Efficient Kernel Calculation for Multirelational Data

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 database systems or in data warehouses built on top of relational database systems. Often, for more data is available than can be processed by standard learning algorithms in reasonable time. This paper presents an extension to kernel algorithms that makes use of the more compact relational representation of data instead of the usual attribute-value representation to significantly speed up the kernel calculation.

Keywords. Support Vector Machines, Efficiency

1 Introduction

Today, most of the data in business applications is stored in relational database systems or in data warehouses built on top of relational database systems. 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.

Cheap storage space and the efficiency of modern database systems in storing and querying data have led to the creation of very large databases, that contain the complete business information of large companies. The task of knowledge discovery in databases is to find hidden knowledge in this data, that may be helpful to better understand and optimize the companies businesses.

As the task of knowledge discovery requires to process extremely large amounts of data, many useful machine learning algorithms cannot be applied, because they were developed for much smaller data sets and do not scale well enough to deal with gigabytes of data. Often, in this case sampling is used in the hope to generate a subset of the data, that is small enough to be processed by the learning algorithm but still reflects the original data close enough to give acceptable results.

To increase the performance and flexibility of data mining applications, research is currently done to move as much of the data mining work into the database to avoid costly transport of data between database servers and application machines. This targets especially at the step of data preprocessing to clean and transform the data. This step can be as complex as the final learning task itself [8, 2]. Even worse, 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.

1.1 Learning and Representation

The relational data model specifies that data is kept in relations. A relation is a set of tuples where each attribute value in the tuple is a member of a fixed domain. In practice, relations are stored in database tables, where each table row defines a tuple of the relation and each table column defines an attribute of the relation, where the attribute domain is given by a fixed column type. Ideally, each relation stands for a certain real-world concept, that cannot be split up into meaningful sub-concepts, e.g. a bank customer (given by name, address and customer number), a banking accout (given by customer number, account number and credit limit) or an account transaction (given by two account numbers and an amount of money).

The trick with multirelational data is, that the tables do not have to be taken on their own, but can be combined to query the data in very complex ways. The relational algebra which describes the semantics of database queries – implemented in the standard query language SQL – is based on three main operators: selection, projection and join. A selection selects tuples from a relation with respect to different criteria. Projection selects attributes out of a relation. A join combines the data of two different relations based on the equality of some specified attributes. While selection and projection decreases the size of the data, a join of two tables of size n and m can produce a table of size $n \cdot m$.

So why is that a problem for data mining? With the notable exception of Inductive Logic Programming [6], most learning algorithms cannot deal with multirelational data but are based on attribute-value representation of the data. To generate this representation, all the information that is necessary for learning has to be compiled into a single relation, which means building up a complex query with possibly many joins. Think of combining the personal and account information of a bank costumer with every of his transactions to build up a data set to detect fraud. By this tranformation, the concise and usually very natural multirelational representation is bloated to a large, redundant single-relational representation. That is, the size of the data the learner has to handle is very much increased.

In this paper, an algorithmic solution is presented that allows for certain types of learning algorithms – learning algorithms based on kernel functions – to make use of the multi-relational structure behind the attribute-value representation to increase the efficiency of the training. The discussion is restricted to the case of joining two or more tables. The extension to the case of constructing an attribute-value representation using also selection and projection is straight-forward.

The next chapter will give an introduction to Support Vector Machines (SVMs) as the most prominent representative of the class of kernel machines. Especially, the problem of efficiently solving the SVM problem will be discussed. Chapter 3 will introduce the idea of kernel evaluation on joined data and Chapter 4 will give experimental results.

2 Kernel Machines

2.1 Support Vector Machines

The principles of Support Vector Machines and of statistical learning theory [12] 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 [12] and [1] for a more detailed introduction on SVMs and [11] for an introduction on SVMs for regression.

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

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

of the learner 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 $(x_i, y_i)_{i=1\dots n}$ 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

$$\begin{aligned} W(\alpha) \quad &= \quad -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j x_i \cdot x_j && (2)\\ & \quad\quad + \sum_{i=1}^{n}\alpha_i \to \min && (3)\\ w.r.t. \quad & \quad\quad \sum_{i=1}^{n}\alpha_i y_i = 0 && (4)\\ & \quad\quad 0 \le \alpha_i \le C. && (5)\end{aligned}$$

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 its support vectors $\{x_i | \alpha_i \neq 0\}$.

2.2 Kernels

Support Vector Machines also allow the use of nonlinear 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 $f(x) = \sum_{i=1}^{n} y_i \alpha_i K(x_i, x) + b$.

Popular kernel functions are the radial basis kernel

$$K_\gamma(x, y) = \exp(-\gamma ||x - y||^2),$$

the polynomial kernel

$$K_d(x, y) = (x * y + 1)^d$$

or the neural net kernel

$$K_{a,b}(x, y) = \tanh(a \cdot x * y + b).$$

Actually, almost every kernel function, that is practically used, is a function of either the linear product of the euclidian distance of two examples, $K(x, y) = f(x * y)$ or $K(x, y) = f(||x - y||)$.

2.3 SVM Implementations

In practical implementations of Support Vector Machines it turns out that solving the quadratic optimization problem (2)-(5) 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. [7] 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)-(5), 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 $W(\alpha)$. 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 variable i is updated, the kernel row $K_{i\cdot} = (K(x_i, x_1), \ldots, K(x_i, x_n))$ 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. Usually a least-recently-used cache strategy is used for this.

For optimization of Support Vector Machines, the important observation is that calculating the kernel function is the most expensive part of training Support Vector Machines.

2.4 Kernel Machines

The trick of replacing the linear product by a kernel function to increase the hypothesis space of a learning algorithm to a much greater class of non-linear functions has been applied to other learning than Support Vector Machines as well, for example to Principal Component Analysis [10] or Kernel Fisher Discriminant Analysis [5]

For these algorithms, the same performance arguments for the evaluation of kernel function apply as for SVMs.

3 Efficient Kernel Evalutation on Joined Data

As already said, the compilation of multirelational data into a single relation is bloating up the concise multirelational representation considerably. When joining two tables, in the worst case every row of the first table is joined with every row of the second table. This means, the same piece of information of a row in the original table is used over and over again in the final, single table. But what if we could make use of the original data instead of the large final data?

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 means, 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 of the vectors $x^{(M)}$ and $x^{(N)}$, 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 kernel cache as two independent caches.

See for example the data set given in Figure 1. It consists of seven five-dimensional examples, so to hold its entire kernel matrix, seven kernel rows have to be cached. But actually, this data set can be viewed as a join of two tables, where the first table contributes the attributes $\{$x1, x2, x3$\}$ and the second tables contributes the attributes $\{$x4, x5$\}$. These tables are shown in Figure 2. To hold the respective kernel matrixes of both tables, a total of only six rows has to be cached.

y	x1	x2	x3	x4	x5
1	0.1	-0.3	0.2	0.3	-0.5
1	-0.4	0.2	0.1	0.7	0.6
-1	0.1	-0.3	0.2	0.7	0.6
1	-0.2	0.9	-0.5	0.3	-0.5
1	-0.4	0.2	0.1	-0.8	0.1
-1	-0.2	0.9	-0.5	0.7	0.6
-1	-0.2	0.9	-0.5	-0.8	0.1

Figure 1 Example Data Set.

x1’	x2’	x3’
0.1	-0.3	0.2
-0.4	0.2	0.1
-0.2	0.9	-0.5

x1”	x2”
0.3	-0.5
0.7	0.6
-0.8	0.1

Figure 2 Join Subsets of the Example Data Set.

To reconstruct the data set from Figure 1, we also need the information of Figure 3 of how to combine the two individual tables. Altogether, the storage of the complete table requires to store 42 values for the data plus 7 kernel rows of dimension 7, while the storage of the decomposed join requires to store only 36 values for the data and 6 kernel rows of dimension 3 and 2, respectively.

To compute a kernel row $K_{i\cdot} = (K(x_i, x_1), \ldots, K(x_i, x_n))$, we first compute the rows of inner products $(x_i^{(M)} * x_1^{(M)}, \ldots, x_i^{(M)} * x_n^{(M)})$ and $(x_i^{(N)} * x_1^{(N)}, \ldots, x_i^{(N)} * x_n^{(N)})$ (or squared euclidian distances, as the case might be) in the subspaces given by the attributes in M and N. The rows corresponding to example i in this subspaces are cached to avoid their re-computation, as many of the original examples may have the same projection in one of the subspaces (e. g. think of the bank transaction example, where the examples belonging to one customer can be projected to the same customer information M, but different transaction information N).

The final kernel row can then be calculated from this

y	row1	row2
1	1	1
1	2	2
-1	1	2
1	3	1
1	2	3
-1	3	2
-1	3	3

Figure 3 Join Information of the Example Data Set.

kernel rows, by each adding up two entries from this rows and eventually applying a kernel-specific function f to these values (see section 2.2). All that needs to be known to combine the single kernel rows to the joined kernel row are the mappings that map an index i of an example in the join table to the indexes of its components in each of the component tables (as shown in Figure 3). This mapping can be easily computed given the query that would be used to generate the join data. Actually, the mapping is generated by the same query, just that not all the data but only the corresponding index values are used.

3.1 Cache Strategies

Assuming the learning algorithm may only use some maximal amount of cache memory, there are different strategies how the memory can be split up between the different kernel caches.

The easiest cache strategy is to split up the available cache memory evenly between the caches. A more clever way would be also possible to split up the cache memory depending on the size of the data from each kernel. This would ensure that each of the sub-kernels can cache the same fraction of rows. Assuming that the kernel rows that need to be cached are distributed evenly over the kernel rows of each of the sub-kernels, this would be an optimal cache strategy, as there would be an equal probability of a cache miss in every kernel.

One could also distribute the overall available cache memory among the sub-kernels dynamically. Whenever a new kernel row has been computed an there is not enough space left in the cache, the least-recently-used cache row of all sub-kernel caches is moved out of the cache. This would be useful in the case where only a very limited number of kernel rows from one sub-kernel is ever used while much more kernel rows from the other sub-kernels are needed. Actually, this situation has an interesting link to feature selection for SVMs: The more important the features of one sub-kernel are, the less kernel rows of this kernel will be needed in later iterations, because most of the values of this features that make an example lie far away from the decision boundary can be recognized easily very early in the optimization process.

As the computation of the overall kernel row from the rows of the sub-kernels is not trivial, it may be a good idea to use a two-level caching approach: All available memory that is not used to cache rows of the sub-kernels can be used to cache additional rows of the overall kernel. Especially for high dimensional data, the performance gain by not needing to recompute the cache rows will exceed the overhead of having to maintain two cache structures by far.

Test no.	1	2	3	4	5
Cache (kB)	512	1024	2048	3072	4096

Test no.	6	7	8	9
Cache (kB)	5120	6144	7168	8192

Figure 4 Cache Size in kB in the tests.

4 Experiments

For the experiments, an artificial data set was generated that consisted of 1000 examples drawn from the cartesian product of two tables of 100 examples with dimension 1000 each. The examples where classified with a linear decision function with 1% of noise and correspondingly, a linear kernel was used. The SVM implementation mySVM [9] was used in the experiments.

The high dimensionality of the examples was chosen to make the calculation of the inner product between two examples costly, such that cache misses will have a high impact on runtime. However, the results are valid regardless of the dimension of the examples, because the size of the kernel matrix – and therefore the caching process – is independent of the dimension of the examples. In terms of runtime, the only influence of the examples dimension is a linear factor when the inner product is calculated.

In the final SVM solution, 380 out of the 1000 examples ended up as support vectors. In a first experiment, a standard SVM was compared to a SVM using caching of the sub-kernels with a fixed, evenly split cache size. The overall cache size was varied between 8 MB and 0.5 MB (see the table in Figure 4). A quick calculation shows, that caching the complete kernel matrix on the level of the joined data would need 7.6 MB of cache and caching all kernel rows corresponding to support vectors would need 2.9 MB of cache.

In Figure 5, the average runtime of two SVM implementations is compared. The line labeled ”global cache” shows the runtime of a usual SVM implementation, that caches the kernel rows over all attributes. We see, that for small cache sizes, the runtime increases dramatically (for 512kB of cache 8618s, almost 2.5 hours). For large enough cache sizes (4 MB or more), the runtime stays constant at about 114s. In the later case, the cache was large enough to contain the whole kernel matrix, such that no kernel values had to be re-computed. The line labeled ”local cache” shows the performance of a SVM that uses an own cache of the inner products for the attributes in each part of the join. Here, the runtime stays constant at about 118s for all tested cache sizes. This means, even

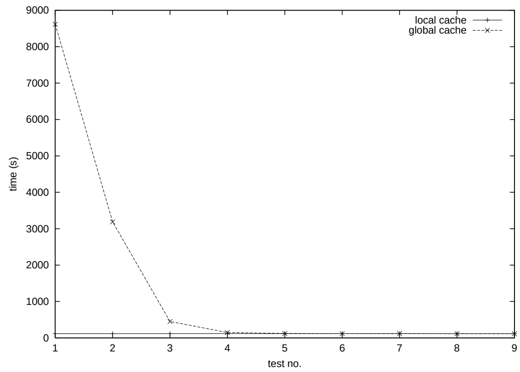

Figure 5 Comparison of the average runtime of the local and the global caching approach.

the smaller cache sizes were still large enough to hold the complete sub-matrixes. In this case, the runtime is dramatically reduced compared to the ”global cache” SVM!

In the experiments, the ”local cache” SVM was slightly slower than the ”global cache” with full cache (118s compared to 114s). This small difference is not the result of a statistical error but was to be expected: getting a kernel row in the ”local cache” SVM involves combining the kernel rows returned from the subcaches into a single row, which means one addition for each example in the training set. In the ”global cache” SVM, the row has only to be read from the cache, which can be done in constant time.

But what if we combined both caching strategies? We saw that the cache sizes for a full subcache are very small, compared to the complete kernel cache. This means, for all but very small total cache sizes, there is still enough space to cache some of the kernel rows on the global level. Figure 6 compares the runtime of both approaches. Here, the runtime with the combined cache approach was about one half to one third of the runtime of the local approach, depending on the total size of the cache.

5 Conclusion

In this paper, a caching algorithm for kernel machines was presented, that makes use of relational structures in the data. This allows for a much more efficient and compact calculation of the kernel values compared to the usual attribute-value representation. The cache algorithm was tested for SVMs, but can be used for other kernel algorithms as well.

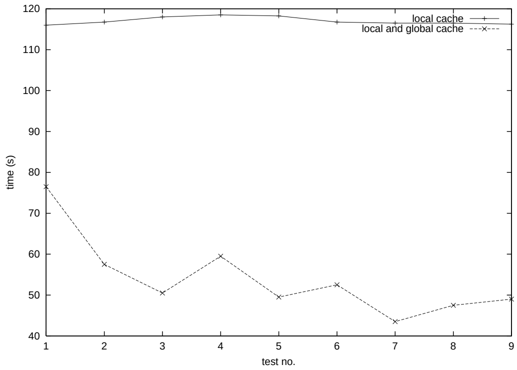

Figure 6 Comparison of the average runtime of the local and the combined global and local caching approach.

Acknowledgments

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

References

1. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.
2. 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.
3. T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11. MIT Press, Cambridge, MA, 1999.
4. Jörg-Uwe Kietz, Regina Zücker, and Anca Vaduva. Mining Mart: Combining Case-Based-Reasoning and Multi-Strategy Learning into a Framework to reuse KDD-Application. In R.S. Michalki and P. Brazdil, editors, *Proceedings of the fifth International Workshop on Multistrategy Learning (MSL2000)*, Guimares, Portugal, May 2000.
5. Sebastian Mika, Gunnar Rätsch, Jason Weston, Bernhard Schölkopf, and Klaus-Robert Müller. Fisher discriminant analysis with kernels. In Y.-H. Hu, J. Larsen, E. Wilson, and S. Douglas, editors, *Neural Networks for Signal Processing IX*, pages 41–48. IEEE, 1999.
6. Stephen Muggleton, editor. *Inductive Logic Programming*. Academic Press, 1992.
7. 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.
8. Dorian Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, 1999.
9. Stefan Rüping. *mySVM-Manual*. Universität Dortmund, Lehrstuhl Informatik VIII, 2000. http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/.
10. Bernhard Schölkopf, Robert C. Williamson, Alex J. Smola, and John Shawe-Taylor. Sv estimation of a distribution's support. In S.A. Solla, T.K. Leen, and K.-R. Müller, editors, *Neural Information Processing Systems 12*. MIT Press, 2000. forthcoming.
11. Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. Technical report, NeuroCOLT2 Technical Report Series, 1998.
12. V. Vapnik. *Statistical Learning Theory*. Wiley, Chichester, GB, 1998.