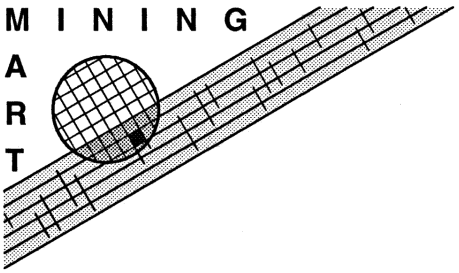

Enabling End-User Datawarehouse Mining
Contract No. IST-1999-11993
Deliverable No. D14.3

Feature Selection with Support Vector Machines

Timm Euler

University of Dortmund, Computer Science VIII
D-44221 Dortmund, Germany
euler@ls8.cs.uni-dortmund.de
http://www-ai.cs.uni-dortmund.de

December 19, 2002

Abstract

This deliverable describes the usage of the Support Vector Machine (SVM) algorithm for automatic feature selection. After explaining SVMs and the approach to feature selection in general (chapter 1), the work done for implementing this algorithm and a feature selection operator in the MiningMart system is described in chapter 2.

Chapter 1

Feature Selection with Support Vector Machines

This chapter has three sections. The first explains Support Vector Machines (SVMs, section 1.1) in general. The second (1.2) deals with one of their specific properties, namely the possibility to estimate the generalisation error of an SVM after one training run. The last section (1.3) describes how this specific property can be used to perform automatic feature selection in a Machine Learning scenario.

1.1 Support Vector Machines

Support Vector Machines (SVMs, [CV95]) are a development from statistical learning. They are based on the idea of *structural risk minimisation* ([Vap82]). The following explanations are based on [Joa01].

The learning scenario is formalised as follows. Let S be a training set of N examples represented by $\vec{x}_1, \ldots, \vec{x}_N$ from the vector space $X = \mathbb{R}^n$. Each of these vectors is associated with a class y from a set Y:

$$S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_N, y_N)\}$$

Here we only need the simplest case of binary classification, so we choose $Y = \{1, -1\}$. The aim is to learn a function or hypothesis $h : X \longrightarrow Y$ from a hypothesis space H such that the probability that an error is made on a randomly drawn example is as low as possible. Assuming an unknown distribution $\Pr(\vec{x}, y)$ of examples, we want to minimise

$$Err(h) = \Pr(h(\vec{x}) \neq y \,|\, h) = \int L(h(\vec{x}), y)\, d\Pr(\vec{x}, y)$$

where L is a simple loss function

$$L(h(\vec{x}), y) = \begin{cases} 0 & h(\vec{x}) = y \\ 1 & else \end{cases}$$

Structural risk minimisation is based on the fact that the error rate $Err(h)$ of a hypothesis can be related to the complexity of H and the training error rate $Err_{tr}(h)$. The complexity of H is the VC dimension d, defined as the maximum number of examples that a function from H can separate correctly given an arbitrary classification of the examples.

The following bound ([Vap98]) gives the relation between error rate and complexity d of an hypothesis h, where N is the number of training examples and $1-\eta$ is the probability that the bound holds.

$$Err(h) \leq Err_{tr}(h) + O\left(\frac{d\ln\left(\frac{N}{d}\right) - \ln(\eta)}{N}\right) \tag{1.1}$$

This means that the true error $Err(h)$ is dependent on the training error on the one hand and on the complexity of the used learning functions on the other hand. Thus, simple functions cannot usually lead to a good training error because they cannot separate the examples well enough. In contrast, very complex functions give good training errors, but also high values for the right hand side of the bound above (overfitting). In both cases, the bound is rather loose.

The structural risk minimisation approach is therefore to choose a structure of nested hypothesis spaces with increasing complexity:

$$H_1 \subset H_2 \subset \ldots \subset H_i \subset \ldots \quad \text{where} \quad \forall i : d_i \leq d_{i+1}$$

Then the task is to choose the index i such that equation 1.1 is minimised. The Support Vector Machine approach is to choose, from the hypotheses that solve the learning task, the one from the H_i with lowest i, which is explained in the following.

In general, Support Vector Machines find a hyperplane that separates the training set according to the given classification[1]. There can be several such hyperplanes; SVMs choose the one that maximises the distance to the nearest points (see figure 1.1). This distance is the so-called *margin*. The reason for maximising the margin is that a high margin corresponds to a low VC dimension of the separating hyperplane ([Vap82]). This is formalised as follows. The hyperplane to be found is of the form $\vec{w}\cdot\vec{x}+b=0$ with normal vector $\vec{w}$ and distance to the origin $b/\|\vec{w}\|$. Thus what we need to find is the zero of a function f with $f(\vec{x}) = \vec{w}\cdot\vec{x}+b$, where all training examples $\vec{x}_i$ are separated correctly:

$$y_i(\vec{w}\cdot\vec{x}_i+b) > 0 \quad \forall i = 1,\ldots,N$$

But the function f is not uniquely fixed by this. By also requring

$$y_i(\vec{w}\cdot\vec{x}_i+b) \geq 1 \quad \forall i = 1,\ldots,N \tag{1.2}$$

[1] We assume for the moment that such a hyperplane can be found. Below, we will return to the case where this is not true.

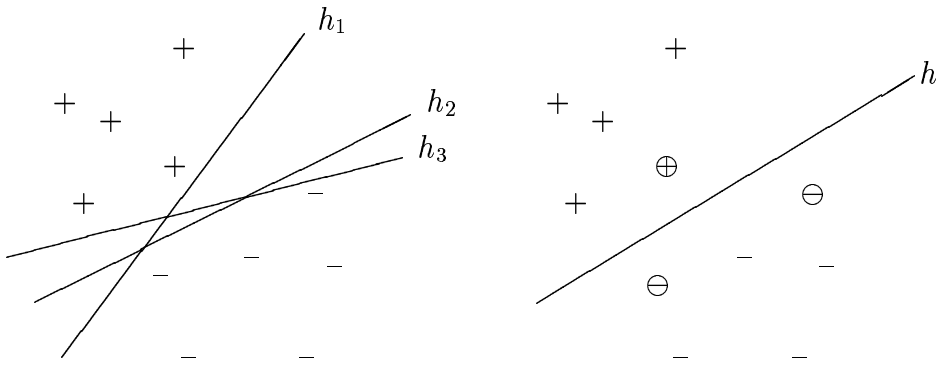

Figure 1.1: Separation of points by lines in the two-dimensional plane. On the left several separators, on the right the one with maximal distance to the nearest points (called support vectors; circled).

---

the function is fixed and a certain minimal distance δ of the nearest points to the hyperplane is enforced. These points are called *Support Vectors*, they define the margin δ. They alone determine the position of the separating hyperplane. They lie on hyperplanes parallel to the first one and, due to (1.2), they can be written in the form

$$\begin{aligned} \vec{w} \cdot \vec{x}_i + b &= 1 \quad \text{for } y_i = 1 \\ \vec{w} \cdot \vec{x}_i + b &= -1 \quad \text{for } y_i = -1 \end{aligned}$$

So the distance of these parallel hyperplanes to the origin is $|1 - b| \,/\, \|\vec{w}\|$ or, respectively, $|-1 - b| \,/\, \|\vec{w}\|$, thus their distance to the first hyperplane, the margin, is $\delta = 1 \,/\, \|\vec{w}\|$. So by minimising $\|\vec{w}\|$ one maximises the margin.

The relation between high margin and low VC dimension is achieved by the following lemma. Let all example vectors $\vec{x}_i$ be contained in a ball with radius R and let $|\vec{w} \cdot \vec{x}_i + b| \geq 1$ hold for them. Then the set of hyperplanes $h(\vec{x}) = sign\{\vec{w} \cdot \vec{x} + b\}$ in $\mathbb{R}^n$, seen as hypotheses, has a VC dimension d which is bounded by

$$d \leq min(\left[\frac{R^2}{\vec{w}^2}\right], n) + 1 \,.$$

Thus the VC dimension is dependent on $\|\vec{w}\|$, the euclidian length of the normal vector $\vec{w}$ of the separating hyperplane. In sum, Support Vector Machines thus have to solve the following optimisation problem:

$$\begin{aligned} &\text{Minimise:} \quad V(\vec{w}, b) = \|\vec{w}\| \\ &\text{subject to:} \quad y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1 \,, \;\; \forall i = 1, \ldots, N \end{aligned} \tag{1.3}$$

To predict the class of unseen examples $\vec{x}$, we compute on which side of the hyperplane they are:

$$y = sign(\vec{w} \cdot \vec{x} + b) \,.$$

However, optimisation problem (1.3) does not have a solution if no separating hyperplane exists. Thus we must allow training errors to occur in general. This is done by including an upper bound on the number of training errors in the objective function of (1.3). Then this upper bound and the length of $\vec{w}$ are minimised simultaneously:

$$\begin{array}{ll} \text{Minimise:} & V(\vec{w}, b, \vec{\xi}) = \|\vec{w}\| + C \sum_{i=1}^{N} \xi_i \\ \text{subject to:} & y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1 - \xi_i \,, \\ & \xi_i > 0 \,, \ \forall i = 1, \ldots, N \end{array} \tag{1.4}$$

The ξ_i are called slack variables. If a training example is on the wrong side of the hyperplane, the correspnding ξ_i is greater than 1. Therefore $\sum_{i=1}^{N} \xi_i$ is an upper bound on the number of training errors. The factor C above is a parameter that allows to trade off training error vs. model complexity. A small value for C will increase the number of training errors, while a large C will lead to a behaviour similar to that of the previous optimisation problem.

Since the two optimisation problems (1.3) and (1.4) can be numerically difficult, the Wolfe dual form is used which can be solved efficiently. This is given below for (1.4).

$$\begin{array}{ll} \text{Minimise:} & W(\vec{\alpha}) = -\sum_{i=1}^{N} \alpha_i + \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j (\vec{x}_i \cdot \vec{x_j}) \\ \text{subject to:} & \sum_{i=1}^{N} y_i \alpha_i = 0 \,, \\ & 0 \leq \alpha_i \leq C \,, \ \forall i = 1, \ldots, N \end{array} \tag{1.5}$$

From the solution of this problem, the classification rule can be computed as

$$\begin{array}{lll} \vec{w} \cdot \vec{x} & = \sum_{i=1}^{N} \alpha_i y_i (\vec{x}_i \cdot \vec{x}) & \text{and} \\ b & = y_{sv} - \vec{w} \cdot \vec{x}_{sv} & \end{array} \tag{1.6}$$

where $\vec{x}_{sv}$ is any support vector with $0 < \alpha_i < C$. It can be seen that the resulting weight vector of the hyperplane is constructed as a linear combination of the training examples. Only support vectors have a coefficient α_i that is non-zero.

In summary, Support Vector Machines find for a given set of training examples the hyperplane that separates them best and maximises the distance to the nearest examples, because this hyperplane has the lowest VC dimension, so the structural risk is minimised. A parameter C can be used to trade off the training error against the complexity of the learned model. During training, a vector $\vec{\xi}$ and a vector $\vec{\alpha}$ are computed which describe each training example.

1.2 Estimation of the generalisation error of SVMs

This section deals with a specific property of SVMs that allows to estimate their generalisation error after one training run. This property was discovered by Thorsten Joachims and is described in more detail in [Joa01]. The proofs of the claims made here can be found there.

Usually, after training a learner, its performance can only be determined on a separate set of examples that were not used for training, but whose classes are known. By comparing the known classes to the predicted ones, an empirical error is found and taken as the true error of the learned model. The closest one can get to the true error is by using all classified examples for training except for one, testing on this one example and repeating this for all examples. Averaging over the single errors renders the so-called leave-one-out error. The problem is that for N training examples, this requires N learning runs which is usually not feasible. In practice, the number of examples held out for testing is often increased to N/j, and this is repeated j times with disjunct test sets. So the number of learning runs is reduced to j, where often $j = 10$ is chosen. This process is called cross-validation.

This scenario is applicable to any learner. However, in this section a method to estimate the empirical error is presented which works only for Support Vector Machines. The method is called $\xi\alpha$-estimation because its inputs are the two vectors $\vec{\xi}$ and $\vec{\alpha}$ described in the last section. What follows is a slightly simplified definition of the $\xi\alpha$-estimator.

Definition 1 Let $\vec{\xi}$ and $\vec{\alpha}$ be the vectors computed during a training run of the Support Vector Machine as described in section 1.1, optimisation problem 1.5. The **$\xi\alpha$-estimator of the error rate** for a hyperplane h is

$$Err_{\xi\alpha}(h) = \frac{d}{N} \text{ with } d = |\{i : (\alpha_i R^2 + \xi_i) \geq 1\}| \tag{1.7}$$

where N is the number of training examples and R^2 is an upper bound on the kernel function evaluated on any pair of examples.

This definition refers to the kernel function of the SVM. In section 1.1, only linear SVMs were presented in which the kernel function equals the dot product. In [Joa01], other kernel functions and their use are presented.

The key measure in this definition is obviously d. It counts the number of examples for which the inequality $(\alpha_i R^2 + \xi_i) \geq 1$ holds. There is a connection between this inequality and those examples that can produce a leave-one-out error if they are not used for training, but for testing. More precisely, if an example $(\vec{x}_i, y_i)$ is not classified correctly by a SVM trained on a sample *without* it, then for this example the inequality must hold for a SVM trained on the sample *with* it. Therefore, all examples for which the inequality does not hold do not produce a leave-one-out error. So the $\xi\alpha$-estimator is an approximation to the leave-one-out error which is never too low, i.e. never too optimistic. It can be computed during the training run of a SVM at no extra cost. Empirical tests have shown that the estimator is often, but not always, tight enough for applications. In particular for text data it works well. Details and proofs can be found in [Joa01].

1.3 Feature Selection using the estimated error

Real-world machine learning tasks often involve a large number of examples using a complex representation both for the examples themselves and the hypotheses to be learned. This makes a number of tasks computationally rather demanding. One remedy can be the simplification of example representation. Referring back to the learning scenario presented in section 1.1, this can mean to change the representation of examples from elements of $\mathbb{R}^n$ to elements of $\mathbb{R}^m$ with $m < n$. If this is to be done automatically, the simplest way is probably to select m dimensions from the n given ones according to some criteria, and remove the other dimensions. This is the task of feature selection. It is generally described in MiningMart deliverable 14.1, and the following descriptions are based on this.

As explained in deliverable 14.1, there are basically two approaches to feature selection called the *wrapper* and the *filter* approach. The latter uses data characteristics and is not dependent on the learning method, while the former tunes the feature selection to the learning method used and is therefore in general more effective, but also computationally more demanding.

Feature selection with Support Vector Machines is based on the wrapper approach. The general wrapper scenario is the following. Let F_n be the set of n given features. We search for a subset of F_n which reduces the computational time needed for learning without losing generalisation performance, possibly even improving generalisation performance. Given a subset F_m with m features, $m \leq n$, a learning run is started using the example representation based on F_m. The learned model is then evaluated. Since a proper evaluation involves cross-validation (see section 1.2), in fact j learning runs are needed for the evaluation of feature set F_m (where usually $j = 10$). In this way an error is associated with the feature set F_m. This error can guide the search for other feature subsets. Please refer to deliverable 14.1 for further explanations, in particular the forward and backward selection search methods.

The crucial point in terms of computational complexity are the j learning runs that are needed to evaluate one feature set. Since there are 2^n subsets of F_n, the search must cover a large space in which the j learning runs are needed at every node. So reducing the computational complexity at such a node will reduce the overall computation time greatly. At this point the special property of Support Vector Machines described in the previous section can be exploited. Using $\xi\alpha$-estimation, an (estimated) error can be associated with each feature subset after only one learning run. This error can be used to guide the search for the best feature subset in the same way as the empirical error found in cross-validation; it reduces the time for feature selection by the factor j with respect to the general wrapper scenario.

This approach was implemented and tested in the work done for this deliverable, as described in the following chapter.

Chapter 2

Feature Selection in the MiningMart System

This chapter describes first some work done to make the MiningMart system able to deal with automatic feature selection (section 2.1). Second, it explains how the MiningMart operator *FeatureSelectionWithSVM* was implemented (section 2.2). The last section (2.3) briefly describes a small experiment that was used to test the implementation and the general validity of the approach.

2.1 Automatic Feature Selection in MiningMart

In the MiningMart system, a preprocessing case is described on the conceptual level using a graphical tool, the Case editor. The idea is to conceptually design a complete chain of steps that process the data. Then the MiningMart compiler translates these steps to the relational level, i.e. to the particular data used in this preprocessing case. In this scenario, all steps and their exact input and output are known before the chain is translated.

However, if automatic feature selection occurs in the chain, the output is the subset of features that the operator for this step selects. This subset depends on the data and is therefore not known before compilation (translation to relational level). Since the features of the concepts that form the input and output of each step are part of the conceptual design of the chain, the problem arises how to model the output of an automatic feature selection step, and in fact the inputs and outputs of all steps that depend on this step. One option could be to compile the chain up to the feature selection step, and then use the results for further modelling. However, this would mean to interrupt the modelling process in an awkward and counter-intuitive way for an unknown length of time, since the compilation time depends on the size of the data and the number of previous steps.

The chosen solution to this problem is to ease modelling for the user

as much as possible, and let the compiler deal with the problem of how to account for unknown feature subsets. This works as follows. On the conceptual level which the user models, all features of the concepts are always present. In particular, the output of a step involving automatic feature selection is a concept with the same features as the input concept. This ensures that all features can be used in further modelling. However, those features that were not selected in the feature selection step are not present on the relational level in the output of that step and in all dependent steps—they are called *deselected* features. The compiler handles all problems that may arise from this situation.

In fact, there is one type of problem that could not be handled by the compiler, namely the deselection of features that model some important aspects of the conceptual model, like relations between concepts. For example, if a database table has a column with the primary key for this table, it may make sense to have a feature for this column in the concept that represents this table, to be able to use join operators etc. Such features should of course never be deselected. Therefore every operator that involves automatic feature selection uses an input parameter that specifies the superset of features from which it may select a subset. This avoids the deselection of conceptually necessary features.

However, there remain a few situations in which the compiler must deal with deselected features. The problem concerns all operator input parameters of type *Feature*, i.e. *BaseAttributes* and *MultiColumnFeatures*. To handle this, the compiler uses the information about operator parameters that is stored in the M4 model (see MiningMart deliverable 18). This information includes the type of parameter, whether it is input or output, whether a single *Feature* is expected or a list of *Features*, and also whether the parameter is optional or obligatory.

The compiler distinguishes the following situations when the input parameters for a step are loaded, and operates accordingly.

1. The parameter is a *single optional feature*. In this case the compiler checks whether the feature is conceptually present. If not, the user decided not to employ it, which is allowed because it is an optional parameter. No further action is needed. Otherwise, if the feature is conceptually present, the compiler checks whether it happens to be deselected on the relational level. This is true if there are no M4 column objects associated with the feature. If the feature is deselected, the operator is executed as if the feature was not set on the conceptual level—as if the user decided not to employ it. If the feature is not deselected, the operator is executed with the parameter set.

2. The parameter is a *single obligatory feature*. If the feature is not present on the conceptual level, an exception is thrown because the

operator cannot work when the parameter is not set. Otherwise the compiler checks the relational level. If the feature is deselected, an exception is thrown with an informing message to the user ("obligatory parameter XX was deselected in a previous step") and compilation stops. If the feature is not deselected, the operator is executed normally.

3. The parameter is an *optional list of features*. Here the compiler reduces the list of features that the operator is executed on to the subset of features that are not deselected. This subset may be empty, in which case the operator is executed as if the parameter was not set by the user on the conceptual level in the first place. Otherwise the operator is executed on the reduced subset.

4. The parameter is an *obligatory list of features*. In this case the minimal number of elements in the list is also specified in the parameter information tables for this operator. The compiler again reduces the list of features that the operator is executed on to the subset of features that are not deselected. If this subset does not have at least the minimum number of elements that the operator needs, compilation stops with an informing message to the user. Otherwise the operator is executed as if the list of features had been set to the reduced list by the user (on the conceptual level) in the first place.

The two mechanisms of allowing the user to specify a superset of features for an automatic feature selection operator to select from, and of having the compiler deal with deselected features later in the chain, ensure that modelling can be successfully completed on the conceptual level before compilation, and that some parts can be re-modelled if features are deselected that are obligatory in a later step. The user has then the option to remove that feature from the superset of allowed features in the input to the feature selection operator.

Part of the compiler extension for automatic feature selection done for this workpackage was also the provision of an abstract superclass for all operators that use automatic feature selection. This meant that the feature selection operators from deliverables 14.2, 14.4 and 14.5 could be incorporated easily.

2.2 The MiningMart operator

For this workpackage, the operator *FeatureSelectionWithSVM* was incorporated into the MiningMart system. It uses two implementations of the Support Vector Machine algorithm, one that runs in main memory, and one that runs inside the database. The operator has a parameter called *UseDB_SVM*

(see also deliverable 18) with which the user can choose which algorithm to use. The algorithm that runs inside the database is only recommended for very large datasets. Both algorithms are also used by other MiningMart operators that involve the Support Vector Machine.

To be able to use these algorithms from the MiningMart system, wrapper implementations for them were also completed. The main task of the wrappers is to transform the input as it is specified in MiningMart to the implementation-specific formats of the two algorithms. For example, SVMs can deal with boolean values, but only in numeric form (like 1 and 0) and not in nominal form (like *true, false*). So the wrappers read the data from the database according to the information they get from the compiler, convert it like in the boolean example if necessary, and print it to standard input for the SVM in main memory or set up a parameter table in the database for the SVM that runs inside the database. Then they call the algorithm. The result of SVM learning is the separating hyperplane, which is a linear combination of the support vectors. While for the *FeatureSelectionWithSVM* operator only the $\xi\alpha$-estimator is needed, other operators that use the SVM need to be able to use the hyperplane to predict the class of an unseen example. Therefore, it is an important task of the SVM wrappers to transform the function that computes the class of unseen examples (equation 1.6 on page 4) into an SQL function that can then be called inside the database. The form of this SQL function depends on the kernel type of the SVM. The wrappers read the vector $\vec{\alpha}$ and the support vectors from the output of the specific SVM algorithm that was called and implement an SQL function in the database that can be called with an unseen example and returns its class (or its predicted value for a regression SVM). Because the support vectors are needed for each evaluation of the function, and the number of support vectors may be large, they are stored in an extra table inside the database which is used by the function. The wrappers return the name and parameter list of the SQL function to the compiler.

FeatureSelectionWithSVM provides two simple search strategies, the forward selection and the backward selection which are described in deliverable 14.1. However, a simple interface to the SVM algorithms ensures that more complex search strategies can easily be implemented. This interface calls the SVM with a current feature subset and returns the $\xi\alpha$-estimation that the SVM computed on the data that corresponds to the input concept of this operator.

Forward selection starts with an empty feature set. To the current feature set, one feature of the remaining features is added, and the SVM is started on the extended set. Afterwards, the feature is removed and the next of the remaining features is tried. The feature that yields the best $\xi\alpha$-estimation is kept and the search continues on the next level if the best estimation for a set with an added feature is higher than the estimation for the current feature set.

Backward selection starts with the full feature set. It removes a feature, collects the $\xi\alpha$-estimation, then adds the feature again and continues with the next feature. The subset that yields the best estimation is kept and the search continues on the next level if the best estimation for a set with a removed feature is higher than the estimation for the current feature set.

A variant of these search strategies that was tried in the experiments was to continue the search not only when a new feature set gave a better estimation, but also when it returned an equal value for the estimated error. In this way a bigger part of the search space is explored if the estimation values do not distinguish well between different feature subsets.

The parameters for this operator are:

- *TheInputConcept*: Specifies the data on which to learn.
- *TheTargetAttribute*: Specifies the attribute that corresponds to the class label.
- *PositiveTargetValue*: Specifies the value in *TheTargetAttribute* that represents the positive class.
- *TheAttributes*: Specifies the superset of features that the operator may select from.
- *KernelType, C, Epsilon*: Specific parameters for the SVM algorithm.
- *SearchDirection*: Specifies whether to use forward or backward selection.
- *TheKey*: Specifies the primary key of *TheInputConcept*. Only needed for the SVM algorithm inside the database.
- *SampleSize*: Specifies a maximum number of examples to use for learning.
- *UseDB_SVM*: Specifies which of the two algorithms to use.
- *TheOutputConcept*: Specifies which concept will contain the selected features.

2.3 Experiments

Some basic experiments with the operator *FeatureSelectionWithSVM* were carried out to test the implementation and to check that this approach to feature selection can be useful. For these experiments, a learning task that involved insurance data with 23 features was used. The experiments repeated an earlier study at the University of Dortmund, this time using the MiningMart system. (However, the data was already given in preprocessed

format in a database table, so that no special preprocessing was needed in this case.) MiningMart was used to sample a training set and a test set from the data, to apply automatic feature selection and to evaluate the results of learning after feature selection.

No feature selection was used in the original study and the first task was to repeat the experiment with the full set of features, using the parameter settings for the SVM that were used originally. This was achieved; the accuracy in this case was 81%. Using forward selection, only one feature was selected before the search stopped due to the fact that adding a second feature did not improve the estimated generalisation error. However, the accuracy after learning with this one feature was not better than for random classification. When a search variant that continues until the estimated error actually decreases was used, eleven features were selected but the accuracy did not improve much. This shows that the implemented search strategy is probably too simple, because it may happen that two feature sets that differ only in one feature do not lead to a difference in the estimated error. A broader search strategy seems more promising.

With backward selection, four features were removed and the accuracy using the remaining 19 features improved slightly to 82.6%. This shows that the general approach to feature selection presented here can work: the representation of examples could be simplified at no loss, and even a slight improvement, of the generalisation performance. Yet, more experiments on different datasets using improved search strategies are needed to support this result.

Bibliography

[CV95] Corinna Cortes and Vladimir N. Vapnik. Support–vector networks. *Machine Learning Journal*, 20:273–297, 1995.

[Joa01] Thorsten Joachims. *The Maximum-Margin Approach to Learning Text Classifiers: Methods, Theory, and Algorithms.* PhD thesis, Fachbereich Informatik, Universität Dortmund, 2001.

[Vap82] V. Vapnik. *Estimation of Dependencies Based on Empirical Data.* Springer, 1982.

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