SIPping from the Data Firehose

George H. John
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Brian Lent
Stanford University
Department of Computer Science
Stanford, CA 94305

`{gjohn,lent}@cs.stanford.edu`
`http://www-cs-students.stanford.edu/~{gjohn,lent}`

Abstract

When mining large databases, the data extraction problem and the interface between the database and data mining algorithm become important issues. Rather than giving a mining algorithm full access to a database (by extracting to a flat file or other directly-accessible data structure), we propose the SQL Interface Protocol (SIP), which is a framework for interaction between a mining algorithm and a database. The data continues to reside entirely within the database management system (DBMS), but the query interface to the database gives the data mining algorithm sufficient information to discover the same patterns it would have found with direct access to the data. This model of interaction brings several advantages; for example, it allows a mining algorithm to be parallelized automatically just by using a parallelized DBMS to answer queries. We show how two families of mining algorithms may be implemented as "SIPpers," and we discuss related work in databases that should further enhance performance in the future.

Introduction

Data mining algorithms are commonly expressed as programs that operate directly on a data file. Early algorithms from machine learning and statistics often loaded the entire file into main memory as a first step. As we move into industrial applications involving the mining of large data warehouses, this model becomes insufficient. Today, "small" data warehouses are on the order of ten to twenty gigabytes in size. Not only is it infeasible to load such databases into main memory to mine the data, it also becomes quite unwieldy to manually extract data into flat files for analysis.

The data mining process begins with precisely stating a problem to be solved, and then deciding which data are appropriate to use in building a solution. The next step is to obtain the relevant data and process them into a form that a data mining algorithm can understand. Though often a simple problem technically, in practice this can be an arduous process, rife with shortcomings: duplicated data (if extracted into flat files), managing sets of files (to overcome some operating systems' filesize limits), having to repeat the entire process (if the data model changes), and so forth.

Not only is it cumbersome to give a mining algorithm access to the entire contents of a database, but it is also unnecessary. Many mining algorithms are based on relatively simple summary statistics of the database, in which case the full information contained in the database is superfluous. Ideally, we should instead work within a framework where the mining algorithm simply gets the statistics it needs from a DBMS. This can be achieved through our proposed SQL Interface Protocol (SIP). Rather than drinking from the data firehose, the mining algorithm can direct the DBMS to process the volumes of raw data to calculate the necessary statistics, and thus the data mining algorithm needs only to take a *sip* of this concentrated extract.

Information Requirements and SIP

The key to implementing a mining algorithm as a SIPper will be determining exactly what information the algorithm needs. As an example, let us imagine a customer retention problem in newspaper subscribers. During the course of building a classification model, a mining algorithm might need to know the probability that a subscriber will cancel, given that they are in an older age group and have had recent negative interaction with the customer service department. Based on this probability and results of previous queries, the mining algorithm will issue many new statistical queries. As well as querying the probability of a single event as above, the algorithm might want to know the joint distribution—the probability of observing each of the possible combinations of values of CANCEL, AGE, and SERVICE.

The SQL `GROUP BY` query gives exactly the desired results, returning a convenient table of counts showing how often any particular combination of the selected set of variable values occurred in the entire database,

```
> SELECT CANCEL, AGE, SERVICE, COUNT(*)
  FROM SUBSCRIBERS
  WHERE DELIVERYTIME=MORNING AND
        PAYSCHEDULE=SPECIAL
  GROUP BY CANCEL, AGE, SERVICE

CANCEL  AGE    SERVICE  COUNT
------  ---    -------  -----
yes     old    positive   430
yes     old    negative  2214
yes     young  positive   322
yes     young  negative   631
no      old    positive 31227
no      old    negative 23419
no      young  positive 11993
no      young  negative 15526
```

Figure 1: Example GROUP BY query in SQL, with COUNT aggregation.

or in some selected (using WHERE) subset of the records (Ullman 1988). Figure 1 shows an example GROUP BY query and result where we restrict our attention to morning-delivery subscribers on a special payment schedule (which might be the rates offered by a special promotion). The result is a table of counts broken down by whether or not the subscriber canceled, their age group, and their recent customer service rating.

Given a set of attribute names ϕ and a set of attribute-value assignments ψ, SQL GROUP BY queries can easily answer the count query $N(\phi|\psi)$, which is a table of the number of times each possible assignment of values to the variables in ϕ was observed, considering only records where ψ holds. In the example above ϕ ={CANCEL, AGE, SERVICE} and ψ ={DELIVERYTIME = MORNING, PAYSCHEDULE = SPECIAL}. ϕ and or ψ might be null. When ϕ is null, $N(|\psi)$ returns a single count of all records where ψ holds. When ψ is null, $N(\phi)$ is a count of each possible value assignment to variables in ϕ, counting over the entire table. When both are null, $N()$, or simply N, is the total number of tuples in the relation. We assume that all queries are with respect to a single relation. If necessary, we can construct a view to make a single relation out of many tables.

The count query leaves to the mining algorithm the job of estimating the probability distribution from the counts, which is appropriate. Counts, rather than probabiliites, are also important for statistical hypothesis tests that some algorithms use during mining.

We will refer to a query as a *sip* if it meets certain criteria given in the definitions below. A *pure* sip returns only statistics, and can answer the $N(\phi|\psi)$ count queries mentioned above. An *impure* sip returns an actual fragment of the database, but the intent is for the fragments to be small and the number of impure queries to be low.

Definition 1 *A* **Pure Sip** *is an SQL query of a DBMS by a mining algortihm of the form*
SELECT $\phi, f(*)$
FROM D
WHERE ψ
GROUP BY ϕ
where f is an aggregation operator (e.g., count, sum, avg, stddev, min, max), ϕ is a list of attribute names, ψ is a list of attribute names with associated constraints, and D is the name of the relation.

Some data mining algorithms can *almost* be implemented using only pure sips, but occasionally require more direct access to the database. For such algorithms we use impure sips.

Definition 2 *A* **k-th Degree Impure Sip** *is an SQL query of a DBMS by a mining algortihm of the form*
SELECT ϕ
FROM D
WHERE ψ
where ϕ includes exactly k attribute names.

Other algorithms, such as neural nets, need access to each data record during mining and will not be able to work within this framework. For an n-dimensional relation, the nth degree impure sip is a copy of the entire relation, and thus any mining algorithm that normally uses flat files could as well be implemented in SIP using nth degree impure sips. This obviates the need for flat file extracts, but is otherwise uninteresting and is not in the spirit of the SIP framework.

We will next discuss the C4.5 (Quinlan 1993) classification tree algorithm and the simple Bayesian classifier, two examples of algorithms that can be implemented as SIPpers.

The C4.5 Classification Tree Algorithm

Classification trees predict the class of a record given the values of the rest of the attributes by asking a series of questions terminating in a prediction. During the tree growing phase, most tree-mining algorithms repeatedly select a question to ask at a node (*e.g.*, "what is the value of SERVICE?"). Once a question is determined, the database records are partitioned by the answer and child nodes are created for each partition. The process repeats on each child, until a stopping criterion is met and the node is made into a leaf that predicts the most frequent class in that partition.

During tree growth, C4.5 uses either the *information gain* or the *gain-ratio* splitting criterion to determine which question to use at a node. Recall the newspaper customer attrition example. Assume that we are two levels deep in the tree, at a node that is reached when DELIVERYTIME = MORNING and PAYSCHEDULE = SPECIAL. The two candidate questions are the binary split by AGE or SERVICE. C4.5's information gain splitting criterion minimizes the *conditional entropy* of the class given the attribute, $H(C|X) = -\sum_x p(X{=}x) \sum_c p(C{=}c|X{=}x) \log_2 p(C{=}c|X{=}x)$.

All of the probabilities involved in this calculation can be estimated from counts. To calculate the conditional entropy of the class given a single attribute, we would use a pure sip where ϕ contains the attribute and the class, and ψ is the set of constraints defined by the path from the current node to the root. The example query of Figure 1 does not fit this specification; its ϕ includes two attributes. This returns more information than we would need to calculate the conditional entropy for either attribute by itself, but might be more efficient than issuing two separate queries. In this example, the conditional entropy is smallest for SERVICE, which would be picked as the best question by information gain. Gain-ratio is similarly based on counts that can be obtained from a sip.

During pruning, the C4.5 algorithm tests the hypothesis that replacing a node with a leaf will not increase the error of the tree. If the hypothesis cannot be rejected, pruning occurs, and the node is replaced with a leaf. The hypothesis test is based on counts: in the leaf and interior node, it counts the number of training records from each class. The counts may be obtained through pure sips. (They can also be stored in the tree structure during training.)

The standard approach to splitting on a continuous attribute requires a second degree impure SIP to get the attribute and class values for all records. However, prediscretization would alleviate the problem, and several methods have shown promise (Dougherty, Kohavi & Sahami 1995).

Simple Bayes

The simple Bayesian classifier directly characterizes the conditional distribution of the class given the attributes (John & Langley 1995). It makes strict ("simple") assumptions that allow it to build a model by estimating only a small number of parameters. Although one would think the model too simple to be of any use, a surprising characteristic of the algorithm is that it works quite well on many problems.

When mining for predictive patterns such as the simple Bayesian classifier, we are interested in the conditional distribution of the class given the observed attributes. Bayes' rule tells us that the conditional distribution of the class given the attributes, $P(C|\mathbf{X})$, equals $P(\mathbf{X}|C) \times P(C)/P(\mathbf{X})$. By assuming conditional independence between the attributes and the class, this equals $\prod_i P(X_i|C) \times P(C)/P(\mathbf{X})$, which is the model for the the simple Bayesian classifier.

During mining, Simple Bayes gathers statistics (counts) required to model $P(C)$ and $P(X|C)$ for each input attribute X. Using count queries and maximum likelihood estimation, we can estimate $\hat{P}(C)$ as $N(C)/N$. For each nominal or categorical attribute X we estimate $\hat{P}(X|C)$ as $N(X,C)/N(C)$. For continuous attributes we assume $\hat{P}(X|C{=}c)$ is Gaussian and we estimate the mean and standard deviation separately for each c (which we can also do using `GROUP BY` with the `AVG` and `STDDEV` aggregation operators).

All of the queries used in estimating $P(C)$ and $P(X|C)$ can be expressed as pure sips, where the aggregation function is either `COUNT` (for categorical attributes), or `AVG` and `STDDEV` (for numeric attributes).

Many algorithms for building more general Bayesian networks, which relax the conditional independence assumption, are also based on counts and may be expressed in the SIP framework. For example, Friedman & Goldszmidt (1996) present a fast algorithm for learning more general Bayes nets that could still be implemented using only pure sips.

Related and Future Work

In SIP, the relationship between mining algorithm and database management system is essentially that of customer and vendor. The vendor, the DBMS, needs to provide fast and accurate responses. Research in databases could speed up sips in three ways: by serving precomputed results, by extending query optimization to handle a set of similar queries issued at once, and by parallelizing and optimizing the DBMS code.

In the past two years, a number of researchers in databases have focused on efficiently computing aggregates, an important operation in on-line analytical processing (OLAP) databases used for decision support. Graefe (1993) describes a sort-based and hash-based method for efficiently computing the answer to a single `GROUP BY` query, and would thus be useful in speeding up a single pure sip. Regarding multiple similar queries, Gray, Chaudhuri, Bosworth, Layman, Reichart, Venkatrao, Pellow & Pirahesh (1996) introduced the `CUBE BY` operator. In our terminology, the `CUBE BY` operator takes a set of attributes ϕ and returns the answers to all pure sips (Definition 1) on the power set of ϕ. Sarawagi, Agrawal & Gupta (1996) present the PipeSort algorithm for efficiently computing the data cube.

Even more promising is the trend towards off-line precomputation of queries, which has been fueled by the need for fast responses in decision support systems (*e.g.*, Mervyn's retail warehouse has 2400 precomputed aggregations). Gupta, Harinarayan & Quass (1995) describe new query-optimization methods that use precomputed aggregates when available, and Gupta, Harinarayan, Rajaraman & Ullman (forthcoming) describe indexing structures for precomputed results.

Efficiently answering simple count queries is already an important topic in databases, because counts are used in query optimization. Whang, Kim & Wiederhold (1994) explore the multi-level grid file method for precomputing and incrementally updating multidimensional counts, which is logarithmic in the number of attributes. Slower but more exact methods for precomputing and updating statistics are studied in *statistical databases* (Michalewicz 1992).

Most of the top DBMS vendors have parallelized their systems. With a clean and modular interface between the mining components and the relational DBMS, performance benefits realized by the DBMS will immediately benefit the mining process as well.

Looking at the other aspect of the relationship between mining algorithms and databases, a mining algorithm might serve a decision support database by proposing queries that would be of interest to a human expert looking at the same database (Jeffrey Ullman, personal communication), and thus the mining algorithm can guide the precomputation process.

The inspiration for the SQL interface protocol was the *statistical query model* of learning (Kearns 1993). Any mining algorithm that fits the statistical query model will be SIP-compliant. The idea that most classification tree splitting criteria are based on simple probability estimates is described in John (1996).

Conclusion

As noted by Silberschatz, Stonebreaker & Ullman (1995), as databases and data mining algorithms are more frequently used together, the interface between the two is an important area that merits further research and development. Since the database community has already developed efficient and parallel algorithms for building, maintaining, and retrieving information from large databases, data mining algorithms should clearly leverage their efforts. An obvious approach, embodied in our SQL Interface Protocol, is to rely upon a database management system to provide the statistics that a mining algorithm needs to build a model. For two common families of mining algorithms, this is straightforward. Current research in databases may provide extensive performance enhancements by precomputing and maintaining answers to many possible queries, or by optimizing sets of related queries.

Acknowledgments George John's work was supported under a National Science Foundation Graudate Research Fellowship. We thank the MIDAS (MIning Data At Stanford) group for ideas and encouragement.

References

Dougherty, J., Kohavi, R. & Sahami, M. (1995), Supervised and unsupervised discretization of continuous features, in *Machine Learning: Proceedings of the 12th International Conference*, Morgan Kaufmann.

Friedman, N. & Goldszmidt, M. (1996), Building classifiers using Bayesian networks, in *AAAI-96: Proceedings of the Thirteenth National Conference on Artificial Intelligence*, AAAI Press/MIT Press.

Graefe, G. (1993), "Query evaluation techniques for large databases", *ACM Computing Surveys* **25**(2), June, pp. 73–170.

Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F. & Pirahesh, H. (1996), "Data cube: A relational aggregation operator generalizaing group-by, cross-tab, and sub totals", *Knowledge Discovery and Data Mining* **1**(1).

Gupta, A., Harinarayan, V. & Quass, D. (1995), Aggregate-query processing in data warehousing environments, in *Proceedings of the 21st Very Large Databases Conference*, pp. 358–369.

Gupta, H., Harinarayan, V., Rajaraman, A. & Ullman, J. D. (forthcoming), Index selection for OLAP, in *International Conferece on Data Engineering*.

John, G. H. (1996), Robust linear discriminant trees, in D. Fisher & H. Lenz, eds, *Learning From Data: Artificial Intelligence and Statistics V*, Lecture Notes in Statistics, Springer-Verlag, New York, chapter 34.

John, G. H. (1997), Enhancements to the Data Mining Process, PhD thesis, Computer Science Department, Stanford University, Stanford, California.

John, G. H. & Langley, P. (1995), Estimating continuous distributions in Bayesian classifiers, in *Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, Morgan Kaufmann Publishers, San Mateo.

Kearns, M. (1993), Efficient noise-tolerant learning from statistical queries, in *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, ACM, New York, pp. 392–401.

Michalewicz, Z. (1992), *Statistical and Scientific Databases*, Ellis Horwood.

Quinlan, J. R. (1993), *C4.5: Programs for Machine Learning*, Morgan Kaufmann.

Sarawagi, S., Agrawal, R. & Gupta, A. (1996), On computing the data cube, Technical report, IBM, Almaden Research Center, San Jose, CA.

Silberschatz, A., Stonebreaker, M. & Ullman, J. (1995), "Database research: Achievements and opportunities into the 21st century", NSF Workshop on the Future of Database Systems Research.

Ullman, J. D. (1988), *Principles of Database and Knowledge-Base Systems: Volume 1: Classical Database Systems*, Addison-Wesley.

Whang, K.-Y., Kim, S.-W. & Wiederhold, G. (1994), "Dynamic maintenance of data distribution for selectivity estimation", *VLDB Journal*, Jan, pp. 29–51.