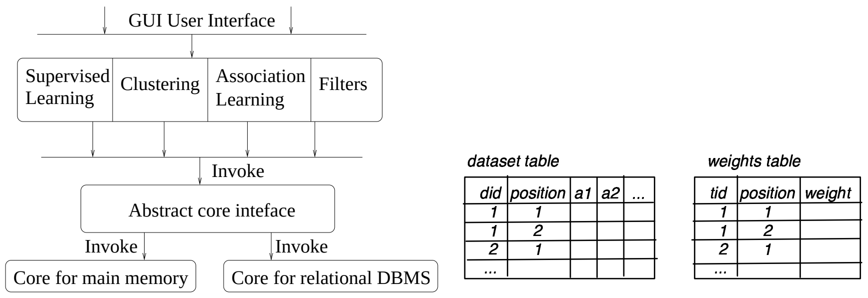
Our basic approach couples Weka and the database rather loosely. The basic model uses the DBMS as a simple storage with the facility to retrieve records individually from the database, perform all computation in main memory, and write any necessary changes back to the database. However, accessing records individually is expensive, so WekaDB also implements several generally applicable optimizations. First, data is transferred in chunks between the database and WekaDB instead of one record at a time whenever possible. Second, many of the storage manager interface methods are implemented using advanced SQL statements; in particular, we take advantage of aggregate functionality (like sum, avg) provided by the DBMS. Third, some popular libraries (e.g., pre-processing filters) that were originally implemented on top of the storage interface, have been reimplemented to take advantage of DBMS functionality. Furthermore, even though WekaDB itself eases data size limitations, the implementations of the machine learning algorithms can create large internal data structures, imposing indirect limitations. In order to address this issue WekaDB provides database implementations for typical main memory data structures, like arrays. The algorithms can access these data structures as if they were implemented in main-memory.

We present an empirical evaluation of WekaDB on both synthetic and real data, using several machine learning algorithms from the original Weka package. The results show significant improvement in terms of scalability, while still providing reasonable execution time. For one of the algorithms, k-means clustering, we also compare with an implementation developed specifically for using a database backend [10]. In some situations, WekaDB's implementation even outperforms this specialized solutuion. In general, our approach is a practical solution providing scalability of data mining algorithms without requiring machine learning developers to be database experts.

2 Weka

Weka [9] is a popular, open source, machine learning software package implementing many state-of-the-art machine learning algorithms. These algorithms all access the data through one well-defined data-structure `core`. The data is represented by two main-memory data structures defined in `core`. A `Dataset` is a set of `Datarecord` objects. Each data record in a dataset consists of the same number of attribute/value pairs and represents one unit of data, e.g., information about a single customer. Additionally, the records have *weight* attributes, which are used by some learning algorithms.

`Dataset` keeps attribute and type information, and maintains a `DR vector` pointing to individual `Datarecord` objects. At the start of any algorithm, an initial `Dataset` object *DS* is created. Then, data records are loaded from an input file into individual `Datarecord` objects. For each object, a pointer is inserted into the `DR vector` of *DS*.



dataset table

did	position	a1	a2	...
1	1			
1	2			
2	1			
...				

weights table

tid	position	weight
1	1	
1	2	
2	1	
...		

(a) Architecture of WekaDB (b) WekaDB tables

Fig. 1. WekaDB

During the computation, a copy *DS*′ of a `Dataset` object *DS* could be made. Copying is lazy. Initially *DS*′ shares the `DR vector` with *DS*. Only when a `Datarecord` object *o* in *DS*′ needs to be modified, a new copy of the `DR vector` is created. All pointers in this vector still point to the old `Datarecord` objects. Then, a new copy of *o* is created, and the corresponding entry in the vector is adjusted accordingly.

Furthermore, `Dataset` provides methods to access and manipulate the data records: `enumerateRecords()` allows to iteratively retrieve data records, while `Record(index)` allows access to a record based on its index in the `DR vector`. General information about the data set can be returned by methods like `numRecords()`. There are also `delete/add` methods, which remove/add the corresponding `Datarecord` pointer from the `DR vector`. The `sort()` method sorts the `DR vector` based on an attribute. Summary statistics are provided by methods such as `sumofWeights()`.

3 Data in WekaDB

Fig. 1(a) shows the the redesigned system architecture. Based on the `core` interface from Weka, we defined a general data structure interface. Any data source that implements this interface can be plugged into Weka. Our new storage manager uses a relational database (currently DB2). `Dataset` and `Datarecord` have been modified to access the database. In `Dataset`, the `DR vector` was replaced by a `P-vector`. Instead of pointing to a `Datarecord`, each entry of the vector contains a `position` integer representing a record in the database. A `Datarecord` object is created (and the record loaded to main memory), whenever it is accessed by the learning algorithm.

3.1 Database Design

Data records reside in the database. Two extreme design alternatives are as follows.

- *Full table copy*: For each `Dataset` object *DS* there is one table TDS containing all records to which the `P-vector` of *DS* points. Making a copy *DS*′ of *DS* leads to the creation of a table TDS', and records in TDS are copied to TDS'.
- *Lazy approach*: There is only one `Dataset` table with a special attribute `did` indicating to which `Dataset` object a record belongs. Initially, all records have the same value *IDS* for `did`. When a copy *DS*′ is made from an existing `Dataset` object *DS*,

DS' shares the records with DS. Only if DS' changes a record for computation purposes, a copy of the original record is inserted into the `Dataset` table with the `did` attribute set to IDS'. This new record will be updated. Each `Dataset` object has to keep track of the set of `did` values with which its records might be labeled.

The full table copy approach is very time consuming if the machine learning algorithm performs many `Dataset` copy operations. However, it might be necessary if an algorithm adds or replaces attributes, i.e., changes the schema information. The lazy approach mirrors the lazy instantiation of new `Datarecord` objects in the main-memory implementation of `core`. Since most machine learning algorithms do not change attribute values, this seems to be the most efficient approach. However, many algorithms do change the weight attributes associated with the records. If this happens, basically all `Dataset` objects will have their own set of records in the `Dataset` table.

Therefore, we store the more static attribute information in a `Dataset` table and the frequently changing weight information in a `weight` table (Figure 1(b)). The `Dataset` objects share the same records in the `Dataset` table unless they change attribute values. If an algorithm never changes attribute values there is one set of data records in `Dataset` with `did`$= IDS$. In contrast, the `weight` table contains, for each existing `P-vector` (`Dataset` objects might share `P-vectors`) its own set of weight records. This set contains as many weight records as there are entries in the `P-vector`. Note that a `P-vector` might have fewer entries than the total number of records with `did` $= IDS$ (e.g., in decision tree construction).

The `Dataset` table has one attribute for each attribute of the original data, a `did` attribute as described above, and a `position` attribute that links the record to the `P-vector` of the `Dataset` object. If a record dr has $did = IDS$ and $position = X$, then DS's `P-vector` has one entry with value X. The `weight` table has attributes `tid` and `position` similar to the `did` and `position` attributes in the `Dataset` table, and a `weight`. In order to match the weights with the corresponding records in the `Dataset` table, we have to join over the `position` and match `did/tid` attributes. Both `Dataset` and `weight` tables have several indices (clustered and unclustered) on `position` and `did/tid` attributes in order to speed up the most typical data access.

Some algorithms change the structure, i.e., they remove or add attributes. This is usually only done in the preprocessing phase. In such cases, full table copies are made. When preprocessing is completed, a `filtereddataset` table is created, which will then be used instead of the original `dataset` table.

3.2 Main Memory Data Structures

Figure 2 shows how the main memory data structures are adjusted in order to allow for the main memory and database storage implementation to co-exist. The abstract class `AbstractDataset` implements variables and methods used in both storage implementations. `MMDataset` is the original `Dataset` implementation in Weka, and `DBDataset` is the abstract class for our relational database implementation. It contains commonly two subclasses. Recall that the `Dataset` object might have to keep track of several `did`. However, if an algorithm never changes attribute values (except the weights), there will be only one `did` value for all records. Hence, we allow algorithms to declare this fact in advance, and then use a simpler implementation which can ignore `did`. The class

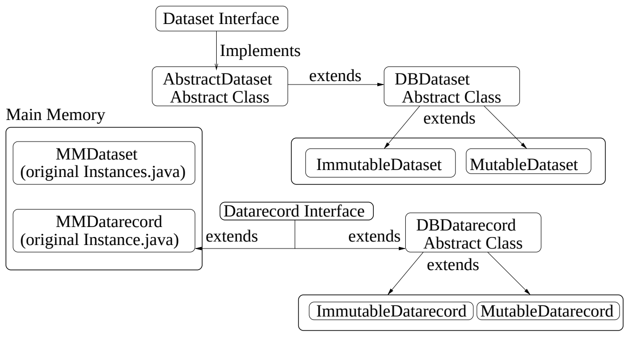


Fig. 2. WekaDB: Dataset and Datarecord

`MutableDataset` supports all the functions that allow algorithms to change attribute values, while `ImmutableDataset` does not support those functions (only the weights are allowed to change). The same class structure is used for data records.

4 Database Access

WekaDB accesses the database using a standard JDBC API. For space reasons, we only outline here how the `ImmutableDataset` class accesses the database. A special `load` interface allows the transfer of records from a file in ARFF format (used by Weka) to the database, creating `dataset` and `weight` tables and the corresponding records.

When an algorithm starts, an initial `ImmutableDataset` object is created with a corresponding `P-vector` based on the information in the `dataset` table. No data records are loaded. The `P-vector` is the only memory-based data structure that grows linearly with the size of the data set. It is needed because algorithms can reorder the records during the computation, for instance by sorting and resampling. In WekaDB, this is done by reordering the entries in the `P-vector`. This vector represents the data in the correct order for a specific `ImmutableDataset` object while the records in the `dataset` and `weight` table are unordered.

If a copy DS' is made from an `ImmutableDataset` object DS, it can share the `P-vector` and `tid` value with DS, or it can create its own `P-vector`, and receive a new `tid` value. In the latter case, it must call the `add` method for each record to which it wants to refer. This method adds the position of the record to the `P-vector` and inserts a weight record into the `weight` table, with the same position and the new `tid` value. No records are added into the `dataset` table for `ImmutableDataset` objects. The new ”copy” of the data set is represented by the new P-vector.

Data records are accessed via the `enumerateRecords()` and `Record(index)` methods of the `ImmutableDataset` class. We only describe the latter here. `Record(index)` on object DS loads the record with position p if DS's `P-vector` V has $V[index] = p$. The (slightly simplified) SQL statement is

```
SELECT * FROM dataset, weight
WHERE weight.tid = IDS
AND dataset.position = p
AND weight.position = p
```

If `Record(index)` is called in a loop accessing all records one by one, the statement is executed repeatedly, and data transfer takes place for each record. Looping is very common in data mining algorithms, which makes this type of data access very expensive. Hence, retrieval will be faster if we load a whole set of records with one query, buffer them within `core` and then provide the records to the user on request within the loop. Hence, we implemented a buffered version of `Record(index)`, which retrieves *B* records at a time. *B* is an adjustable parameter. In the buffered implementation, when a record with position *p* (determined as above) is requested, we first check if the record is already in main memory. If so, the record is returned right away. Otherwise, we use the following (slightly simplified) SQL statement to retrieve *B* consecutive records, starting at position *p*:

```
SELECT *  FROM dataset, weight
WHERE weight.tid = IDS
AND dataset.position =  weight.position
AND dataset.position >= p
AND dataset.position < (p+B)
```

The *B* retrieved records are stored in a JDBC `ResultSet`. For data mining algorithms that access the data sequentially and do not perform any sorting, buffering can dramatically decrease the number of database accessed. If an algorithm had sorted the entries in the `P-vector`, the benefits of buffering are limited.

5 Using more Database Functionality

The loose-coupling approach discussed so far performs all computation on data records in main memory. It might be more efficient to perform some computation within the database by applying advanced SQL functionality. This leads to a semi-tight coupling between Weka and database. For that purpose we modified several `core` methods. As a simple example, our implementation of `sumOfWeights` of the `ImmutableDataset` class uses the SQL aggregate function `sum` to perform the operation within the database. Another example is `sort()`, which orders the data based on the values of one attribute. A main memory implementation requires retrieving records possibly multiple times from the database. In contrast, we use an SQL statement with an `order by` clause.

Data preprocessing [5] is a common step in many algorithms. It is used to clean data, and perform data transformation. Weka provides a set of filters using the `filters` interface. The implementation itself is built on top of `core`. Weka's main memory implementation of the filters accesses records one by one, and stores all the filtered data in a queue. This adds considerable overhead, and reduces scalability due to the queue data structure. We reimplemented the filters using a database oriented approach that does not require loading any records into main memory. For instance, for the filter that replaces all missing attribute values with the modes/means observed in the data, we precompute modes and means with SQL aggregate functions and use `update` SQL statements to replace missing values with these modes and means.

Since the machine learning algorithms are developed without considering space limitations they might create their own data structures that limit scalability. For instance, the logistic regression algorithm implemented in Weka normalizes the input data and stores it in a 2-dimensional array (one dimension represents the records, the other the normalized attributes). This is done by a pass through the data set using `Record(index)` calls. This array is, in fact, as large as the entire data set. Our approach is to provide adequate support in order to help developers eliminate such limitations. Normalizing records and then accessing the normalized data in a systematic way seem to be standard steps usable in various algorithms. Therefore, we offer extra normalization methods as part of the `Dataset` class. The methods use SQL queries to perform the normalization and store the normalized values in the database. The normalized data can be retrieved through an interface that provides the standard array representation. It can be used without knowing that a database implementation is used. Whenever a normalized record is accessed through the array interface, a corresponding SQL query retrieves the record from the database.

6 Optimizing JDBC Applications

Our implementation uses several standard mechanisms to speed up the JDBC application. First, our system uses a single database connection for all database access to optimize connection management. Since transaction management is expensive, we bundle related operations in a single transaction in order to keep the number of transactions small. Third, since the system runs in single user mode, we run our transactions in the lowest isolation mode provided by the database to minimize the concurrency control overhead. Finally, we use JDBC's `PreparedStatement` objects as much as possible, since these statements are parsed and compiled by the database system only once, and later calls use the compiled statements, improving performance significantly.

7 Empirical evaluation

We evaluate WekaDB using the logistic regression algorithm, the Naive Bayes algorithm and the K-means clustering algorithm, on both synthetic and real data sets. The first two algorithms are used for classification problems, while the last one is an unsupervised learning algorithm. We note that at the moment, all algorithms in the Weka package, except the decision tree construction algorithm, work seamlessly with WekaDB. Once logistic regression and k-means clustering were fully functional, the other algorithms worked with WekaDB without any further tweaking.

The synthetic data sets were generated using the program of [1]. We generated training data sets with 10,000 to 1,000,000 records, and one testing data set with 5000 records. Each data set has 10 numerical attributes and 1 class attribute, without missing values. We also run tests with filters for replacing missing values and for discretizing continuous attributes. The results were very similar to the ones reported here.

The real data set is based on the AVIRIS (Airborne Visible/Infrared Imaging Spectrometer) data set, originally created at JPL (Jet Propulsion Laboratory, California Institute Technology) and extensively corrected by CCRS (Canadian Center for Remote Sensing, Natural Resources Canada). It contains hyperspectral data that was captured by the NASA/JPL AVIRIS sensor over Cuprite, Nevada on June 12, 1996 (19:31UT) (see [6] for more information). The original data set contains 314, 368 records and 170 attributes. For the purpose of the experiments, we generated four different data sets, containing 12669, 19712, 35055 and 78592 records respectively, and one testing data set containing 3224 records. Each data set has 168 numeric attributes and 1 nominal class attribute without missing values.

We restricted the memory size to be used by Weka/WekaDB to 64MB in order to avoid long running times and be able to run many experiments. All experiments use the default values for all the algorithms, unless otherwise specified.

In all the experiments we measure the runtime of the algorithms when we increase the size of the training data set. In all cases, the main-memory implementation of Weka is significantly faster (between 2-3 orders of magnitude). However, the maximum number of instances that it can handle is below 40000. WekaDB, on the other hand, can handle up to 700000 - 800000 instances, which is a 20-fold improvement.

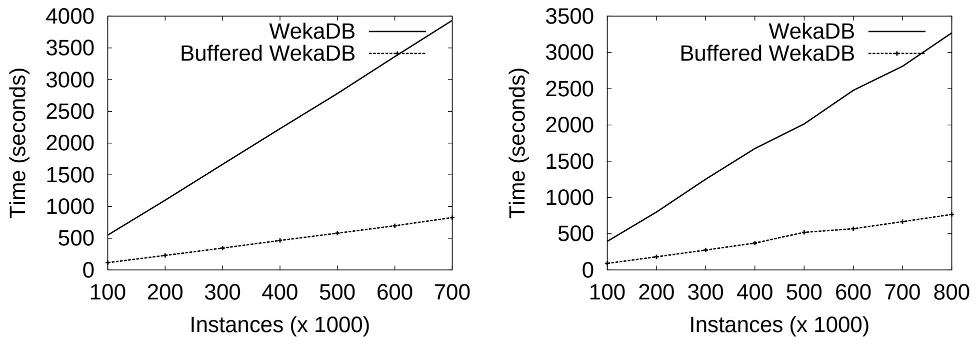
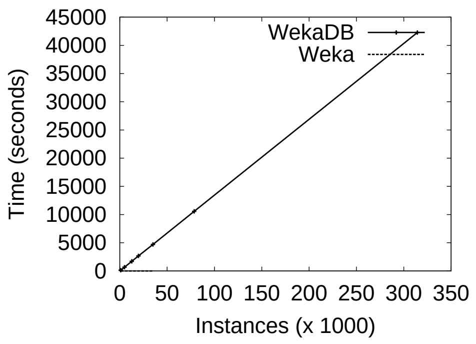


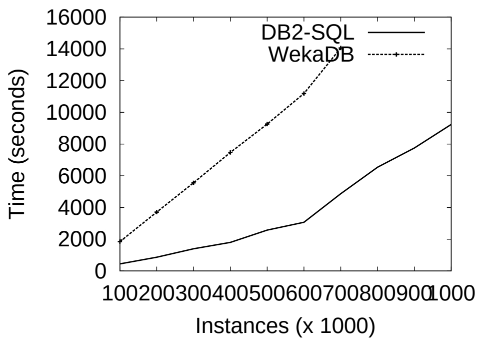
Fig. 3. WekaDB for Naive Bayes (left) and Logistic regression (right) on synthetic data

Figure 3 illustrates the running time on the synthetic data set for WekaDB with a buffer (size 10000) and without using a buffer for Naive Bayes (left figure) and logistic regression (right figure). As illustrated, the computation time increases linearly with the number of instances, which should be expected, as both algorithms have to loop through the data. A similar linear increase is observed in the original Weka implementation, but with a much smaller slope, of course. Using the buffer yields a 5-fold improvement in speed. Weka's computation time is 11 seconds at 28000 records (not shown in the figure), compared to 368 seconds for WekaDB with a buffer. Hence, Weka's computation is roughly 30 times faster. For logistic regression, the time difference is more pronounced, with Weka finishing in 0.9 seconds, compared to 251 seconds for WekaDB. However, Weka is showing a significant space limitation running out of memory at 29000 instances. At 700000 and 800000 records respectively, WekaDB also runs out of memory, because at this point the position vector becomes a memory constraint (since it grows linearly with the size of the data set). Recall that we use only 64MB. If we assume that 1GB of main memory is available, we can expect WekaDB to handle on the order of 10,000,000 records while Weka will handle less than 500,000.

Figure 4(a) presents the performance of the original (main-memory) Weka implementation compared to WekaDB with a buffer on the AVIRIS dataset. The results are consistent with those on the synthetic data: Weka can only handle 35000 instances, roughly 10% of the size of the original data set. WekaDB is roughly 1000 times slower at 35000 instances, but can handle the entire data set successfully. Performance scales

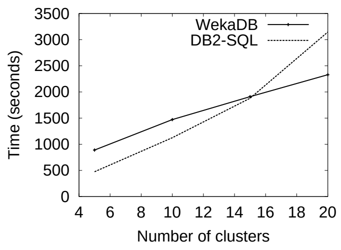


(a) Weka and buffered WekaDB
for Logistic regression (AVIRIS data)

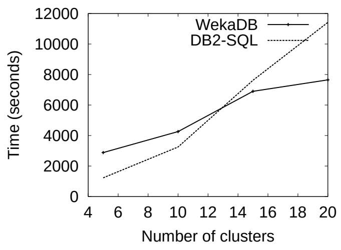


(b) DB-SQL and buffered WekaDB
for k-means clustering, $k = 2$ (synthetic data)

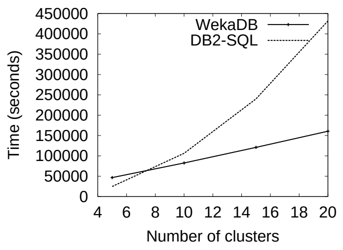
Fig. 4. Performance Comparisons



(a) 20000 records



(b) 50000 records



(c) 300000 records

Fig. 5. DB-SQL and buffered WekaDB for k-means clustering on different dataset sizes (synthetic data)

linearly with the number of instances. The reason for the large discrepancy in running time is the number of attributes, which is much larger than for the synthetic data. Very similar results are obtained with Naive Bayes.

For k-means clustering, the comparison of Weka, WekaDB and buffered WekaDB is very similar to those presented before, so we omit it here. Instead, Fig.4(b) compares WekaDB using a buffer to a k-means algorithm proposed by Ordonez [10], which also stores the data in a database and re-implements the computation in the database. This algorithm takes particular care to take advantage of any database-specific optimizations. We refer to it as DB2-SQL. As expected, the optimized algorithm can scale better (since there are no limiting memory data structures), and is faster for the particular number of clusters requested ($k = 2$). However, further analysis of the algorithms shows an interesting trend. Fig. 5 shows the behavior of the two algorithms as we vary the number of clusters between 5 and 20, for different dataset sizes. As the number of clusters increases, the computation time of WekaDB grows linearly, while that of DB2-SQL grows super-linearly. Hence, when there are many clusters, the simple k-means algorithm in WekaDB outperforms the specialized DB2-SQL implementation.

As the number of instances also increases, WekaDB has even better performance. This is due to the fact that in the DB2-SQL implementation, the cluster centers and all auxiliary memory structures are in the database. As instances have to be compared to the cluster centers in order to decide where they belong, the computation time degrades. Also, larger numbers of clusters typically require more iterations of the algorithms. In WekaDB, since the number of clusters is still relatively small, a lot of the processing is done in main memory, which makes it much faster.

8 Conclusions

This paper presented an approach to the integration of learning algorithms with relational databases. We built an extension of the well-known Weka data mining library, WekaDB, which allows the data used by the learning algorithms to reside on secondary storage. This change is transparent to the developers of machine learning algorithms. Our empirical results show that this approach provides scalability up to very large data sets. From the point of view of evaluating empirically the performance of new data mining algorithms, we believe that WekaDB provides an interesting benchmark, since it provides a faithful implementation and execution of the learning algorithms. Other approaches, such as resampling and special-purpose algorithms, can be compared to it in terms of accuracy, as well as scalability and computation time. Also, in principle, WekaDB allows any new algorithms that are added to the Weka package to be able to work immediately on data stored in a database, without any further modifications. We currently work on removing the remaining memory limitation for WekaDB, by eliminating the need to have an internal memory data structure linear in the size of the data set. The idea is to store the position as additional attribute in the dataset table.

References

1. R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engieering*, 5(6), 1993.
2. J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest: A framework for fast decision tree construction of large datasets. *Int. Conf. on Very Large Data Bases*, 1998.
3. A. W. Moore and M. Lee. Cached sufficient statistics for efficient machine learning with large data sets. *Journal of Artificial Intelligence Research*, 8, 1998.
4. W. Du Mouchel, C. Volinsky, T. Johson, C. Cortes, and D. Pregibon. Squashing flat files flatter. *ACM Int. Conf. on Knowledge Discovery and Data Mining*, 1999.
5. D. Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, 1999.
6. B. J. Ross, A. G. Gualtieri, F. Fueten, and P. Budkewitsch. Hyperspectral image analysis using genetic programmming. *The Genetic and Evolutionary Computation Conf.*, 2002.
7. S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. *ACM SIGMOD Int. Conf. on Management of Data*, 1998.
8. J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. *Int. Conf. on Very Large Data Bases*, 1996.
9. I. H. Witten and E. Frank. Data mining software in Java. http://www.cs.waikato.ac.nz/ml/weka/.
10. Carlos Ordonez. Programming the K-means Clustering Alogrithm in SQL. *ACM Int. Conf. on Knowledge Discovery and Data Mining*, 2004.