Support Vector Machines in Relational Databases

Stefan Rüping*

*CS Department, AI Unit, University of Dortmund, 44221 Dortmund, Germany, E-Mail 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 SVM implementation 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.

Keywords. Support Vector Machines, Efficiency

1 Introduction

There exist many efficient implementations of Vapnik's Support Vector Machine [8] (see for example `http://www.kernel-machines.org/` for a list of available Support Vector software). 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. Relational databases are built upon a well-defined theoretical model of how data can be stored and retrieved and can deal with most questions that revolve around data in real-world settings, such as efficiency and effectiveness of storage and queries, security of the data, usability and handling of meta data.

On the other hand, available Support Vector 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. Depending on the problem, preprocessing steps have to be taken to clean and transform the data, that can be as complex as the final learning task itself [6],[2]. The same preprocessing steps have to be taken in order to apply the result to new examples.

In [4], Kietz et. al. describe that 50 - 80% of the efforts in real-world application of knowledge discovery are spent on finding an appropriate pre-processing of the data. They present a meta-data based framework to the re-use of KDD-applications that is centered on keeping as much data and data operations in the database as possible.

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. Slow network connections can have a serious impact on the overall runtime.

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 simple SQL statements to compute intermediate results whenever possible.

2 Support Vector Machines

The principles of Support Vector Machines and of statistical learning theory [8] are well known, so we give only a short introduction to the parts that are important in the context of this paper. In particular, we will only discuss Support Vector Machines for classification. See [8] and [1] for a more detailed introduction on SVMs and [7] for an introduction on SVMs for regression.

Support Vector Machines try to find a function $f(x) = wx + b$ that minimizes the expected Risk

$$R[f] = \int\int L(y, f(x))dP(y|x)dP(x) \tag{1}$$

by minimizing the regularized risk $R_{\text{reg}}[f]$, which is the weighted sum of the empirical risk $R_{\text{emp}}[f]$ with respect to the data $(x_i, y_i)_{i=1\ldots n}$ and a complexity term $||w||^2$

$$R_{\text{reg}}[f] = \frac{1}{2}||w||^2 + CR_{\text{emp}}[f].$$

This optimization problem can be efficiently solved in its dual formulation

$$\begin{aligned} W(\alpha) &= -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j x_i \cdot x_j + \sum_{i=1}^{n}\alpha_i \rightarrow \min && (2) \\ w.r.t. \quad & \sum_{i=1}^{n}\alpha_i y_i = 0 && (3) \\ & 0 \leq \alpha_i \leq C. && (4) \end{aligned}$$

The resulting decision function is given by $f(x) = \sum_{i=1}^{n} y_i \alpha_i x_i x + b$. It can be shown that the SVM solution depends only on it's support vectors $\{x_i | \alpha_i \neq 0\}$.

Support Vector Machines also allow the use of non-linear decision functions via the use of kernel function, which replace the inner product $x_i \cdot x_j$ by an inner product in some high dimensional feature space $K(x_i, x_j) = \Phi(x_i) \cdot \Phi(x_j)$. Then the decision function becomes $f(x) = \sum_{i=1}^{n} y_i \alpha_i K(x_i, x) + b$.

2.1 SVM Implementations

In practical implementations of Support Vector Machines it turns out that solving the quadratic optimization problem (2)-(4) with standard algorithms is not efficient enough, because these algorithms often require that the quadratic matrix $K = (K(x_i, x_j))_{1 \leq i,j \leq n}$ has to be computed beforehand and stored in main memory. Three tricks can speed up the calculation of the SVM solution dramatically.

Working set decomposition: To improve the efficiency of the SVM calculation, Osuna et. al. [5] suggest to split the problem into a sequence of simpler problems by fixing most variables and optimizing only on the rest, the so-called working set. This procedure is iterated until all variables satisfy the optimality conditions of the global problem. These optimality conditions, the Kuhn-Tucker conditions of the quadratic optimization problem (2)-(4), are essentially conditions on the gradient of the target function $W(\alpha)$ and on its Lagrangian multipliers. Joachims [3] proposes an efficient and effective method for selecting this working set.

Shrinking: Joachims also proposes two other improvements to the optimization problem. Usually most variables α lie at their boundaries 0 or C and tend to stay there from very early on in the optimization process. This is the case because usually the rough location of the decision boundary is found very early while most time is spent to find its exact location. Therefore, examples that lie far away from the decision boundary can be spotted easily. This is exploited by the idea of shrinking the optimization problem: Variables that are optimal at 0 or C for a certain number of iterations are fixed at that position and not re-examined in any further iteration.

Kernel caching: The third trick to improve SVM efficiency involves the caching of kernel functions. Both the selection of the working set and the check of the optimality conditions require the computation of the gradient ∇ of $W(\alpha)$. In fact, the computation of the gradient is the most expensive part of the SVM. The i-th component of the gradient itself is given by $\nabla_i = \sum_{j=1}^{n} \alpha_j K(x_j, x_i) - 1$. The values $s_i = \sum_{j=1}^{n} \alpha_j K(x_j, x_i)$ 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'.

Therefore, whenever a variable is updated, the kernel row $K_{i\cdot}$ is needed to incrementally update the gradient. As mostly only a certain subset of all variables gets into the working set at all, caching these kernel rows can significantly improve performance.

3 An SVM Implementation for Relational Databases

We will now show how an SVM can implemented on top of a relational database[1], that meets the following conditions:

1. It runs entirely inside the database, such that guarantees about the consistency as well as the security of the data can be given.
2. It does use as little main memory as needed for an efficient implementation. In particular, it does not duplicate the complete example set in its memory space.
3. It uses standard interfaces to access the database, so that it is database independent.
4. The evaluation of the decision function on new examples is as easy as possible.
5. It is as efficient as possible.

The first goal is achieved by the use of Java Stored Procedures. Java Stored Procedures allow to run Java programs inside the database server, that can directly access the database tables. To achieve the third goal, the JDBC standard is used to send simple SQL queries to the database.

In this section, we make the following assumption on the database schema: The examples $(x_i, y_i)_{i=1\ldots n}$ are stored in a single table where the d components of the vectors x_i are stored in the columns `att_1... att_d`. The index i is stored in a column `index`.

3.1 A Simple Approach

From the discussion in section 2 it is clear, that the only access to the examples' x-values 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 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.

3.2 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. For example, for the linear kernel $K(x_i, x_j) = x_i \cdot x_j$, 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>
```

[1]In the implementation of this work, an Oracle 8.1.6 database was used.

Similarly, a radial basis kernel function $K(x_i, x_j) = \exp(-\gamma||x_i - x_j||^2)$ can be expressed as the following SQL query:

```
select
exp(-<gamma>*(pow(X.att_1-Y.att_1,2) +...
              + pow(X.att_d-Y.att_d,2)))
from EXAMPLES
where X.index = <i> and Y.index = <j>
```

Here `<gamma>`, `<i>` and `<j>` stand for the actual values of γ, i and j. Note that for this query to be efficient, there needs to be a database index on the column *index*.

Most popular kernel functions, e.g. polynomial, neural network or anova kernels, depend on either the inner product or the euclidian distance of the example vectors, therefore it is possible to give corresponding SQL queries for these kernel functions as well.

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. Detailed information on these datasets and how the runtime was measured follows in section (4).

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

Comparing these results with we can see, that the new version is about 40% faster.

3.3 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

3.4 Shrinking

Shrinking has a big effect on runtime, because information on shrinked examples does not need to be updated in further iterations. This means, 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. There are two ways to do this: We could either add a column `shrinked` to the examples table and do the query

```
select
<KERNEL_SELECT>,
Y.index
from EXAMPLES X,
     EXAMPLES Y
where X.index = <i>
  and Y.shrinked = ’false’
```

(once again, `<KERNEL_SELECT>` stands for one of the select-statements defined in section (3.2)) or we can create a table `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)
```

The advantage of the first alternative is that the query can be done by a simple scan over the examples table with little extra cost and without adding new tables. On the other hand, this alternative requires that the user has the privileges to modify the examples table. This is a serious drawback in any situations, where data security is an issue. The second alternative does not suffer from this drawback.

3.5 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 $f(x) = \sum_{i \in SV} y_i \alpha_i K(x_i, x) + b$ on new examples. In this section we will show, that this can be simply done with pure SQL statements.

3.5.1 Linear Kernel: 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`.

3.5.2 General Kernels: In general kernels, we need the support vectors and their α-values to predict a new example. We assume that we still have the training examples in table `EXAMPLES` and we have a table `MODEL` that contains the values $y_i \alpha_i$ and the index of the corresponding vector (to simplify the calculation, $y_i \alpha_i$ is stored instead of α_i alone).

Then we can calculate $\sum_{i \in SV} y_i \alpha_i K(x_i, x)$ using the kernel select-statement and the SQL `sum` keyword as:

```
select sum(z.alpha * <KERNEL_SELECT>)
from MODEL Z, EXAMPLES X, EXAMPLES_TO_PREDICT Y
where X.index = Z.index
and Y.index = <i>
```

The value of b can be stored in the same table as the α's by using the index value `null`. Then the whole decision function is calculated by:

```
select alpha +
  (select sum(z.alpha * <KERNEL_SELECT>)
   from MODEL Z, EXAMPLES X, EXAMPLES_TO_PREDICT Y
   where X.index = Z.index
   and Y.index = <i>
  ) as f
from test_model
where alpha in (select alpha from test_model where key is null);
```

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.

The database experiments were made on a Sun Enterprise 250 that was equipped with a double UltraSparc II 400MHz processor and 1664 MB of main memory, running an Oracle 8.1.6

database. The C++ experiment were made on a Sun Ultra with a UltraSparc IIi 440 MHz processor and 256 MB of main memory. As the kernel cache was kept at a size of 40 MB in all experiments, the different memory equipment should not influence the results.

Three datasets were used in the comparison. The first data set PAT consisted of a simple artificial classification task with 100 examples and a linear target function. The second data set REG is an artificial regression problem with 2000 examples and a linear target function. The third data set, CYC is a real-world dataset of 157 examples. The task is to classify the state of the german business cycle (upswing or downswing) from several economic variables. A radial basis kernel with parameter $\gamma = 1$ was used in the experiments. The data sets are summarized in the following table

Name	Size	Dimension	#SVs
Pat	100	27	47
Reg	2000	27	56
Cyc	157	13	157

To get clearer results, 5-fold cross-validation was done on each of the data sets and the CPU time of each learning run was recorded. In each learning run, the resulting decision functions of both implementations were equal up to sensibly small numerical errors.

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
Cyc	0.24s	0.13s	0.37s	16.46s	44.48

The experiments show, that the database version is slower than the standalone version by a factor of 20 to 45. If this difference is acceptable has to be evaluated with respect to the individual application's requirements.

5 Discussion and Further Research

In this paper we made the assumption that the data is given in a database table in attribute-value form. While this may be the most prominent way of representing examples, there are other representations, that have interesting properties.

5.1 Sparse Data Format

For data that is sparse, i.e. many attributes are zero (e.g. text data), the attribute-value format is not optimal because too much space is lost storing zeros. Also, in SVM kernel calculation much time is spent in unnecessary numerical operations, because attributes where both (in the case of kernels based on the euclidian distance) or even one (for kernels based on the inner product) value is zero, do not amend to the value of the kernel function on the respective examples.

Therefore SVM software, e.g. SVM^{light} [3], often stores examples in a format where only the non-zero values of the examples vector together with their attribute number are stored. In relational databases, this format could be used in form of a table that consists of the columns `example_id`, `attribute_id` and `attribute_value`.

Following the earlier discussion, to show that SVMs with the most commonly used kernel functions can be efficiently trained on data in this representation, it suffices to show that the inner product and the euclidian distance of two examples can be calculated in this representation.

For the inner product is suffices to sum up all the products of attribute values where the examples have the given ids and the attribute indices are equal. In SQL:

```
select sum(x.attribute_value * y.attribute_value)
from EXAMPLES x, EXAMPLES y
where x.example_id = <i>
  and y.example_id = <j>
  and x.attribute_id = y.attribute_id;
```

To calculate the squared euclidian distance, first the squared distance of all attributes that exists in both the vectors x_i and x_j can be calculated in a similar way to that of the inner product. In fact, only the `select`-part of the statement has to be adapted. Then the squared distance of all attributes that exist in vector x_i but not in vector x_j to the vector 0 can be calculated by

```
select sum(x.attribute_value * x.attribute_value)
from sparse x
where x.example_id = <i>
  and not exists (
    select attribute_id
    from sparse
    where attribute_id = x.attribute_id and example_id = <j>
    )
```

Then the results of the three query of attributes in x_i and x_j, x_i without x_j and x_j without x_i can be added up to give the final result.

5.2 Joins

In relational databases, data is typically not stored in one but in multiple relations. For example, a clinical information system may store minutely recorded vital signs of its intensive care patients together with demographic data like age, sex or height that do not change during a patients stay or even information about drug ingredients that are invariant over all patients. In attribute-value representation, for example the patients age would have to be stored over and over again for all time-points where a vital sign was recorded. In a relational database, this information would be typically stored in three tables `vital_signs`, `demographic` and `drug_ingredients`.

As the SVM cannot deal with multi-relational data, the different tables would have to be joined together for the SVM to access them, e.g. like

```
select *
from  vital_signs, demographic
where vital_signs.patient_id = demographic.patient_id
```

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 Support Vector Machines. 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: $(x_M, x_N)\cdot(y_M, y_N) = x_M \cdot y_M + x_N \cdot y_N$. A similar observation holds for the euclidian distance: $||(x_M, x_N) - (y_M, y_N)||^2 = ||x_M - y_M||^2 + ||x_N - y_N||^2$.

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 or the euclidian distances, respectively, 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.

5.3 Discussion

This paper proposed an implementation of a Support Vector Machine 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.

Careful analysis and optimization has 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.

References

1. Burges, C.: A Tutorial on Support Vector Machines for Pattern Recognition. Data Mining and Knowledge Discovery**2** (1998) 121–167
2. Chapman, P., Clinton, J., Khabaza, T., Reinartz, T., Wirth, R.: The CRISP–DM Process Model. The CRIP–DM Consortium (1999)
3. Joachims, T.: Making large-Scale SVM Learning Practical. In: Schölkopf, B., Burges, C., Smola, A. (eds): Advances in Kernel Methods - Support Vector Learning. MIT Press, Cambridge, MA (1999)
4. Kietz, J., Zücker, R., Vaduva, A. Mining Mart: Combining Case-Based-Reasoning and Multi-Strategy Learning into a Framework to reuse KDD-Applications. In: Michalki, R.S., Brazdil, P. (eds): Proceedings of the fifth International Workshop on Multistrategy Learning (MSL2000). Guimares, Portugal (2000)

5. Osuna, E., Freund, R., Girosi, F.: An improved training algorithm for support vector machines. In: Principe, J., Giles, L., Morgan, N., Wilson, E. (eds.): Neural Networks for Signal Processing VII — Proceedings of the 1997 IEEE Workshop. IEEE, New York (1997) 276–285
6. Pyle, D.: Data Preparation for Data Mining. Morgan Kaufman (1999)
7. Smola, A., Schölkopf, B.: A tutorial on support vector regression Technical Report, NeuroCOLT2 Technical Report Series (1998)
8. Vapnik, V.: Statistical Learning Theory. Wiley, Chichester, UK (1998)