Support Vector Machines in Relational Databases

Stefan Rüping
Department of Computer Science, Universität Dortmund
rueping@ls8.cs.uni-dortmund.de

Abstract: Today, most of the data in business applications is stored in relational databases. Relational database systems are so popular, because they offer solutions to many problems around data storage, such as efficiency, effectiveness, usability, security and multi-user support. To benefit from these advantages in Support Vector Machine (SVM) learning, we will develop an implementation of the SVM learning algorithm, that can be run inside a relational database system. Even if this kind of implementation obviously cannot be as efficient as a standalone implementation, it will be favorable in situations, where requirements other than efficiency for learning play an important role.

1 Introduction

There exist many efficient implementations of Vapnik's SVM [Vap98] [1]. So why would another SVM implementation be of interest? In this paper we aim for an implementation, that is more adapted to the needs of the user in real-world applications of knowledge discovery.

Today, most of the data in business applications is stored in relational data-bases or in data warehouses built on top of relational databases. On the other hand, available SVM software is either implemented as a standalone tool in a programming language like C, or as part of a numerical software such as Matlab.

Of course, it would be easy to export the relevant data from the database, run the SVM software and load the results back into the database, but this approach suffers from various drawbacks:

Usability: Learning algorithms in general cannot be applied independently. Preprocessing steps have to be taken to clean and transform the data, that can be as complex as the final learning task itself [Pyl99],[CCK+99]. The same preprocessing steps have to be taken in order to apply the result to new examples.

Efficiency for learning: While a standalone SVM application can be expected to be much more efficient than an SVM as a database application, the time that is necessary to transfer the data from the database to the application cannot be neglected.

[1] see for example http://www.kernel-machines.org/ for a list of available SVM software

Efficiency for prediction: The evaluation of the final decision function is relatively easy. Calling an external application to evaluate every new example would be extremely ineffective.

Security: Commercial database management system offer fine grained possibilities to control, which user can access or modify which data. If the data is exported from the database, expensive additional measures have to be taken to guarantee this level of security.

In this paper, we approach this problem by implementing an SVM that can be run entirely inside a database server. We do this by making use of Java Stored Procedures as the core of the program and the use of pure SQL statements to compute intermediate results whenever possible.

2 Support Vector Machines

The principles of SVMs and of statistical learning theory [Vap98] are well known, so we omit an introduction of the SVM algorithm in this paper. See [Vap98] and [Bur98] for a introduction to SVMs.

The only thing we need to know is that SVMs find a function $f(x) = wx + b$ based on data (x_i, y_i) and that the calculation depends on the x-values only via the inner product $x_i \cdot x_j$ (the results of this paper can be generalized to non-linear SVMs, where the inner product is replaced by a kernel function $K(x_i, x_j)$).

In practical implementations of SVMs it turns out that three tricks can speed up the calculation of the SVM solution dramatically:

Working set decomposition: Osuna et. al. [OFG97] suggest to iteratively split the problem into a sequence of simpler problems by fixing most variables and optimizing only on the rest, the so-called working set.

Shrinking: Variables that are optimal at their lower or upper bound for a certain number of iterations are fixed at that position and not re-examined in any further iteration.

Kernel caching: The values $s_i = \sum_{j=1}^{n} \alpha_j K(x_j, x_i)$, that are needed to compute the gradient of the target function can be computed once and be updated by $s_i' = s_i + (\alpha_j' - \alpha_j)K(x_i, x_j)$ whenever a variable changes from α_j to α_j'.

3 An SVM Implementation for Relational Databases

The only access to the examples x-values in SVMs is via the kernel function K. So, as the most simple approach one could use any given SVM implementation and replace the call of the function $K(x_i, x_j)$ by the call of a function

$$K(read\_from\_database(i), read\_from\_database(j))$$

Unfortunately, this simple approach does not work very well. The reason for this is, that any access to the database is far more expensive than a simple memory access. To make the code more efficient, we need to reduce the number and size of database queries as much as possible.

3.1 Database Kernel Calculation

There is a more efficient way to access the examples: As we do need only the value of $K(x_i, x_j)$, there is no need to read both x and y from the database, if we can read $K(x_i, x_j)$ directly. Then, instead of $2d$ number, only one number has to be read from the database. This can be easily accomplished in SQL. The following SQL statement gives the value of K:

```
select X.att_1 * Y.att_1 +...+ X.att_d * Y.att_d
from EXAMPLES where X.index = <i> and Y.index = <j>
```

To demonstrate the effect of this optimizations, we give the runtime of this version on two data sets, one linear classification task PAT and one linear regression task REG.

Dataset	Old Version	New Version
PAT	23.81s	13.94s
REG	1156.26s	676.64s

3.2 Kernel Rows

The experiment in the last section shows, that there is still need for improvement. The reason for the inefficiency of the last approach is that a lot of time in the database is spent analyzing the query and looking up the data tables. Once the tables are found, calculating the result is relatively easy. This means, that a very limiting factor for performance is the number of calls to the database and not so much not the size of the data itself.

In section (2) we have seen that the kernel values are not accessed randomly, but in terms of kernel rows. So we can optimize the database access, if we select the whole kernel row in one query:

```
select <KERNEL_SELECT>, Y.index
from EXAMPLES X, EXAMPLES Y where X.index = <i>
```

Here the term `<KERNEL_SELECT>` stands for the SQL term that constructs the kernel value from the attributes, e.g. `X.att_1 * Y.att_1 +...+ X.att_d * Y.att_d`.

We also need to get the index of Y to make a kernel row of the result set, as the order the results are returned in is not defined.

From the following table we can see, that this optimization reduces the runtime by about 15% to 35%.

Dataset	Old Version	New Version
PAT	13.94s	11.96s
REG	676.64s	426.66s

3.3 Shrinking

Shrinking has a big effect on runtime, because information on shrinked examples does not need to be updated in further iterations. The only kernel information needed in later iterations is that of the sub-matrix of non-shrinked examples. To get only these kernel entries, the query to select a kernel row can be adapted.

What we need to do is to adjust the `from EXAMPLES Y` part of the kernel SQL statement, such that only non-shrinked examples are considered. We can create a table named `free_examples` that contain only the indices of non-shrinked examples. Then the kernel query becomes:

```
select <KERNEL_SELECT>, Y.index
from EXAMPLES X, EXAMPLES Y where X.index = <i>
   and Y.index in (select index from free_examples)
```

3.4 The Decision Function

To be useful for application in real-world databases, we do need also an efficient way to evaluate the SVM decision function $f(x) = \sum_{i \in SV} y_i \alpha_i x_i \cdot x + b$ on new examples. This can simply be done with pure SQL statements.

With the linear kernel we can make use of the linearity and write $f(x) = \sum_{i \in SV}(y_i \alpha_i x_i \cdot x) + b = (\sum_{i \in SV} y_i \alpha_i x_i) \cdot x + b =: w \cdot x + b$. So we only need to calculate the vector w and the constant b after learning and can write

```
select <w_1> * X.att_1 + ... + <w_d> * X.att_d + <b> as f
from X in TOPREDICT
```

to get the f-values from the examples in table `TOPREDICT`.

4 Experiments

We used two implementations of the SVM to compare the efficiency of the database version of the SVM to a C++ standalone version. Both SVMs used the same algorithm and parameters. Two datasets were used in the comparison. The first data set PAT consisted of a simple artificial classification task, the second data set REG is an artificial regression problem.

In the case of the standalone version, also the time needed to create the input files from the database tables was recorded. The following table shows the time needed to access the data from the database for the standalone C++ -Version, the CPU time of the standalone version and the total time for the standalone version. This is compared to the CPU time of the database version:

Name	Db Access	C++ SVM	C++ Total	Db SVM	Factor
Pat	0.29s	0.16s	0.45s	8.73s	19.40
Reg	6.06s	3.48s	9.54s	364.72s	38.23

The experiments show, that the database version is slower than the standalone version by a factor of 20 to 40. If this difference is acceptable has to be evaluated with respect to the individual application's requirements.

5 Discussion and Further Research

In relational databases, data is typically not stored in one but in multiple relations. As the SVM cannot deal with multi-relational data, the different tables would have to be joined together for the SVM to access them. In the worst case, the join of two tables of size m and n can have the size $m \cdot n$, when every row of the first table can be joined with every row of the second table. Of course, one would like to avoid having to store this data as an intermediate step.

Fortunately there is a trick in the case of SVMs. The important observation is, that the inner product of two $n+m$-dimensional points (x_M, x_N) and (y_M, y_N) can be calculated as the sum of an n- and an m-dimensional inner product: $(x_M, x_N) \cdot (y_M, y_N) = x_M \cdot y_M + x_N \cdot y_N$.

This mean, instead of a kernel matrix of size $(n \cdot m)^2$ it suffices to compute two matrixes of size n^2 and m^2 of the inner products and calculate the kernel values from them. In the case of kernel caching, this trick allows for a far more efficient organization of the cache as two independent caches.

5.1 Discussion

This paper proposed an implementation of a SVM on top of a relational database. Even as this implementation obviously cannot be as efficient as a standalone implementation with direct access to the data, considerations such as data security, platform-independence and usability in a database-centered environment suggest that this is a significant improvement for SVM applications in real-world domains. It has been shown, that the optimal usage of database structures can significantly improve performance.

Acknowledgments: The financial support of the Deutsche Forschungsgemeinschaft (SFB 475, ”Reduction of Complexity for Multivariate Data Structures”) is gratefully acknowledged.

Literaturverzeichnis

[Bur98] C. Burges. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[CCK+99] Pete Chapman, Julian Clinton, Thomas Khabaza, Thomas Reinartz, and Rüdiger Wirth. The CRISP–DM Process Model. Technical report, The CRIP–DM Consortium NCR Systems Engineering Copenhagen, DaimlerChrysler AG, Integral Solutions Ltd., and OHRA Verzekeringen en Bank Groep B.V, March 1999. This Project (24959) is partially funded by the European Commission under the ESPRIT Program.

[OFG97] E. Osuna, R. Freund, and F. Girosi. An Improved Training Algorithm for Support Vector Machines. In J. Principe, L. Giles, N. Morgan, and E. Wilson, editors, *Neural Networks for Signal Processing VII — Proceedings of the 1997 IEEE Workshop*, pages 276–285, New York, 1997. IEEE.

[Pyl99] Dorian Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, 1999.

[Vap98] V. Vapnik. *Statistical Learning Theory*. Wiley, Chichester, GB, 1998.