Journal of Statistical Software

April 2006, Volume 15, Issue 9. http://www.jstatsoft.org/

Support Vector Machines in R

Alexandros Karatzoglou
Technische Universität Wien

David Meyer
Wirtschaftsuniversität Wien

Kurt Hornik
Wirtschaftsuniversität Wien

Abstract

Being among the most popular and efficient classification and regression methods currently available, implementations of support vector machines exist in almost every popular programming language. Currently four R packages contain SVM related software. The purpose of this paper is to present and compare these implementations.

Keywords: support vector machines, R.

1. Introduction

Support Vector learning is based on simple ideas which originated in statistical learning theory (Vapnik 1998). The simplicity comes from the fact that Support Vector Machines (SVMs) apply a simple linear method to the data but in a high-dimensional feature space non-linearly related to the input space. Moreover, even though we can think of SVMs as a linear algorithm in a high-dimensional space, in practice, it does not involve any computations in that high-dimensional space. This simplicity combined with state of the art performance on many learning problems (classification, regression, and novelty detection) has contributed to the popularity of the SVM. The reminder of the paper is structured as follows. First, we provide a short introduction into Support Vector Machines, followed by an overview of the SVM-related software available in R and other programming languages. Next follows a section on the data sets we will be using. Then, we describe the four available SVM implementations in R. Finally, we present the results of a timing benchmark.

2. Support vector machines

SVMs use an implicit mapping Φ of the input data into a high-dimensional feature space

defined by a kernel function, i.e., a function returning the inner product $\langle \Phi(x), \Phi(x') \rangle$ between the images of two data points x, x' in the feature space. The learning then takes place in the feature space, and the data points only appear inside dot products with other points. This is often referred to as the "kernel trick" (Schölkopf and Smola 2002). More precisely, if a projection $\Phi : X \rightarrow H$ is used, the dot product $\langle \Phi(x), \Phi(x') \rangle$ can be represented by a kernel function k

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle, \tag{1}$$

which is computationally simpler than explicitly projecting x and x' into the feature space H.

One interesting property of support vector machines and other kernel-based systems is that, once a valid kernel function has been selected, one can practically work in spaces of any dimension without any significant additional computational cost, since feature mapping is never effectively performed. In fact, one does not even need to know which features are being used.

Another advantage of SVMs and kernel methods is that one can design and use a kernel for a particular problem that could be applied directly to the data without the need for a feature extraction process. This is particularly important in problems where a lot of structure of the data is lost by the feature extraction process (e.g., text processing).

Training a SVM for classification, regression or novelty detection involves solving a quadratic optimization problem. Using a standard quadratic problem solver for training an SVM would involve solving a big QP problem even for a moderate sized data set, including the computation of an $m \times m$ matrix in memory (m number of training points). This would seriously limit the size of problems an SVM could be applied to. To handle this issue, methods like SMO (Platt 1998), chunking (Osuna, Freund, and Girosi 1997) and simple SVM (Vishwanathan, Smola, and Murty 2003) exist that iteratively compute the solution of the SVM and scale $O(N^k)$ where k is between 1 and 2.5 and have a linear space complexity.

2.1. Classification

In classification, support vector machines separate the different classes of data by a hyper-plane

$$\langle \mathbf{w}, \Phi(x) \rangle + b = 0 \tag{2}$$

corresponding to the decision function

$$f(x) = \text{sign}(\langle \mathbf{w}, \Phi(x) \rangle + b) \tag{3}$$

It can be shown that the optimal, in terms of classification performance, hyper-plane (Vapnik 1998) is the one with the maximal margin of separation between the two classes. It can be constructed by solving a constrained quadratic optimization problem whose solution $\mathbf{w}$ has an expansion $\mathbf{w} = \sum_i \alpha_i \Phi(x_i)$ in terms of a subset of training patterns that lie on the margin. These training patterns, called *support vectors*, carry all relevant information about the classification problem. Omitting the details of the calculation, there is just one crucial property of the algorithm that we need to emphasize: both the quadratic programming problem and the final decision function depend only on dot products between patterns. This allows the use of the "kernel trick" and the generalization of this linear algorithm to the nonlinear case.

In the case of the L_2-norm soft margin classification the primal optimization problem takes the form:

$$\begin{aligned} \text{minimize} \quad & t(\mathbf{w}, \xi) = \frac{1}{2}\|\mathbf{w}\|^2 + \frac{C}{m}\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & y_i(\langle\Phi(x_i), \mathbf{w}\rangle + b) \geq 1 - \xi_i \qquad (i = 1, \ldots, m) \\ & \xi_i \geq 0 \qquad (i = 1, \ldots, m) \end{aligned} \tag{4}$$

where m is the number of training patterns, and $y_i = \pm 1$. As in most kernel methods, the SVM solution $\mathbf{w}$ can be shown to have an expansion

$$\mathbf{w} = \sum_{i=1}^{m}\alpha_i y_i \Phi(x_i) \tag{5}$$

where non-zero coefficients (support vectors) occur when a point (x_i, y_i) meets the constraint. The coefficients α_i are found by solving the following (dual) quadratic programming problem:

$$\begin{aligned} \text{maximize} \quad & W(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j k(x_i, x_j) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq \frac{C}{m} \qquad (i = 1, \ldots, m) \\ & \sum_{i=1}^{m}\alpha_i y_i = 0. \end{aligned} \tag{6}$$

This is a typical quadratic problem of the form:

$$\begin{aligned} \text{minimize} \quad & c^\top x + \tfrac{1}{2}x^\top H x \\ \text{subject to} \quad & b \leq Ax \leq b + r \\ & l \leq x \leq u \end{aligned} \tag{7}$$

where $H \in \mathbb{R}^{m\times m}$ with entries $H_{ij} = y_i y_j k(x_i, x_j)$, $c = (1, \ldots, 1) \in \mathbb{R}^m$, $u = (C, \ldots, C) \in \mathbb{R}^m$, $l = (0, \ldots, 0) \in \mathbb{R}^m$, $A = (y_1, \ldots, y_m) \in \mathbb{R}^m$, $b = 0$, $r = 0$. The problem can easily be solved in a standard QP solver such as `quadprog()` in package **quadprog** (Weingessel 2004) or `ipop()` in package **kernlab** (Karatzoglou, Smola, Hornik, and Zeileis 2005), both available in R (R Development Core Team 2005). Techniques taking advantage of the special structure of the SVM QP problem like SMO and chunking (Osuna *et al.* 1997) though offer much better performance in terms of speed, scalability and memory usage.

The cost parameter C of the SVM formulation in Equation 7 controls the penalty paid by the SVM for missclassifying a training point and thus the complexity of the prediction function. A high cost value C will force the SVM to create a complex enough prediction function to missclassify as few training points as possible, while a lower cost parameter will lead to a simpler prediction function. Therefore, this type of SVM is usually called C-SVM.

Another formulation of the classification with a more intuitive hyperparameter than C is the ν-SVM (Schölkopf, Smola, Williamson, and Bartlett 2000). The ν parameter has the interesting property of being an upper bound on the training error and a lower bound on

the fraction of support vectors found in the data set, thus controlling the complexity of the classification function build by the SVM (see Appendix for details).

For multi-class classification, mostly voting schemes such as one-against-one and one-against-all are used. In the one-against-all method k binary SVM classifiers are trained, where k is the number of classes, each trained to separate one class from the rest. The classifiers are then combined by comparing their decision values on a test data instance and labeling it according to the classifier with the highest decision value.

In the one-against-one classification method (also called pairwise classification; see Knerr, Personnaz, and Dreyfus 1990; Kreßel 1999), $\binom{k}{2}$ classifiers are constructed where each one is trained on data from two classes. Prediction is done by voting where each classifier gives a prediction and the class which is most frequently predicted wins ("Max Wins"). This method has been shown to produce robust results when used with SVMs (Hsu and Lin 2002a). Although this suggests a higher number of support vector machines to train the overall CPU time used is less compared to the one-against-all method since the problems are smaller and the SVM optimization problem scales super-linearly.

Furthermore, SVMs can also produce class probabilities as output instead of class labels. This is can done by an improved implementation (Lin, Lin, and Weng 2001) of Platt's a posteriori probabilities (Platt 2000) where a sigmoid function

$$P(y = 1 \mid f) = \frac{1}{1 + e^{Af+B}} \tag{8}$$

is fitted to the decision values f of the binary SVM classifiers, A and B being estimated by minimizing the negative log-likelihood function. This is equivalent to fitting a logistic regression model to the estimated decision values. To extend the class probabilities to the multi-class case, all binary classifiers class probability output can be combined as proposed in Wu, Lin, and Weng (2003).

In addition to these heuristics for extending a binary SVM to the multi-class problem, there have been reformulations of the support vector quadratic problem that deal with more than two classes. One of the many approaches for native support vector multi-class classification is the one proposed in Crammer and Singer (2000), which we will refer to as 'spoc-svc'. This algorithm works by solving a single optimization problem including the data from all classes. The primal formulation is:

$$\begin{aligned}
\text{minimize} \quad & t(\{\mathbf{w_n}\}, \xi) = \frac{1}{2} \sum_{n=1}^{k} \|\mathbf{w_n}\|^2 + \frac{C}{m} \sum_{i=1}^{m} \xi_i \\
\text{subject to} \quad & \langle \Phi(x_i), \mathbf{w}_{y_i} \rangle - \langle \Phi(x_i), \mathbf{w}_n \rangle \geq b_i^n - \xi_i \qquad (i = 1, \ldots, m) \qquad (9) \\
\text{where} \quad & b_i^n = 1 - \delta_{y_i,n} \qquad (10)
\end{aligned}$$

where the decision function is

$$\text{argmax}_{n=1,\ldots,k} \langle \Phi(x_i), \mathbf{w}_n \rangle \tag{11}$$

Details on performance and benchmarks on various approaches for multi-class classification can be found in Hsu and Lin (2002b).

2.2. Novelty detection

SVMs have also been extended to deal with the problem of *novelty detection* (or one-class classification; see Schölkopf, Platt, Shawe-Taylor, Smola, and Williamson 1999; Tax and Duin 1999), where essentially an SVM detects outliers in a data set. SVM novelty detection works by creating a spherical decision boundary around a set of data points by a set of support vectors describing the sphere's boundary. The primal optimization problem for support vector novelty detection is the following:

$$\begin{aligned} \text{minimize} \quad & t(\mathbf{w}, \xi, \rho) = \frac{1}{2}\|\mathbf{w}\|^2 - \rho + \frac{1}{m\nu}\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & \langle \Phi(x_i), \mathbf{w}\rangle + b \geq \rho - \xi_i \qquad (i = 1, \ldots, m) \\ & \xi_i \geq 0 \qquad (i = 1, \ldots, m). \end{aligned} \tag{12}$$

The ν parameter is used to control the volume of the sphere and consequently the number of outliers found. The value of ν sets an upper bound on the fraction of outliers found in the data.

2.3. Regression

By using a different loss function called the ϵ-insensitive loss function $\|y - f(x)\|_\epsilon = \max\{0, \|y - f(x)\| - \epsilon\}$, SVMs can also perform regression. This loss function ignores errors that are smaller than a certain threshold $\epsilon > 0$ thus creating a tube around the true output. The primal becomes:

$$\begin{aligned} \text{minimize} \quad & t(\mathbf{w}, \xi) = \frac{1}{2}\|w\|^2 + \frac{C}{m}\sum_{i=1}^{m}(\xi_i + \xi_i^*) \\ \text{subject to} \quad & (\langle \Phi(x_i), \mathbf{w}\rangle + b) - y_i \leq \epsilon - \xi_i & (13) \\ & y_i - (\langle \Phi(x_i), \mathbf{w}\rangle + b) \leq \epsilon - \xi_i^* & (14) \\ & \xi_i^* \geq 0 \qquad (i = 1, \ldots, m) \end{aligned}$$

We can estimate the accuracy of SVM regression by computing the scale parameter of a Laplacian distribution on the residuals $\zeta = y - f(x)$, where $f(x)$ is the estimated decision function (Lin and Weng 2004).

The dual problems of the various classification, regression and novelty detection SVM formulations can be found in the Appendix.

2.4. Kernel functions

As seen before, the kernel functions return the inner product between two points in a suitable feature space, thus defining a notion of similarity, with little computational cost even in very high-dimensional spaces. Kernels commonly used with kernel methods and SVMs in particular include the following:

- the linear kernel implementing the simplest of all kernel functions

$$k(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle \tag{15}$$

- the Gaussian Radial Basis Function (RBF) kernel

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\sigma \|\mathbf{x} - \mathbf{x}'\|^2) \tag{16}$$

- the polynomial kernel

$$k(\mathbf{x}, \mathbf{x}') = \big(\text{scale} \cdot \langle \mathbf{x}, \mathbf{x}' \rangle + \text{offset}\big)^{\text{degree}} \tag{17}$$

- the hyperbolic tangent kernel

$$k(\mathbf{x}, \mathbf{x}') = \tanh\big(\text{scale} \cdot \langle \mathbf{x}, \mathbf{x}' \rangle + \text{offset}\big) \tag{18}$$

- the Bessel function of the first kind kernel

$$k(\mathbf{x}, \mathbf{x}') = \frac{\text{Bessel}^n_{(\nu+1)}(\sigma \|\mathbf{x} - \mathbf{x}'\|)}{(\|x - x'\|)^{-n(\nu+1)}} \tag{19}$$

- the Laplace Radial Basis Function (RBF) kenrel

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\sigma \|\mathbf{x} - \mathbf{x}'\|) \tag{20}$$

- the ANOVA radial basis kernel

$$k(\mathbf{x}, \mathbf{x}') = \left(\sum_{k=1}^{n} \exp(-\sigma (x^k - x'^k)^2) \right)^d \tag{21}$$

- the linear splines kernel in one dimension

$$k(x, x') = 1 + xx' \min(x, x') - \frac{x + x'}{2} (\min(x, x')^2 + \frac{(\min(x, x')^3)}{3} \tag{22}$$

  and for the multidimensional case $k(\mathbf{x}, \mathbf{x}') = \prod_{k=1}^{n} k(x^k, x'^k)$.

The Gaussian and Laplace RBF and Bessel kernels are general-purpose kernels used when there is no prior knowledge about the data. The linear kernel is useful when dealing with large sparse data vectors as is usually the case in text categorization. The polynomial kernel is popular in image processing and the sigmoid kernel is mainly used as a proxy for neural networks. The splines and ANOVA RBF kernels typically perform well in regression problems.

2.5. Software

Support vector machines are currently used in a wide range of fields, from bioinformatics to astrophysics. Thus, the existence of many SVM software packages comes as little surprise. Most existing software is written in C or C++, such as the award winning **libsvm** (Chang and Lin 2001), which provides a robust and fast SVM implementation and produces state of the

art results on most classification and regression problems (Meyer, Leisch, and Hornik 2003), **SVMlight** (Joachims 1999), **SVMTorch** (Collobert, Bengio, and Mariéthoz 2002), **Royal Holloway Support Vector Machines**, (Gammerman, Bozanic, Schölkopf, Vovk, Vapnik, Bottou, Smola, Watkins, LeCun, Saunders, Stitson, and Weston 2001), **mySVM** (Rüping 2004), and **M-SVM** (Guermeur 2004). Many packages provide interfaces to MATLAB (The MathWorks 2005) (such as **libsvm**), and there are some native MATLAB toolboxes as well such as the **SVM and Kernel Methods Matlab Toolbox** (Canu, Grandvalet, and Rakotomamonjy 2003) or the **MATLAB Support Vector Machine Toolbox** (Gunn 1998) and the **SVM toolbox for Matlab** (Schwaighofer 2005)

2.6. R software overview

The first implementation of SVM in R (R Development Core Team 2005) was introduced in the **e1071** (Dimitriadou, Hornik, Leisch, Meyer, and Weingessel 2005) package. The `svm()` function in **e1071** provides a rigid interface to **libsvm** along with visualization and parameter tuning methods.

Package **kernlab** features a variety of kernel-based methods and includes a SVM method based on the optimizers used in **libsvm** and **bsvm** (Hsu and Lin 2002c). It aims to provide a flexible and extensible SVM implementation.

Package **klaR** (Roever, Raabe, Luebke, and Ligges 2005) includes an interface to **SVMlight**, a popular SVM implementation that additionally offers classification tools such as Regularized Discriminant Analysis.

Finally, package **svmpath** (Hastie 2004) provides an algorithm that fits the entire path of the SVM solution (i.e., for any value of the cost parameter).

In the remainder of the paper we will extensively review and compare these four SVM implementations.

3. Data

Throughout the paper, we will use the following data sets accessible through R (see Table 1), most of them originating from the UCI machine learning database (Blake and Merz 1998):

iris This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*. The data set is provided by base R.

spam A data set collected at Hewlett-Packard Labs which classifies 4601 e-mails as spam or non-spam. In addition to this class label there are 57 variables indicating the frequency of certain words and characters in the e-mail. The data set is provided by the **kernlab** package.

musk This dataset in package **kernlab** describes a set of 476 molecules of which 207 are judged by human experts to be musks and the remaining 269 molecules are judged to be non-musks. The data has 167 variables which describe the geometry of the molecules.

promotergene Promoters have a region where a protein (RNA polymerase) must make contact and the helical DNA sequence must have a valid conformation so that the two pieces of the contact region spatially align. The dataset in package **kernlab** contains DNA sequences of promoters and non-promoters in a data frame with 106 observations and 58 variables. The DNA bases are coded as follows: 'a' adenine, 'c' cytosine, 'g' guanine, and 't' thymine.

Vowel Speaker independent recognition of the eleven steady state vowels of British English using a specified training set of LPC derived log area ratios. The vowels are indexed by integers 0 to 10. This dataset in package **mlbench** (Leisch and Dimitriadou 2001) has 990 observations on 10 independent variables.

DNA in package **mlbench** consists of 3,186 data points (splice junctions). The data points are described by 180 indicator binary variables and the problem is to recognize the 3 classes ('ei', 'ie', neither), i.e., the boundaries between exons (the parts of the DNA sequence retained after splicing) and introns (the parts of the DNA sequence that are spliced out).

BreastCancer in package **mlbench** is a data frame with 699 observations on 11 variables, one being a character variable, 9 being ordered or nominal, and 1 target class. The objective is to identify each of a number of benign or malignant classes.

BostonHousing Housing data in package **mlbench** for 506 census tracts of Boston from the 1970 census. There are 506 observations on 14 variables.

B3 German Bussiness Cycles from 1955 to 1994 in package **klaR**. A data frame with 157 observations on the following 14 variables.

Dataset	#Examples	#Attributes				Class Distribution (%)
		b	c	m	cl	
iris	150			5	3	33.3/33.3/33.3
spam	4601			57	2	39.40/60.59
musk	476			166	2	42.99 / 57.00
promotergene	106		57		2	50.00 / 50.00
Vowel	990		1	9	10	10.0/10.0/...
DNA	3186	180			3	24.07/24.07/51.91
BreastCancer	699		9		2	34.48 / 65.52
BostonHousing	506	1		12		(regression)
B3	506			13	4	37.57/15.28/29.93/17.19

Table 1: The data sets used throughout the paper. Legend: b=binary, c=categorical, m=metric, cl = number of classes.

4. `ksvm` in kernlab

Package **kernlab** (Karatzoglou, Smola, Hornik, and Zeileis 2004) aims to provide the R user with basic kernel functionality (e.g., like computing a kernel matrix using a particular kernel),

along with some utility functions commonly used in kernel-based methods like a quadratic programming solver, and modern kernel-based algorithms based on the functionality that the package provides. It also takes advantage of the inherent modularity of kernel-based methods, aiming to allow the user to switch between kernels on an existing algorithm and even create and use own kernel functions for the various kernel methods provided in the package.

kernlab uses R's new object model described in "Programming with Data" (Chambers 1998) which is known as the S4 class system and is implemented in package **methods**. In contrast to the older S3 model for objects in R, classes, slots, and methods relationships must be declared explicitly when using the S4 system. The number and types of slots in an instance of a class have to be established at the time the class is defined. The objects from the class are validated against this definition and have to comply to it at any time. S4 also requires formal declarations of methods, unlike the informal system of using function names to identify a certain method in S3. Package **kernlab** is available from CRAN (http://CRAN.R-project.org/) under the GPL license.

The `ksvm()` function, **kernlab**'s implementation of SVMs, provides a standard formula interface along with a matrix interface. `ksvm()` is mostly programmed in R but uses, through the `.Call` interface, the optimizers found in **bsvm** and **libsvm** (Chang and Lin 2001) which provide a very efficient C++ version of the Sequential Minimization Optimization (SMO). The SMO algorithm solves the SVM quadratic problem (QP) without using any numerical QP optimization steps. Instead, it chooses to solve the smallest possible optimization problem involving two elements of α_i because the must obey one linear equality constraint. At every step, SMO chooses two α_i to jointly optimize and finds the optimal values for these α_i analytically, thus avoiding numerical QP optimization, and updates the SVM to reflect the new optimal values.

The SVM implementations available in `ksvm()` include the C-SVM classification algorithm along with the ν-SVM classification. Also included is a bound constraint version of C classification (C-BSVM) which solves a slightly different QP problem (Mangasarian and Musicant 1999, including the offset β in the objective function) using a modified version of the **TRON** (Lin and More 1999) optimization software. For regression, `ksvm()` includes the ϵ-SVM regression algorithm along with the ν-SVM regression formulation. In addition, a bound constraint version (ϵ-BSVM) is provided, and novelty detection (one-class classification) is supported.

For classification problems which include more then two classes (multi-class case) two options are available: a one-against-one (pairwise) classification method or the native multi-class formulation of the SVM (spoc-svc) described in Section 2. The optimization problem of the native multi-class SVM implementation is solved by a decomposition method proposed in Hsu and Lin (2002c) where optimal working sets are found (that is, sets of α_i values which have a high probability of being non-zero). The QP sub-problems are then solved by a modified version of the **TRON** optimization software.

The `ksvm()` implementation can also compute class-probability output by using Platt's probability methods (Equation 8) along with the multi-class extension of the method in Wu *et al.* (2003). The prediction method can also return the raw decision values of the support vector model:

```
> library("kernlab")
> data("iris")
> irismodel <- ksvm(Species ~ ., data = iris,
```

```
+     type = "C-bsvc", kernel = "rbfdot",
+     kpar = list(sigma = 0.1), C = 10,
+     prob.model = TRUE)
> irismodel

Support Vector Machine object of class "ksvm"

SV type: C-bsvc  (classification)
 parameter : cost C = 10

Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.1

Number of Support Vectors : 32
Training error : 0.02
Probability model included.

> predict(irismodel, iris[c(3, 10, 56, 68,
+     107, 120), -5], type = "probabilities")

          setosa  versicolor   virginica
[1,] 0.986432820 0.007359407 0.006207773
[2,] 0.983323813 0.010118992 0.006557195
[3,] 0.004852528 0.967555126 0.027592346
[4,] 0.009546823 0.988496724 0.001956452
[5,] 0.012767340 0.069496029 0.917736631
[6,] 0.011548176 0.150035384 0.838416441

> predict(irismodel, iris[c(3, 10, 56, 68,
+     107, 120), -5], type = "decision")

          [,1]       [,2]       [,3]
[1,] -1.460398 -1.1910251 -3.8868836
[2,] -1.357355 -1.1749491 -4.2107843
[3,]  1.647272  0.7655001 -1.3205306
[4,]  1.412721  0.4736201 -2.7521640
[5,]  1.844763  1.0000000  1.0000019
[6,]  1.848985  1.0069010  0.6742889
```

ksvm allows for the use of any valid user defined kernel function by just defining a function which takes two vector arguments and returns its Hilbert Space dot product in scalar form.

```
> k <- function(x, y) {
+     (sum(x * y) + 1) * exp(0.001 * sum((x -
+         y)^2))
+ }
```

```
> class(k) <- "kernel"
> data("promotergene")
> gene <- ksvm(Class ~ ., data = promotergene,
+     kernel = k, C = 10, cross = 5)
> gene

Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 10


Number of Support Vectors : 66
Training error : 0
Cross validation error : 0.141558
```

The implementation also includes the following computationally efficiently implemented kernels: Gaussian RBF, polynomial, linear, sigmoid, Laplace, Bessel RBF, spline, and ANOVA RBF.

N-fold cross-validation of an SVM model is also supported by `ksvm`, and the training error is reported by default.

The problem of model selection is partially addressed by an empirical observation for the popular Gaussian RBF kernel (Caputo, Sim, Furesjo, and Smola 2002), where the optimal values of the width hyper-parameter σ are shown to lie in between the 0.1 and 0.9 quantile of the $\|x - x'\|^2$ statistics. The `sigest()` function uses a sample of the training set to estimate the quantiles and returns a vector containing the values of the quantiles. Pretty much any value within this interval leads to good performance.

The object returned by the `ksvm()` function is an S4 object of class ksvm with slots containing the coefficients of the model (support vectors), the parameters used (C, ν, etc.), test and cross-validation error, the kernel function, information on the problem type, the data scaling parameters, etc. There are accessor functions for the information contained in the slots of the ksvm object.

The decision values of binary classification problems can also be visualized via a contour plot with the `plot()` method for the ksvm objects. This function is mainly for simple problems. An example is shown in Figure 1.

```
> x <- rbind(matrix(rnorm(120), , 2), matrix(rnorm(120,
+     mean = 3), , 2))
> y <- matrix(c(rep(1, 60), rep(-1, 60)))
> svp <- ksvm(x, y, type = "C-svc", kernel = "rbfdot",
+     kpar = list(sigma = 2))
> plot(svp)
```

5. svm in e1071

Package **e1071** provides an interface to **libsvm** (Chang and Lin 2001, current version: 2.8),

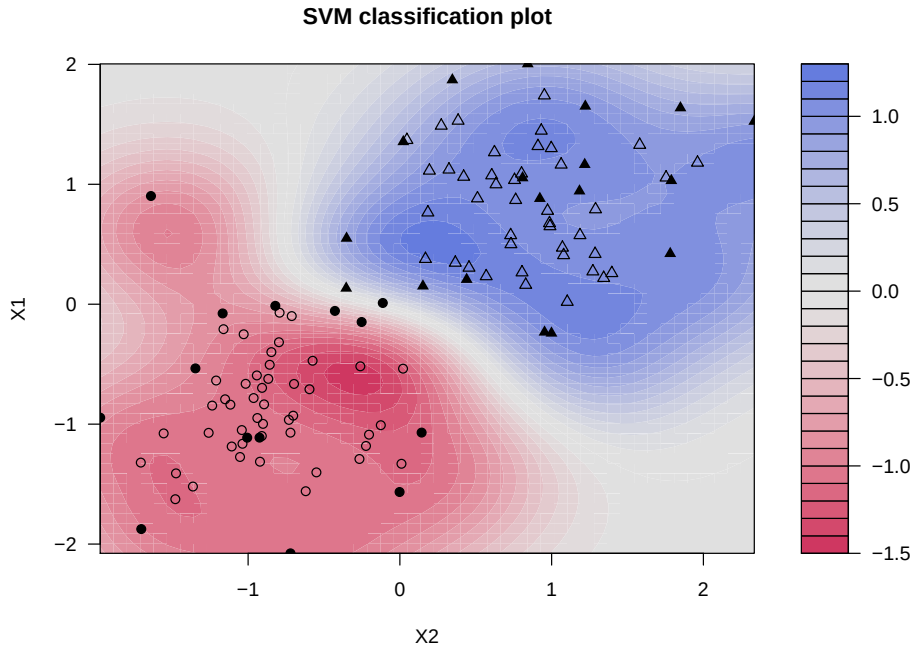

Figure 1: A contour plot of the fitted decision values for a simple binary classification problem.

complemented by visualization and tuning functions. **libsvm** is a fast and easy-to-use implementation of the most popular SVM formulations (C and ν classification, ϵ and ν regression, and novelty detection). It includes the most common kernels (linear, polynomial, RBF, and sigmoid), only extensible by changing the C++ source code of **libsvm**. Multi-class classification is provided using the one-against-one voting scheme. Other features include the computation of decision and probability values for predictions (for both classification and regression), shrinking heuristics during the fitting process, class weighting in the classification mode, handling of sparse data, and the computation of the training error using cross-validation. **libsvm** is distributed under a very permissive, BSD-like licence.

The R implementation is based on the S3 class mechanisms. It basically provides a training function with standard and formula interfaces, and a `predict()` method. In addition, a `plot()` method visualizing data, support vectors, and decision boundaries if provided. Hyper-parameter tuning is done using the `tune()` framework in **e1071** performing a grid search over specified parameter ranges.

The sample session starts with a C classification task on the iris data, using the radial basis function kernel with fixed hyper-parameters C and γ:

```
> library("e1071")
> model <- svm(Species ~ ., data = iris_train,
+     method = "C-classification", kernel = "radial",
+     cost = 10, gamma = 0.1)
> summary(model)

Call:
```

```
 svm(formula = Species ~ ., data = iris_train, method = "C-classification",
+    kernel = "radial", cost = 10, gamma = 0.1)

Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  10
      gamma:  0.1

Number of Support Vectors:  27

 ( 12 12 3 )


Number of Classes:  3

Levels:
 setosa versicolor virginica
```

We can visualize a 2-dimensional projection of the data with highlighting classes and support vectors (see Figure 2):

```
> plot(model, iris_train, Petal.Width ~
+     Petal.Length, slice = list(Sepal.Width = 3,
+     Sepal.Length = 4))
```

Predictions from the model, as well as decision values from the binary classifiers, are obtained using the `predict()` method:

```
> (pred <- predict(model, head(iris), decision.values = TRUE))

[1] setosa setosa setosa setosa setosa setosa
Levels: setosa versicolor virginica

> attr(pred, "decision.values")

  virginica/versicolor virginica/setosa
1            -3.833133       -1.156482
2            -3.751235       -1.121963
3            -3.540173       -1.177779
4            -3.491439       -1.153052
5            -3.657509       -1.172285
6            -3.702492       -1.069637
  versicolor/setosa
1         -1.393419
2         -1.279886
3         -1.456532
```

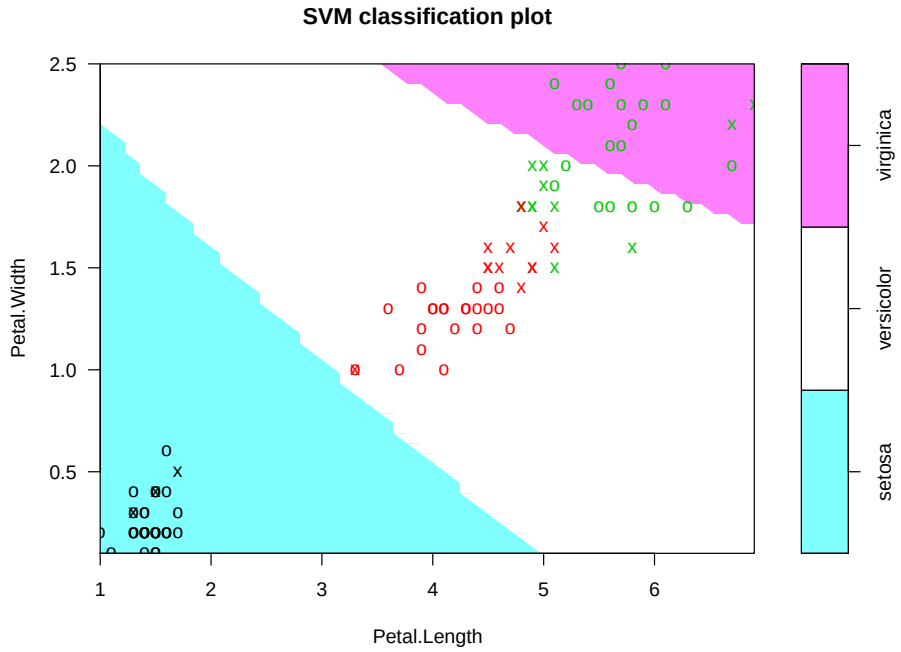

Figure 2: SVM plot visualizing the iris data. Support vectors are shown as 'X', true classes are highlighted through symbol color, predicted class regions are visualized using colored background.

```
4          -1.364424
5          -1.423417
6          -1.158232
```

Probability values can be obtained in a similar way.

In the next example, we again train a classification model on the spam data. This time, however, we will tune the hyper-parameters on a subsample using the `tune` framework of **e1071**:

```
> tobj <- tune.svm(type ~ ., data = spam_train[1:300,
+     ], gamma = 10^(-6:-3), cost = 10^(1:2))
> summary(tobj)

Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:
 gamma cost
 0.001   10
```

```
- best performance: 0.1233333

- Detailed performance results:
  gamma cost     error
1 1e-06   10 0.4133333
2 1e-05   10 0.4133333
3 1e-04   10 0.1900000
4 1e-03   10 0.1233333
5 1e-06  100 0.4133333
6 1e-05  100 0.1933333
7 1e-04  100 0.1233333
8 1e-03  100 0.1266667
```

`tune.svm()` is a convenience wrapper to the `tune()` function that carries out a grid search over the specified parameters. The `summary()` method on the returned object indicates the misclassification rate for each parameter combination and the best model. By default, the error measure is computed using a 10-fold cross validation on the given data, but `tune()` offers several alternatives (e.g., separate training and test sets, leave-one-out-error, etc.). In this example, the best model in the parameter range is obtained using $C = 10$ and $\gamma = 0.001$, yielding a misclassification error of 12.33%. A graphical overview on the tuning results (that is, the error landscape) can be obtained by drawing a contour plot (see Figure 3):

```
> plot(tobj, transform.x = log10, xlab = expression(log[10](gamma)),
+     ylab = "C")
```

Using the best parameters, we now train our final model. We estimate the accuracy in two ways: by 10-fold cross validation on the training data, and by computing the predictive accuracy on the test set:

```
> bestGamma <- tobj$best.parameters[[1]]
> bestC <- tobj$best.parameters[[2]]
> model <- svm(type ~ ., data = spam_train,
+     cost = bestC, gamma = bestGamma, cross = 10)
> summary(model)

Call:
 svm(formula = type ~ ., data = spam_train, cost = bestC, gamma = bestGamma,
+    cross = 10)

Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  10
      gamma:  0.001

Number of Support Vectors:  313
```

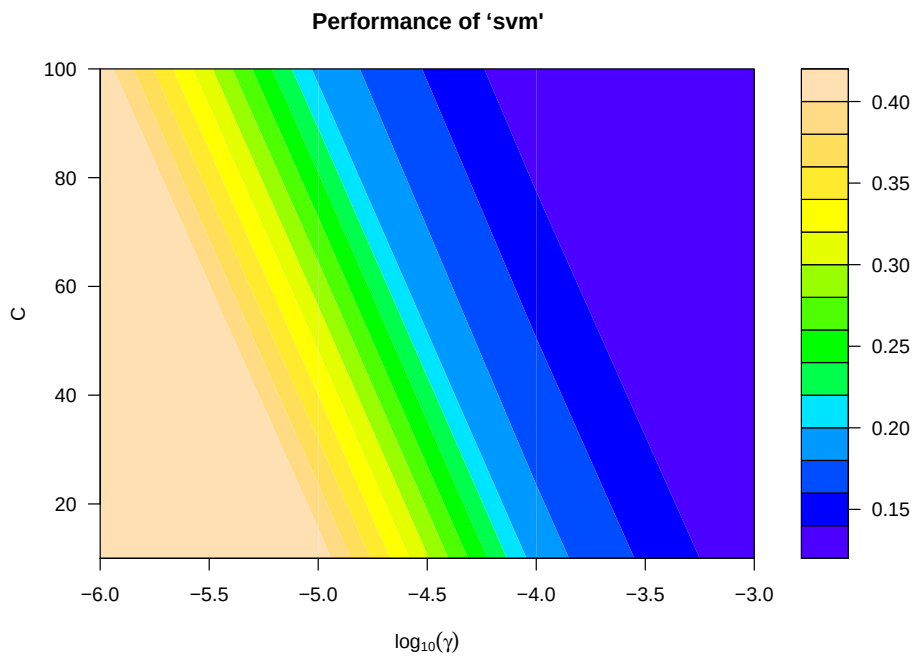

Figure 3: Contour plot of the error landscape resulting from a grid search on a hyper-parameter range.

```
 ( 162 151 )


Number of Classes:  2

Levels:
 nonspam spam

10-fold cross-validation on training data:

Total Accuracy: 91.6
Single Accuracies:
 94 91 92 90 91 91 92 90 92 93

> pred <- predict(model, spam_test)
> (acc <- table(pred, spam_test$type))

pred      nonspam spam
  nonspam    2075  196
  spam        115 1215

> classAgreement(acc)
```

```
$diag
[1] 0.9136351

$kappa
[1] 0.8169207

$rand
[1] 0.8421442

$crand
[1] 0.6832857
```

6. svmlight in klaR

Package **klaR** (Roever *et al.* 2005) includes utility functions for classification and visualization, and provides the `svmlight()` function which is a fairly simple interface to the **SVMlight** package. The `svmlight()` function in **klaR** is written in the S3 object system and provides a formula interface along with standard matrix, data frame, and formula interfaces. The **SVMlight** package is available only for non-commercial use, and the installation of the package involves placing the **SVMlight** binaries in the path of the operating system. The interface works by using temporary text files where the data and parameters are stored before being passed to the **SVMlight** binaries.

SVMlight utilizes a special active set method (Joachims 1999) for solving the SVM QP problem where q variables (the active set) are selected per iteration for optimization. The selection of the active set is done in a way which maximizes the progress towards the minimum of the objective function. At each iteration a QP subproblem is solved using only the active set until the final solution is reached.

The **klaR** interface function `svmlight()` supports the C-SVM formulation for classification and the ϵ-SVM formulation for regression. **SVMlight** uses the one-against-all method for multi-class classification where k classifiers are trained. Compared to the one-against-one method, this requires usually less binary classifiers to be built but the problems each classifier has to deal with are bigger.

The **SVMlight** implementation provides the Gaussian, polynomial, linear, and sigmoid kernels. The `svmlight()` interface employs a character string argument to pass parameters to the **SVMlight** binaries. This allows direct access to the feature-rich **SVMlight** and allows, e.g., control of the SVM parameters (cost, ϵ), the choice of the kernel function and the hyper-parameters, the computation of the leave-one-out error, and the control of the verbosity level.

The S3 object returned by the `svmlight()` function in **klaR** is of class svmlight and is a list containing the model coefficients along with information on the learning task, like the type of problem, and the parameters and arguments passed to the function. The svmlight object has no `print()` or `summary()` methods. The `predict()` method returns the class labels in case of classification along with a class membership value (class probabilities) or the decision values of the classifier.

```
> library("klaR")
```

```
> data("B3")
> Bmod <- svmlight(PHASEN ~ ., data = B3,
+     svm.options = "-c 10 -t 2 -g 0.1 -v 0")
> predict(Bmod, B3[c(4, 9, 30, 60, 80, 120),
+     -1])

$class
[1] 3 3 4 3 4 1
Levels: 1 2 3 4

$posterior
              1          2          3          4
[1,] 0.09633177 0.09627103 0.71112031 0.09627689
[2,] 0.09628235 0.09632512 0.71119794 0.09619460
[3,] 0.09631525 0.09624314 0.09624798 0.71119362
[4,] 0.09632530 0.09629393 0.71115614 0.09622463
[5,] 0.09628295 0.09628679 0.09625447 0.71117579
[6,] 0.71123818 0.09627858 0.09620351 0.09627973
```

7. svmpath

The performance of the SVM is highly dependent on the value of the regularization parameter C, but apart from grid search, which is often computationally expensive, there is little else a user can do to find a value yielding good performance. Although the ν-SVM algorithm partially addresses this problem by reformulating the SVM problem and introducing the ν parameter, finding a correct value for ν relies on at least some knowledge of the expected result (test error, number of support vectors, etc.).

Package **svmpath** (Hastie 2004) contains a function `svmpath()` implementing an algorithm which solves the C-SVM classification problem for all the values of the regularization cost parameter $\lambda = 1/C$ (Hastie, Rosset, Tibshirani, and Zhu 2004). The algorithm exploits the fact that the loss function is piecewise linear and thus the parameters (coefficients) $\alpha(\lambda)$ of the SVM model are also piecewise linear as functions of the regularization parameter λ. The algorithm solves the SVM problem for all values of the regularization parameter with essentially a small multiple ($\approx$ 3) of the computational cost of fitting a single model.

The algorithm works by starting with a high value of λ (high regularization) and tracking the changes to the model coefficients α as the value of λ is decreased. When λ decreases, $||\alpha||$ and hence the width of the margin decrease, and points move from being inside to outside the margin. Their corresponding coefficients α_i change from $\alpha_i = 1$ when they are inside the margin to $\alpha_i = 0$ when outside. The trajectories of the α_i are piecewise linear in λ and by tracking the break points all values in between can be found by simple linear interpolation.

The `svmpath()` implementation in R currently supports only binary C classification. The function must be used through a S3 matrix interface where the y label must be $+1$ or -1. Similarly to `ksvm()`, `svmpath()` allows the use of any user defined kernel function, but in its current implementation requires the direct computation of full kernel matrices, thus limiting the size of problems `svmpath()` can be used on since the full $m \times m$ kernel matrix has to be

computed in memory. The implementation comes with the Gaussian RBF and polynomial kernel as built-in kernel functions and also provides the user with the option of using a precomputed kernel matrix K.

The function call returns an object of class svmpath which is a list containing the model coefficients (α_i) for the break points along with the offsets and the value of the regularization parameter $\lambda = 1/C$ at the points. Also included is information on the kernel function and its hyper-parameter. The `predict()` method for svmpath objects returns the decision values, or the binary labels $(+1, -1)$ for a specified value of the $\lambda = 1/C$ regularization parameter. The `predict()` method can also return the model coefficients α for any value of the λ parameter.

```
> library("svmpath")
> data("svmpath")
> attach(balanced.overlap)
> svmpm <- svmpath(x, y, kernel.function = radial.kernel,
+     param.kernel = 0.1)
> predict(svmpm, x, lambda = 0.1)
```

```
            [,1]
 [1,] -0.8399810
 [2,] -1.0000000
 [3,] -1.0000000
 [4,] -1.0000000
 [5,]  0.1882592
 [6,] -2.2363430
 [7,]  1.0000000
 [8,] -0.2977907
 [9,]  0.3468992
[10,]  0.1933259
[11,]  1.0580215
[12,]  0.9309218
```

```
> predict(svmpm, lambda = 0.2, type = "alpha")
```

```
$alpha0
[1] -0.3809953

$alpha
 [1] 1.000000e+00 1.000000e+00 9.253461e-01
 [4] 1.000000e+00 1.000000e+00 1.110223e-16
 [7] 1.000000e+00 1.000000e+00 1.000000e+00
[10] 1.000000e+00 1.110223e-16 9.253461e-01

$lambda
[1] 0.2
```

8. Benchmarking

In the following we compare the four SVM implementations in terms of training time. In this comparison we only focus on the actual training time of the SVM excluding the time needed for estimating the training error or the cross-validation error. In implementations which scale the data (`ksvm()`, `svm()`) we include the time needed to scale the data. We include both binary and multi-class classification problems as well as a few regression problems. The training is done using a Gaussian kernel where the hyper-parameter was estimated using the `sigest()` function in **kernlab**, which estimates the 0.1 and 0.9 quantiles of $\|x - x'\|^2$. The data was scaled to unit variance and the features for estimating the training error and the fitted values were turned off and the whole data set was used for the training. The mean value of 10 runs is given in Table 2; we do not report the variance since it was practically 0 in all runs. The runs were done with version 0.6-2 of **kernlab**, version 1.5-11 of **e1071**, version 0.9 of **svmpath**, and version 0.4-1 of **klaR**.

Table 2 contains the training times for the SVM implementations on the various datasets. `ksvm()` and `svm()` seem to perform on a similar level in terms of training time with the `svmlight()` function being significantly slower. When comparing `svmpath()` with the other implementations, one has to keep in mind that it practically estimates the SVM model coefficients for the whole range of the cost parameter C. The `svmlight()` function seems to suffer from the fact that the interface is based on reading and writing temporary text files as well as from the optimization method (chunking) used from the **SVMlight** software which in these experiments does not seem to perform as well as the SMO implementation in **libsvm**. The `svm()` in **e1071** and the `ksvm()` function in **kernlab** seem to be on par in terms of training time performance with the `svm()` function being slightly faster on multi-class problems.

	`ksvm()` (**kernlab**)	`svm()` (**e1071**)	`svmlight()` (**klaR**)	`svmpath()` (**svmpath**)
spam	18.50	17.90	34.80	34.00
musk	1.40	1.30	4.65	13.80
Vowel	1.30	0.30	21.46	NA
DNA	22.40	23.30	116.30	NA
BreastCancer	0.47	0.36	1.32	11.55
BostonHousing	0.72	0.41	92.30	NA

Table 2: The training times for the SVM implementations on different datasets in seconds. Timings where done on an AMD Athlon 1400 Mhz computer running Linux.

9. Conclusions

Table 3 provides a quick overview of the four SVM implementations. `ksvm()` in **kernlab** is a flexible SVM implementation which includes the most SVM formulations and kernels and allows for user defined kernels as well. It provides many useful options and features like a method for plotting, class probabilities output, cross validation error estimation, automatic hyper-parameter estimation for the Gaussian RBF kernel, but lacks a proper model selection tool. The `svm()` function in **e1071** is a robust interface to the award winning **libsvm** SVM library and includes a model selection tool, the `tune()` function, and a sparse matrix interface

	`ksvm()` (**kernlab**)	`svm()` (**e1071**)	`svmlight()` (**klaR**)	`svmpath()` (**svmpath**)
Formulations	C-SVC, ν-SVC, C-BSVC, spoc-SVC, one-SVC, ϵ-SVR, ν-SVR, ϵ-BSVR	C-SVC, ν-SVC, one-SVC, ϵ-SVR, ν-SVR	C-SVC, ϵ-SVR	binary C-SVC
Kernels	Gaussian, polynomial, linear, sigmoid, Laplace, Bessel, Anova, Spline	Gaussian, polynomial, linear, sigmoid	Gaussian, polynomial, linear, sigmoid	Gaussian, polynomial
Optimizer	SMO, TRON	SMO	chunking	NA
Model Selection	hyper-parameter estimation for Gaussian kernels	grid-search function	NA	NA
Data	formula, matrix	formula, matrix, sparse matrix	formula, matrix	matrix
Interfaces	`.Call`	`.C`	temporary files	`.C`
Class System	`S4`	`S3`	none	`S3`
Extensibility	custom kernel functions	NA	NA	custom kernel functions
Add-ons	plot function	plot functions, accuracy	NA	plot function
License	GPL	GPL	non-commercial	GPL

Table 3: A quick overview of the SVM implementations.

along with a `plot()` method and features like accuracy estimation and class-probabilities output, but does not give the user the flexibility of choosing a custom kernel. `svmlight()` in package **klaR** provides a very basic interface to **SVMlight** and has many drawbacks. It does not exploit the full potential of **SVMlight** and seems to be quite slow. The **SVMlight** license is also quite restrictive and in particular only allows non-commercial usage. `svmpath()` does not provide many features but can nevertheless be used as an exploratory tool, in particular for locating a proper value for the regularization parameter $\lambda = 1/C$.

The existing implementations provide a relatively wide range of features and options but the implementations can be extended by incorporating new features which arise in the ongoing research in SVM. One useful extension would allowing to weight the observations (Lin and Wang 1999) which is currently not supported by any of the implementations. Other extensions include the return of the original predictor coefficients in the case of the linear kernel (again not supported by any of the four implementations) and an interface and kernel for doing computations directly on structured data like string trees for text mining applications (Watkins 2000).

References

Blake C, Merz C (1998). "UCI Repository of Machine Learning Databases." University of California, Irvine, Dept. of Information and Computer Sciences, URL http://www.ics.uci.edu/~mlearn/MLRepository.html.

Canu S, Grandvalet Y, Rakotomamonjy A (2003). "SVM and Kernel Methods MATLAB Toolbox." Perception Systèmes et Information, INSA de Rouen, Rouen, France. URL http://asi.insa-rouen.fr/~arakotom/toolbox/index.

Caputo B, Sim K, Furesjo F, Smola A (2002). "Appearance-based Object Recognition using SVMs: Which Kernel Should I Use?" In "Proceedings of NIPS Workshop on Statistical Methods for Computational Experiments in Visual Processing and Computer Vision, Whistler, 2002," .

Chambers JM (1998). *Programming with Data*. Springer-Verlag, New York. ISBN 0-387-98503-4.

Chang CC, Lin CJ (2001). "**libsvm**: A Library for Support Vector Machines." URL http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Collobert R, Bengio S, Mariéthoz J (2002). "Torch: A Modular Machine Learning Software Library." URL http://www.torch.ch/.

Crammer K, Singer Y (2000). "On the Learnability and Design of Output Codes for Multiclass Prolems." *Computational Learning Theory*, pp. 35–46. URL http://www.cs.huji.ac.il/~kobics/publications/mlj01.ps.gz.

Dimitriadou E, Hornik K, Leisch F, Meyer D, Weingessel A (2005). "**e1071**: Misc Functions of the Department of Statistics (e1071), TU Wien, Version 1.5-11." URL http://CRAN.R-project.org/.

Gammerman A, Bozanic N, Schölkopf B, Vovk V, Vapnik V, Bottou L, Smola A, Watkins C, LeCun Y, Saunders C, Stitson M, Weston J (2001). "Royal Holloway Support Vector Machines." URL http://svm.dcs.rhbnc.ac.uk/dist/index.shtml.

Guermeur Y (2004). "**M-SVM**." Lorraine Laboratory of IT Research and its Applications, URL http://www.loria.fr/~guermeur/.

Gunn SR (1998). "MATLAB Support Vector Machines." University of Southampton, Electronics and Computer Science, URL http://www.isis.ecs.soton.ac.uk/resources/svminfo/.

Hastie T (2004). "**svmpath**: The SVM Path algorithm." R package, Version 0.9. URL http://CRAN.R-project.org/.

Hastie T, Rosset S, Tibshirani R, Zhu J (2004). "The Entire Regularization Path for the Support Vector Machine." *Journal of Machine Learning Research*, **5**, 1391–1415. URL http://www.jmlr.org/papers/volume5/hastie04a/hastie04a.pdf.

Hsu CW, Lin CJ (2002a). "A Comparison of Methods for Multi-class Support Vector Machines." *IEEE Transactions on Neural Networks*, **13**, 1045–1052. URL http://www.csie.ntu.edu.tw/~cjlin/papers/multisvm.ps.gz.

Hsu CW, Lin CJ (2002b). "A Comparison of Methods for Multi-class Support Vector Machines." *IEEE Transactions on Neural Networks*, **13**, 415–425. URL http://www.csie.ntu.edu.tw/~cjlin/papers/multisvm.ps.gz.

Hsu CW, Lin CJ (2002c). "A Simple Decomposition Method for Support Vector Machines." *Machine Learning*, **46**, 291–314. URL http://www.csie.ntu.edu.tw/~cjlin/papers/decomp.ps.gz.

Joachims T (1999). "Making Large-scale SVM Learning Practical." In "Advances in Kernel Methods – Support Vector Learning," chapter 11. MIT Press. URL http://www-ai.cs.uni-dortmund.de/DOKUMENTE/joachims_99a.ps.gz.

Karatzoglou A, Smola A, Hornik K, Zeileis A (2004). "**kernlab** – An S4 Package for Kernel Methods in R." *Journal of Statistical Software*, **11**(9). URL http://www.jstatsoft.org/counter.php?id=105&url=v11/i09/v11i09.pdf&ct=1.

Karatzoglou A, Smola A, Hornik K, Zeileis A (2005). "**kernlab** – Kernel Methods." R package, Version 0.6-2. URL http://CRAN.R-project.org/.

Knerr S, Personnaz L, Dreyfus G (1990). "Single-layer Learning Revisited: A Stepwise Procedure for Building and Training a Neural Network." In J Fogelman (ed.), "Neurocomputing: Algorithms, Architectures and Applications," Springer-Verlag.

Kreßel U (1999). "Pairwise Classification and Support Vector Machines." In B Schölkopf, CJC Burges, AJ Smola (eds.), "Advances in Kernel Methods – Support Vector Learning," pp. 255–268. MIT Press, Cambridge, MA.

Leisch F, Dimitriadou E (2001). "**mlbench**—A Collection for Artificial and Real-world Machine Learning Benchmarking Problems." R package, Version 0.5-6. URL http://CRAN.R-project.org/.

Lin CF, Wang SD (1999). "Fuzzy Support Vector Machines." *IEEE Transactions on Neural Networks*, **13**, 464–471. URL ftp://ftp.cs.wisc.edu/math-prog/tech-reports/98-18.ps.

Lin CJ, More JJ (1999). "Newton's Method for Large-scale Bound Constrained Problems." *SIAM Journal on Optimization*, **9**, 1100–1127. URL http://www-unix.mcs.anl.gov/~more/tron/.

Lin CJ, Weng RC (2004). "Probabilistic Predictions for Support Vector Regression." URL http://www.csie.ntu.edu.tw/~cjlin/papers/svrprob.pdf.

Lin HT, Lin CJ, Weng RC (2001). "A Note on Platt's Probabilistic Outputs for Support Vector Machines." URL http://www.csie.ntu.edu.tw/~cjlin/papers/plattprob.ps.

Mangasarian O, Musicant D (1999). "Successive Overrelaxation for Support Vector Machines." *IEEE Transactions on Neural Networks*, **10**, 1032–1037. URL ftp://ftp.cs.wisc.edu/math-prog/tech-reports/98-18.ps.

Meyer D, Leisch F, Hornik K (2003). "The Support Vector Machine Under Test." *Neurocomputing*, **55**, 169–186.

Osuna E, Freund R, Girosi F (1997). "Improved Training Algorithm for Support Vector Machines." In "IEEE NNSP Proceedings 1997," URL http://citeseer.ist.psu.edu/osuna97improved.html.

Platt JC (1998). "Fast Training of Support Vector Machines Using Sequential Minimal Optimization." In B Schölkopf, CJC Burges, AJ Smola (eds.), "Advances in Kernel Methods – Support Vector Learning," pp. 185–208. MIT Press, Cambridge, MA. URL http://research.microsoft.com/~jplatt/abstracts/smo.html.

Platt JC (2000). "Probabilistic Outputs for Support Vector Machines and Comparison to Regularized Likelihood Methods." In A Smola, P Bartlett, B Schölkopf, D Schuurmans (eds.), "Advances in Large Margin Classifiers," MIT Press, Cambridge, MA. URL http://citeseer.nj.nec.com/platt99probabilistic.html.

R Development Core Team (2005). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Roever C, Raabe N, Luebke K, Ligges U (2005). "**klaR** – Classification and Visualization." R package, Version 0.4-1. URL http://CRAN.R-project.org/.

Rüping S (2004). "**mySVM** – A Support Vector Machine." University of Dortmund, Computer Science, URL http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/index.html.

Schölkopf B, Platt J, Shawe-Taylor J, Smola AJ, Williamson RC (1999). "Estimating the Support of a High-Dimensonal Distribution." URL http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-99-87.

Schölkopf B, Smola A (2002). *Learning with Kernels*. MIT Press.

Schölkopf B, Smola AJ, Williamson RC, Bartlett PL (2000). "New Support Vector Algorithms." *Neural Computation*, **12**, 1207–1245. URL http://caliban.ingentaselect.com/vl=3338649/cl=47/nw=1/rpsv/cgi-bin/cgi?body=linker&reqidx=0899-7667(2000)12:5L.1207.

Schwaighofer A (2005). "SVM Toolbox for MATLAB." Intelligent Data Analysis group (IDA), Fraunhofer FIRST, URL http://ida.first.fraunhofer.de/~anton/software.html.

Tax DMJ, Duin RPW (1999). "Support Vector Domain Description." *Pattern Recognition Letters*, **20**, 1191–1199. URL http://www.ph.tn.tudelft.nl/People/bob/papers/prl_99_svdd.pdf.

The MathWorks (2005). "MATLAB – The Language of Technical Computing." URL http://www.mathworks.com/.

Vapnik V (1998). *Statistical Learning Theory*. Wiley, New York.

Vishwanathan SVN, Smola A, Murty N (2003). "SimpleSVM." In "Proceedings of the 20th International Conference on Machine Learning ICML-03," AAAI Press. URL http://www.hpl.hp.com/conferences/icml2003/papers/352.pdf.

Watkins C (2000). "Dynamic Alignment Kernels." In A Smola, PL Bartlett, B Schölkopf, D Schuurmans (eds.), "Advances in Large Margin Classifiers," pp. 39–50. MIT Press, Cambridge, MA.

Weingessel A (2004). "**quadprog** – Functions to Solve Quadratic Programming Problems." R package, Version 1.4-7. URL http://CRAN.R-project.org/.

Wu TF, Lin CJ, Weng RC (2003). "Probability Estimates for Multi-class Classification by Pairwise Coupling." *Advances in Neural Information Processing*, **16**. URL http://books.nips.cc/papers/files/nips16/NIPS2003_0538.pdf.

A. SVM formulations

A.1. ν-SVM formulation for classification

The primal quadratic programming problem for the ν-SVM is the following:

$$\begin{aligned} \text{minimize} \quad & t(\mathbf{w}, \xi, \rho) = \frac{1}{2}\|\mathbf{w}\|^2 - \nu\rho + \frac{1}{m}\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & y_i(\langle\Phi(x_i), \mathbf{w}\rangle + b) \geq \rho - \xi_i \qquad (i = 1, \ldots, m) \\ & \xi_i \geq 0 \qquad (i = 1, \ldots, m), \qquad \rho \geq 0. \end{aligned} \tag{23}$$

The dual is of the form:

$$\begin{aligned} \text{maximize} \quad & W(\alpha) = -\frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j k(x_i, x_j) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq \frac{1}{m} \qquad (i = 1, \ldots, m) \\ & \sum_{i=1}^{m}\alpha_i y_i = 0 \\ & \sum_{i=1}^{m}\alpha_i \geq \nu \end{aligned} \tag{24}$$

A.2. spoc-SVM for classification

The dual of the Crammer and Singer multi-class SVM problem is of the form:

$$\begin{aligned} \text{maximize} \quad & W(\alpha) = \sum_{i=1}^{l}\alpha_i\epsilon_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j k(x_i, x_j) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \qquad (i = 1, \ldots, m) \\ & \sum_{m=1}^{k}\alpha_i^m = 0, \qquad (i = 1, \ldots, l) \\ & \sum_{i=1}^{m}\alpha_i \geq \nu \end{aligned} \tag{25}$$

A.3. Bound constraint C-SVM for classification

The primal form of the bound constraint C-SVM formulation is:

$$\begin{aligned} \text{minimize} \quad & t(\mathbf{w}, \xi) = \frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{2}\beta^2 + \frac{C}{m}\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & y_i(\langle\Phi(x_i), \mathbf{w}\rangle + b) \geq 1 - \xi_i \qquad (i = 1, \ldots, m) \\ & \xi_i \geq 0 \qquad (i = 1, \ldots, m) \end{aligned} \tag{26}$$

The dual form of the bound constraint C-SVM formulation is:

$$\begin{aligned} \text{maximize} \quad & W(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j(y_iy_j + k(x_i, x_j)) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq \frac{C}{m} \qquad (i = 1, \ldots, m) \\ & \sum_{i=1}^{m}\alpha_i y_i = 0. \end{aligned} \tag{27}$$

A.4. SVM for regression

The dual form of the ϵ-SVM regression is:

$$\text{maximize}\alpha \in \mathbf{R}^m = \begin{cases} -\frac{1}{2}\sum_{i,j=1}^{m}(\alpha_i^* - \alpha_i)(\alpha_i^* - \alpha_i)k(x_i, x_j) \\ \\ -\epsilon\sum_{i=1}^{m}(\alpha_i^* + \alpha_i) + \sum_{i=1}^{m} y_i(\alpha_i^* - \alpha_i) \end{cases} \tag{28}$$
$$\text{subject to } \sum_{i=1}^{m}(\alpha_i - \alpha_i^*) = 0 \text{ and } a_i, a_i^* \in [0, C/m]$$

The primal form of the ν-SVM formulation is:

$$\begin{aligned} \text{minimize} \quad & t(\mathbf{w}, \xi^*, \epsilon) = \frac{1}{2}\|\mathbf{w}\|^2 + \frac{C}{\nu\epsilon} + \frac{1}{m}\sum_{i=1}^{m}(\xi_i + \xi_i^*) \\ \text{subject to} \quad & (\langle\Phi(x_i), \mathbf{w}\rangle + b) - y_i \geq \epsilon - \xi_i \qquad (i = 1, \ldots, m) \qquad (29) \\ & y_i - (\langle\Phi(x_i), \mathbf{w}\rangle + b) \geq \epsilon - \xi_i^* \qquad (i = 1, \ldots, m) \qquad (30) \\ & \xi_i^* \geq 0, \qquad \epsilon \geq 0, \qquad (i = 1, \ldots, m) \end{aligned}$$

The dual form of the ν-SVM formulation is:

$$\begin{aligned} \text{maximize} \quad & W(\alpha^*) = \sum_{i=1}^{m}(\alpha_i^* - \alpha_i)y_i - \frac{1}{2}\sum_{i,j=1}^{m}(\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j)k(x_i, x_j) \\ \text{subject to} \quad & \sum_{i=1}^{m}(\alpha_i - \alpha_i^*) \end{aligned} \tag{31}$$

$$\alpha_i^* \in \left[0, \frac{C}{m}\right],$$
$$\sum_{i=1}^{m}(\alpha_i + \alpha_i^*) \leq C\nu$$

A.5. SVM novelty detection

The dual form of the SVM QP for novelty detection is:

$$\begin{aligned} \text{minimize} \quad & W(\alpha) = \sum_{i,j} \alpha_i \alpha_j k(x_i, x_j) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq \frac{1}{\nu m} \qquad (i = 1, \ldots, m) \\ & \sum_i \alpha_i = 1 \end{aligned} \tag{32}$$

Affiliation:

Alexandros Karatzoglou
Institute für Statistik und Wahrscheinlichkeitstheorie
Technische Universität Wien
A-1040 Wien, Austria
E-mail: alexis@ci.tuwien.ac.at
Tel: +43/1/58801-10772
Fax: +43/1/58801-10798

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 15, Issue 9 *Submitted:* 2005-10-24
April 2006 *Accepted:* 2006-04-06