WIRTSCHAFTS
UNIVERSITÄT
WIEN

kernlab

An R Package for Kernel Learning

Alexandros Karatzoglou Alex Smola Kurt Hornik Achim Zeileis

http://www.ci.tuwien.ac.at/~zeileis/

Overview

- Implementing learning algorithms:
  - Why should we write software (*and make it available*)?
  - Why should it be open source software?
  - What should be the guiding principles for implementation?
  - Why R?

- Kernel learning implementation in **kernlab**
  - Kernels and kernel expressions
  - Learning algorithms
  - Other tools

Why software?

Authors of learning algorithms usually have an implementation for own applications and running simulations and benchmarks, *but not necessarily in production quality*.

Why should they be interested in taking the extra effort to adapt them to more general situations, document it and make it available to others?

Supplying software that is sufficiently easy to use is an excellent way of *communicating ideas and concepts* to researchers and practitioners.

Given the description of an excellent method and code for a good one, you choose . . . ?

Why open source?

Claerbout's principle

> An article about computational science in a scientific publication is *not* the scholarship itself, it is merely *advertising* of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.

To evaluate the correctness of all the results in such an article, the source code must also be available for inspection. Only this way gradual refinement of computational (and conceptual) tools is possible.

Implementation principles

Task: Turn conceptual tools into computational tools

Goals: desirable features

- easy to use,
- numerically reliable,
- computationally efficient,
- flexible and extensible,
- re-usable components,
- object oriented,
- reflect features of the conceptual method.

Implementation principles

Problem: often antagonistic. E.g., computational efficiency vs. extensibility.

Guiding principle: The implementation should be guided by the properties of the underlying methods while trying to ensure as much efficiency and accuracy as possible.

The resulting functions should do what we think an algorithm does conceptually.

Implementation principles

In practice: Many implementations are still guided by the limitations that programming languages used to have (and some still have) where everything has to be represented by numeric vectors and matrices.

What language features are helpful for improving this?

Implementation principles

Object orientation: Create (potentially complex) objects that represent an abstraction of a procedure or type of data. Methods performing typical tasks can be implemented.

Functions as first-class objects: Functions are a basic data type that can be passed to and returned by another function.

Lexical scope: more precisely *nested lexically scoped functions*. Returned functions (e.g., prediction methods) can have free variables stored in the function closure.

Implementation principles

Compiled code: Combine convenience of using interpreted code and efficiency of compiled code by (byte) compilation or dynamic linking of compiled code.

Re-usable components: The programming environment should provide procedures that the implementation can build on. Likewise, the implementation should create objects that can be re-used in other programs.

Why R?

R offers all these features and more:

- R is a full-featured interactive computational environment for data analysis, inference and visualization.
- R is an open source project, released under GPL.
- Developed for the Unix, Windows and Macintosh families of operating systems by the R Development Core Team.
- Offers several means of object orientation, including S3 and S4 classes.

Why R?

- Everything in R is an object, including functions and even function calls.
- Nested functions are lexically scoped.
- Allows for dynamic linking of C, C++ or FORTRAN code.
- Highly extensible with a fast-growing list of add-on packages.

Why R?

Software delivery is particularly easy:

R itself and more than 470 packages are available (most of them under the GPL) from the Comprehensive R Archive Network (CRAN):

http://CRAN.R-project.org/

and can easily be installed from within R via, e.g.

```
R> install.packages("kernlab")
```

Kernel learning

Motivation: Flexible implementation of a collection of kernel learning techniques, in particular support vector machines (SVMs), employing different kernels.

Goals:

- Make it easy to plug in new kernels (potentially user-defined).
- Provide typical kernel expressions as building blocks to high-level algorithms.
- Provide further tools typically required for kernel learning.

Kernel learning

Problem: Many learning tasks are difficult to solve in the observed feature space.

Solution: Project observations into a high-dimensional space where the learning task is easier to solve.

Kernel trick: Kernels compute dot product

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle$$

in a high dimensional projection space.

Kernel learning

Kernel learners use the dot product representation of data, i.e., typically rely on expressions like the kernel matrix K defined by

$$K_{ij} \;=\; k(x_i, x_j)$$

with $i, j = 1, \ldots, n$.

Changing the kernel changes the projection of the data and hence the distances in the projection space.

Kernel learning in kernlab

Simple idea: Kernels are functions $k(x, x')$ which, given two vectors x and x', compute a scalar.

Implementation: Kernels are represented as objects of class `"kernel"` (extending the `"function"` class).

These functions can be passed as an argument to *generic functions* which evaluate more useful kernel expressions like a kernel matrix.

Kernel expressions

- `kernelMatrix(kernel, x)` computes the kernel matrix

$$K_{ij} = k(x_i, x_j)$$

- `kernelPol(kernel, x, y)` computes the matrix

$$P_{ij} = y_i y_j \, k(x_i, x_j)$$

- `kernelMult(kernel, x, alpha)` computes the vector

$$f_i = \sum_{j=1}^{m} k(x_i, x_j) \, \alpha_j$$

Kernel expressions

```
R> library(kernlab)
R> set.seed(20050305)
R> X <- matrix(rnorm(300), ncol = 100)
```

```
R> rbf1 <- function(x, y) exp(-0.1 * sum((x - y)^2))
R> class(rbf1) <- "kernel"
R> rbf1(X[1, ], X[2, ])
```

```
[1] 4.303873e-10
```

Kernel expressions

```
R> kernelMatrix(rbf1, X)
```

```
             [,1]         [,2]         [,3]
[1,] 1.000000e+00 4.303873e-10 8.321654e-09
[2,] 4.303873e-10 1.000000e+00 3.780441e-10
[3,] 8.321654e-09 3.780441e-10 1.000000e+00
```

Problems:

- These default methods can become very slow.
- For hyperparameter tuning, a notion of *families* of kernels is needed.

Kernel expressions

Solution:

- **kernlab** provides “kernel generating functions” creating kernel functions from commonly used families.

- Using object orientation mechanism memory efficient methods are provided for vectorized computation of kernel expressions in compiled code.

- For user-defined kernels new methods to `kernelMatrix` and friends can be defined.

Kernel families

- Linear kernel `vanilladot`

$$k(x,x') \; = \; \langle x, x'\rangle$$

- Gaussian radial basis `rbfdot`

$$k(x,x') \; = \; \mathsf{exp}(-\sigma \cdot \|x - x'\|^2)$$

- Hyperbolic tangent `tanhdot`

$$k(x,x') \; = \; \mathsf{tanh}\left(\mathsf{scale} \cdot \langle x, x'\rangle + \mathsf{offset}\right)$$

- Polynomial kernel `polydot`

$$k(x,x') \; = \; (\mathsf{scale} \cdot \langle x, x'\rangle + \mathsf{offset})^{\mathsf{degree}}$$

Kernel families

These kernel generating functions create kernel functions objects that store the kernel family and its parameters within the object.

```
R> rbf2 <- rbfdot(sigma = 0.1)
R> rbf2
```

```
Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.1
```

Kernel families

```
R> rbf2(X[1, ], X[2, ])

             [,1]
[1,] 4.303873e-10


R> kernelMatrix(rbf2, X)

             [,1]         [,2]         [,3]
[1,] 1.000000e-00 4.303873e-10 8.321654e-09
[2,] 4.303873e-10 1.000000e+00 3.780441e-10
[3,] 8.321654e-09 3.780441e-10 1.000000e+00
```

Kernel learners

Functions implementing kernel learning algorithms should take the kernel function as an argument.

These high-level functions should also employ the generic functions like `kernelMatrix`

- such that these operations do not have to be re-implemented for each algorithm,
- and to exploit the efficient methods supplied for certain kernel families.

Kernel learners

- Support vector machine: `ksvm`
- Relevance vector machine: `rvm`
- Gaussian processes: `gausspr`
- Ranking: `ranking`
- Online learning: `onlearn`
- Spectral clustering: `specc`
- Kernel principal component analysis: `kpca`
- Kernel feature analysis: `kfa`
- Kernel canonical correlation analysis: `kcca`

Other tools

Quadratic optimizer: in many kernel-based algorithms (especially SVMs) a quadratic programming solver is needed. **kernlab** provides `ipop`, an optimizer using an interior point code for

$$\begin{array}{ll} \text{minimize} & c^\top x + \frac{1}{2} x^\top H x \\ \text{subject to} & b \leq Ax \leq b + r \\ & l \leq x \leq u \end{array}$$

Other tools

Incomplete Cholesky decomposition: As the kernel matrix is usually of low rank, employing an incomplete Cholesky factorization T is useful:

$$K = TT^\top$$

This is implemented in `chol.reduce` in **kernlab**.

Support vector machine

- C- and ν-SVM for classification, regression,
- Novelty detection (one-class classification),
- One-against-one and multi-class SVM formulation,
- Built-in cross-validation,
- Class probabilities output,
- Scaling of variables,
- Automatic σ estimation for RBF kernels.

- `ksvm` returns fitted object of class `"ksvm"`,
- Methods for `predict`, `show`, and `plot` are provided,
- Slots of fitted objects can be extracted via accessor functions.

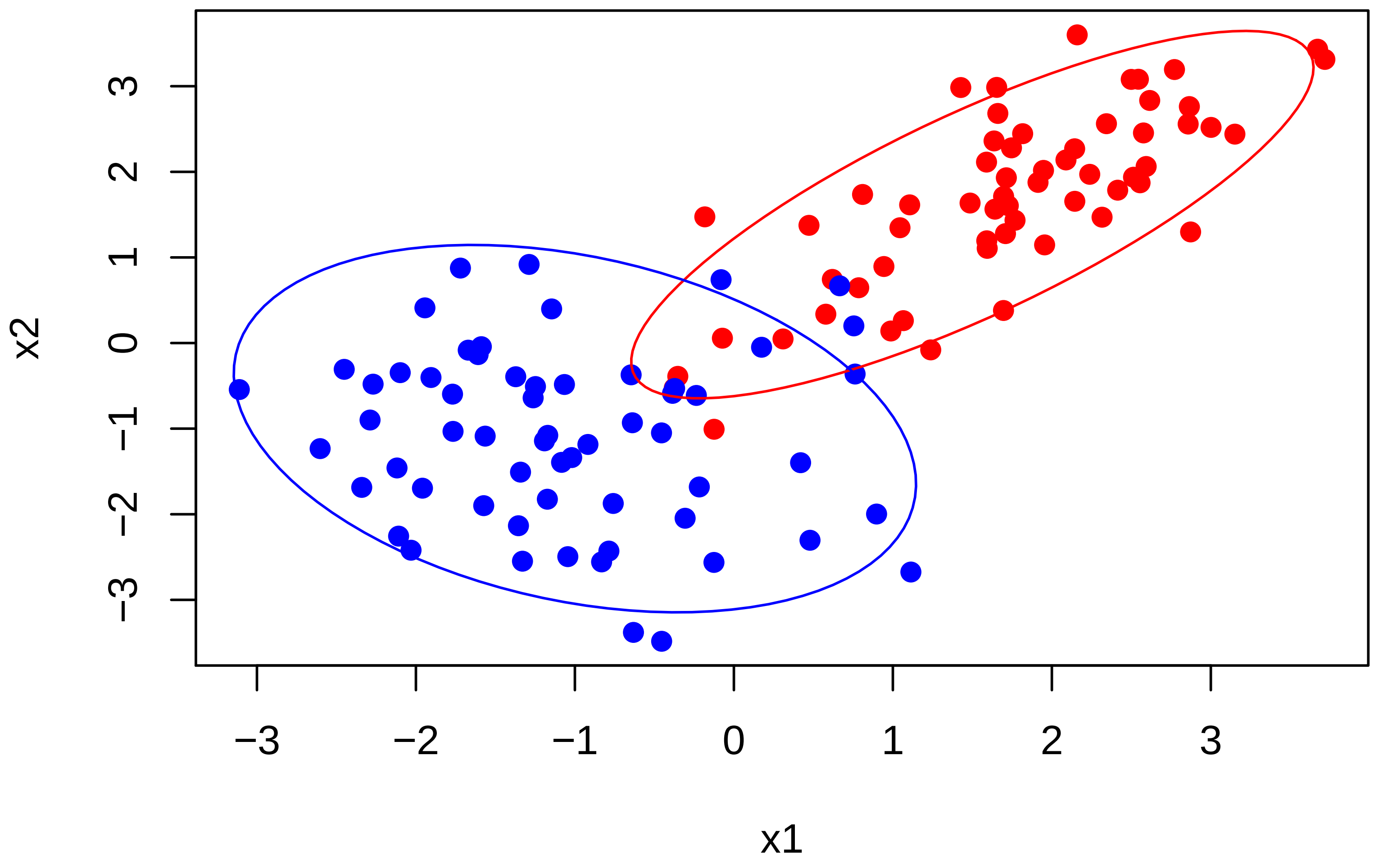
Examples: SVM

```
R> library(mvtnorm)
R> x1 <- rmvnorm(60, mean = c(1.5, 1.5),
  sigma = matrix(c(1, 0.8, 0.8, 1), ncol = 2))
R> x2 <- rmvnorm(60, mean = c(-1, -1),
  sigma = matrix(c(1, -0.3, -0.3, 1), ncol = 2))
R> X <- rbind(x1, x2)

R> ex1 <- data.frame(x1 = X[,1], x2 = X[,2],
  class = factor(rep(letters[1:2], c(60, 60))))
R> rm(x1, x2, X)

R> plot(x2 ~ x1, data=ex1, pch=19, col=rep(c(2, 4), c(60, 60)))
R> library(ellipse)
R> lines(ellipse(0.8, centre = c(1.5, 1.5), level = 0.9), col = 2)
R> lines(ellipse(-0.3, centre = c(-1, -1), level = 0.9), col = 4)
```

Examples: SVM



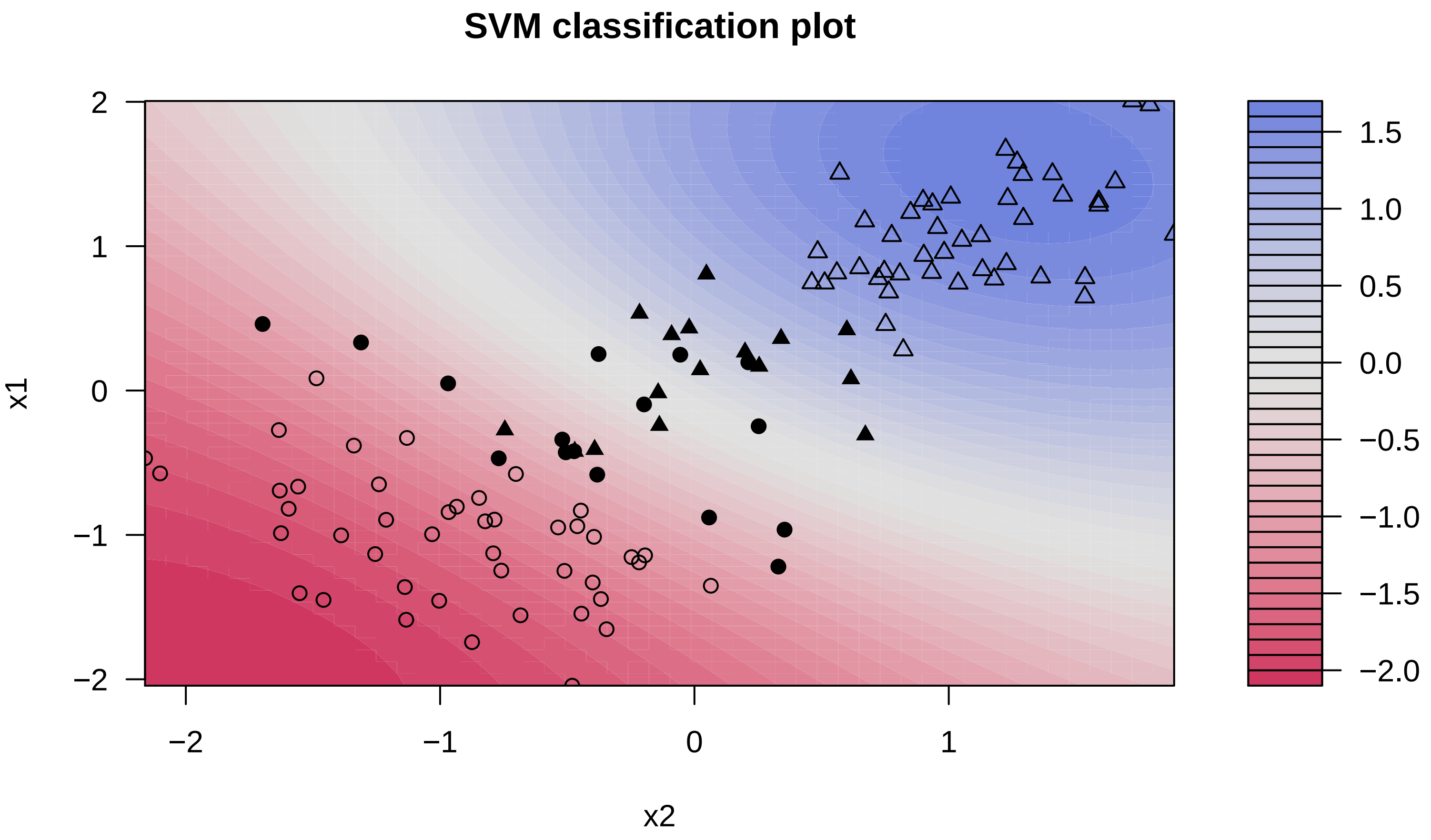
Examples: SVM

```
R> fm1 <- ksvm(class ~ x1 + x2, data = ex1, kernel = rbf1)

R> plot(fm1)
```

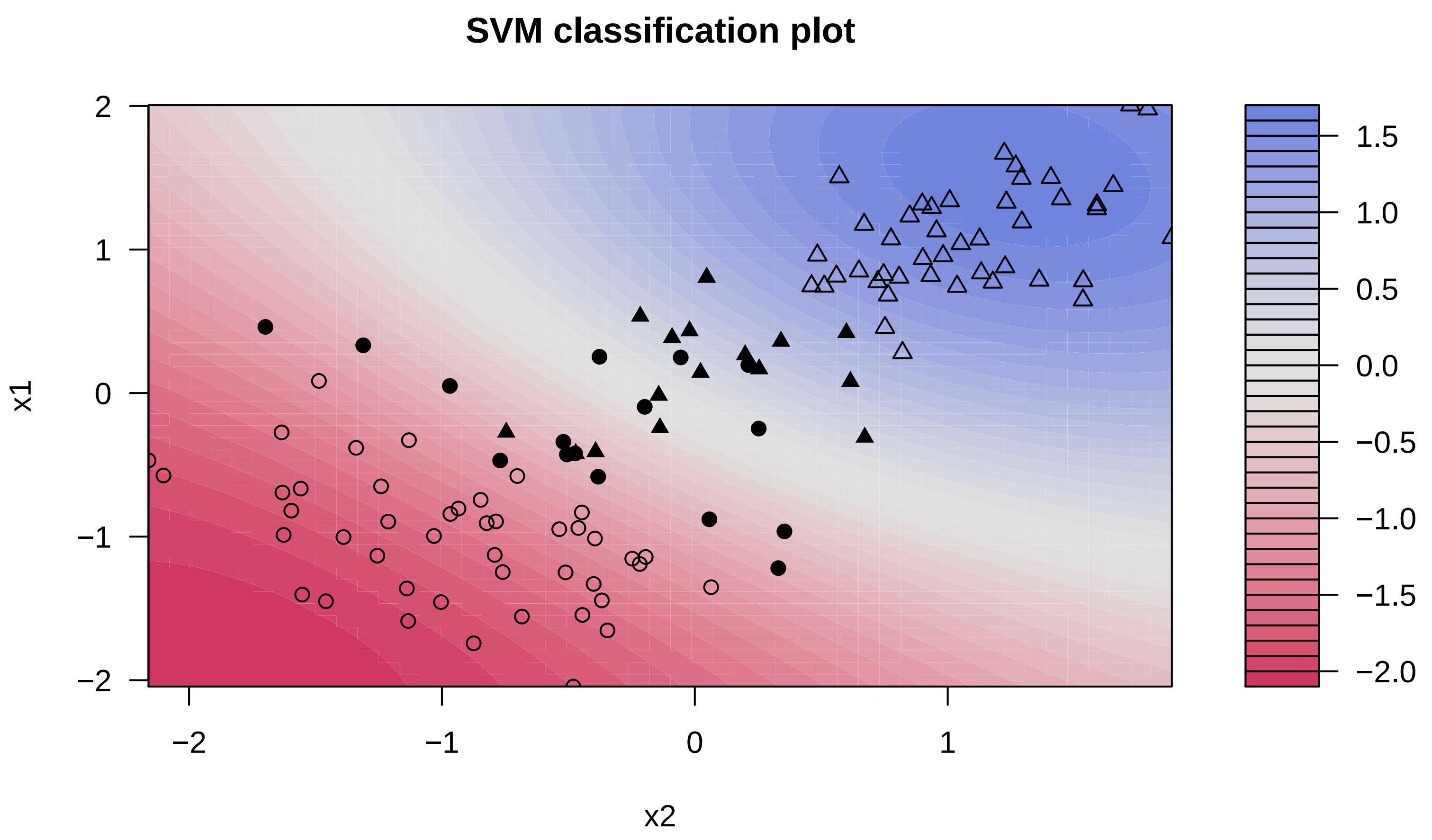
Examples: SVM

```
R> fm1 <- ksvm(class ~ x1 + x2, data = ex1, kernel = rbf1)
R> plot(fm1)
```



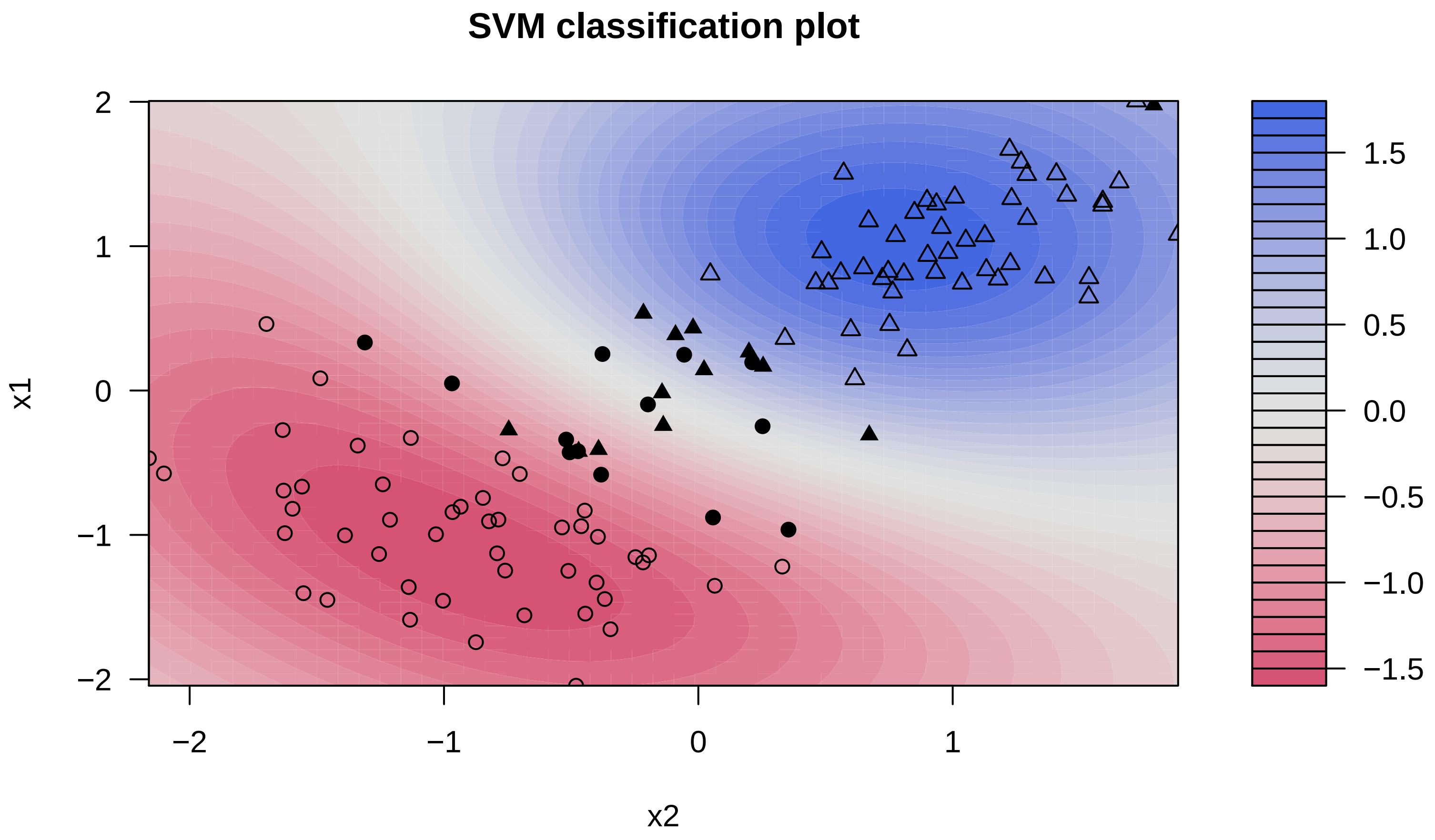
Examples: SVM

```
R> fm2 <- ksvm(class ~ x1 + x2, data = ex1, kernel = rbf2)
R> plot(fm2)
```



Examples: SVM

```
R> fm <- ksvm(class ~ x1 + x2, data = ex1)
R> plot(fm)
```



Examples: RVM

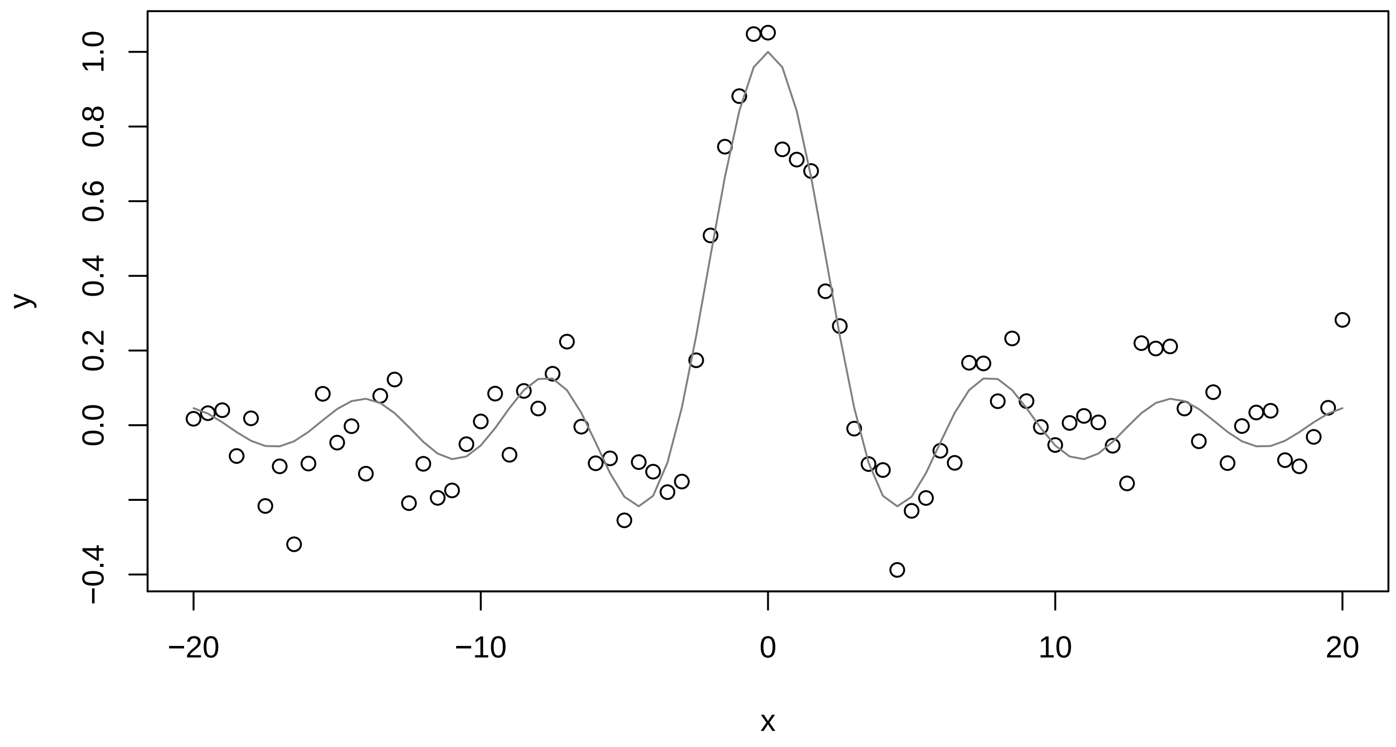
```
R> x <- seq(-20, 20, 0.5)
R> ymean <- sin(x)/x
R> ymean[41] <- 1

R> ex2 <- data.frame(x = x, ymean = ymean,
  y = ymean + rnorm(81, sd = 0.1))

R> rm(x, ymean)
```

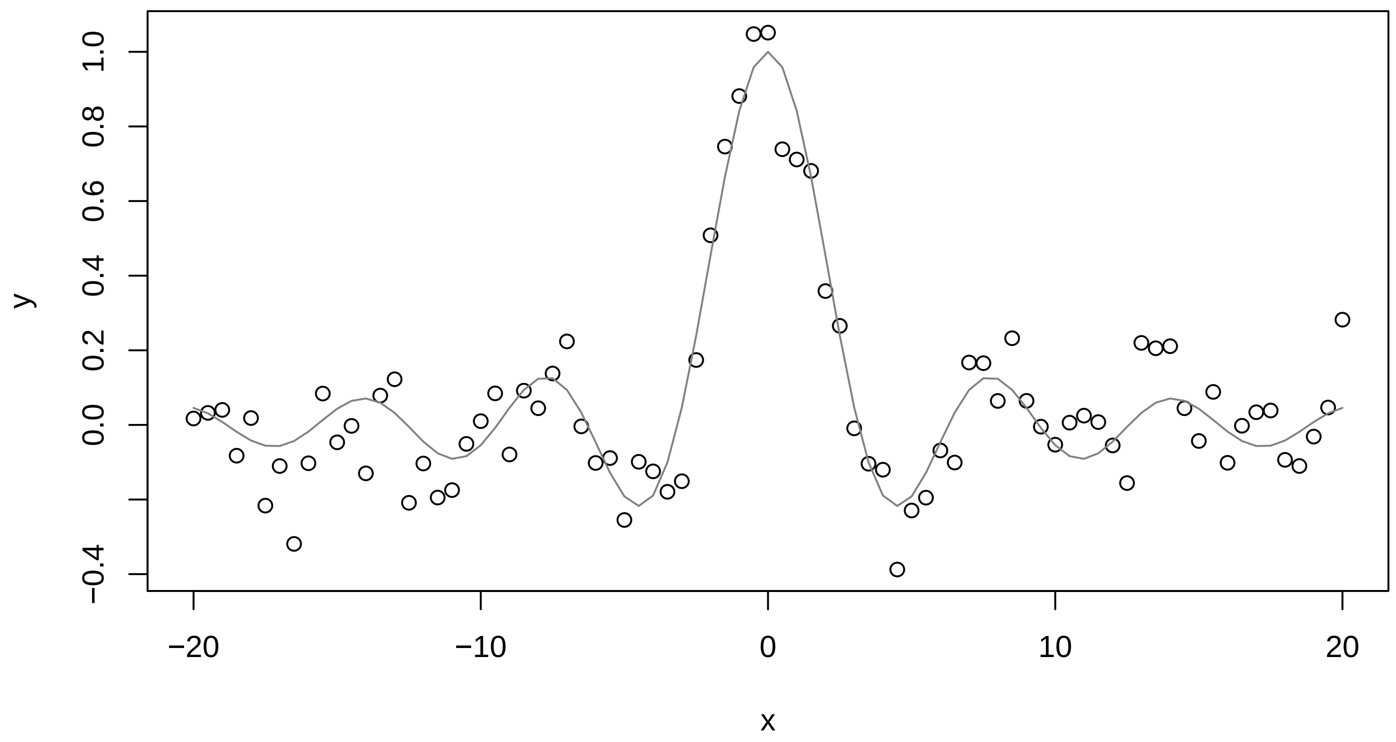
Examples: RVM

```
R> plot(y ~ x, data = ex2)
R> lines(ymean ~ x, data = ex2, col = grey(0.5))
```



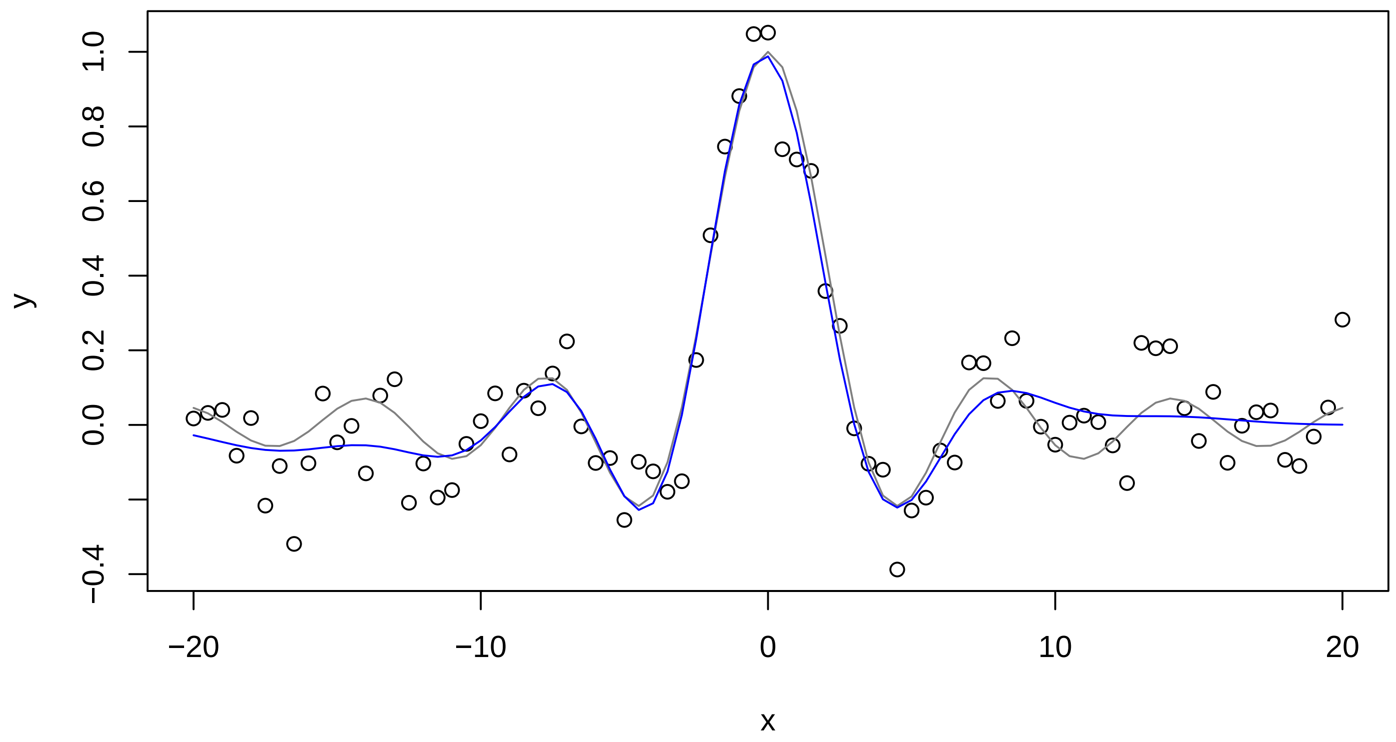
Examples: RVM

```
R> fm <- rvm(y ~ x, data = ex2)
```



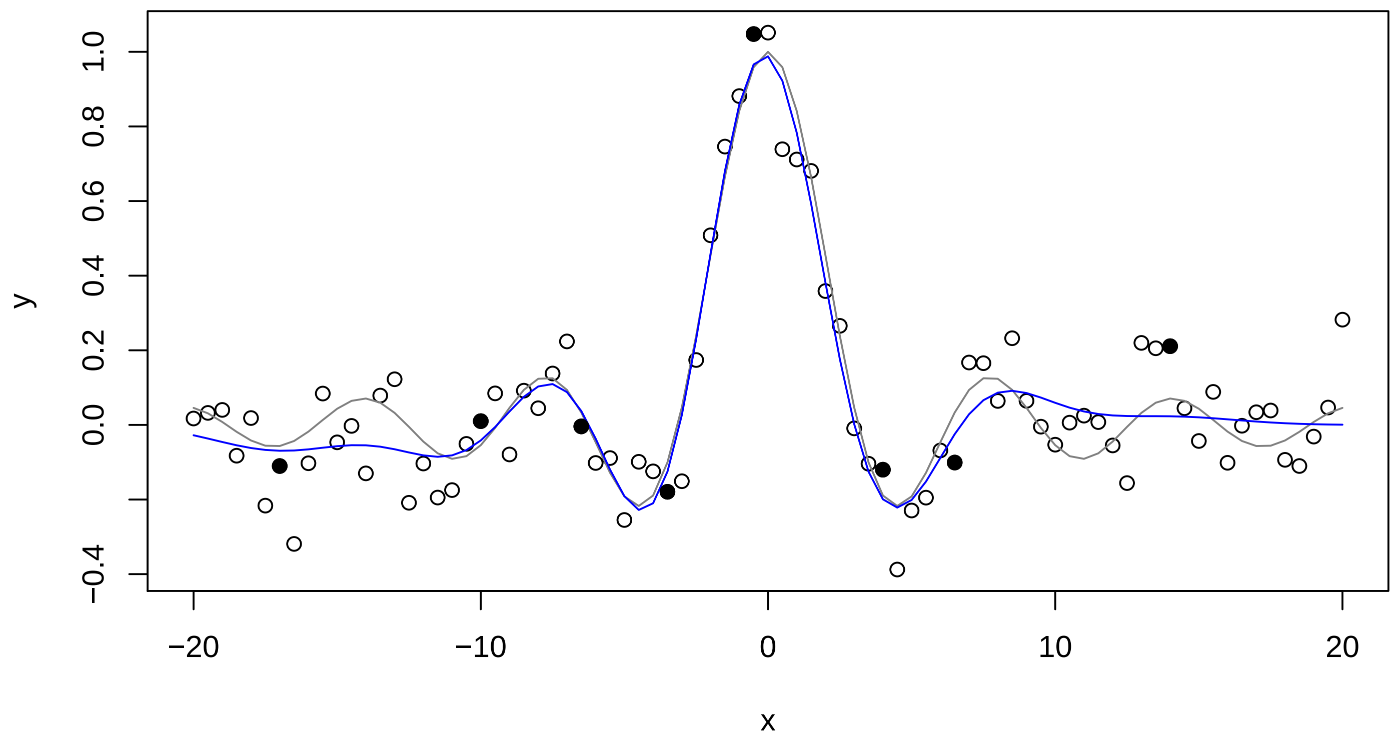
Examples: RVM

```
R> fm <- rvm(y ~ x, data = ex2)
R> lines(predict(fm, ex2) ~ x, data = ex2, col = 4)
```



Examples: RVM

```
R> fm <- rvm(y ~ x, data = ex2)
R> lines(predict(fm, ex2) ~ x, data = ex2, col = 4)
R> points(y[RVindex(fm)] ~ x[RVindex(fm)], data = ex2, pch = 19)
```



Examples: Spam data

Data set collect at the Hewlett-Packard labs:

- classifies 4601 e-mails as spam or non-spam,
- further 57 variables indicating frequency of certain words or characters.

Load data and select test set indices:

```
R> data(spam)
R> tindex <- sample(1:nrow(spam), 10)
```

Examples: Spam data

Fit SVM on training set:

```
R> fm <- ksvm(type ~ ., data = spam[-tindex, ], kernel = "rbfdot",
+      kpar = "automatic", C = 60, cross = 5)
```

```
Using automatic sigma estimation (sigest) for RBF or laplace kernel
```

Examples: Spam data

```
R> fm


Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 60

Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.0106338562059120

Number of Support Vectors : 875
Training error : 0.027881
Cross validation error : 0.061862
```

Examples: Spam data

Predictions on test set:

```
R> predict(fm, spam[tindex, ])
```

```
 [1] nonspam nonspam nonspam spam    spam    nonspam nonspam spam
[10] nonspam
Levels: nonspam spam
```

```
R> table(predict(fm, spam[tindex, ]), spam[tindex, 58])
```

```
          nonspam spam
  nonspam 7       0
  spam    0       3
```

Summary

kernlab provides a comprehensive, flexible and extensible toolbox for kernel learning in R.

1. Kernel functions

- Users can create and explore their own kernels.
- A function which takes two arguments and returns a scalar.
- Kernel can be plugged into any kernel learning algorithm like SVMs etc.
- Memory-efficient methods for kernel expressions can (*but do not have to*) be supplied.

Summary

2. Kernel learners

- Large collection of state-of-the art kernel learners.
- Infrastructure for creating new high-level algorithms.

3. Implemented in R

- Standard formula interface for model specification.
- Standard methods for predictions etc.
- Full statistical toolbox and powerful visualization techniques.

References

Alexandros Karatzoglou, Alex Smola, Kurt Hornik, Achim Zeileis (2004). “**kernlab** – An S4 Package for Kernel Methods in R,” *Journal of Statistical Software*, **11**(9).
URL `http://www.jstatsoft.org/v11/i09/`

Achim Zeileis (2004). “Implementing a Class of Structural Change Tests: An Econometric Computing Approach,” Department of Statistics and Mathematics, Wirtschaftsuniversität Wien, *Research Report Series*, **7**.

Torsten Hothorn, Friedrich Leisch, Achim Zeileis, Kurt Hornik (2004). “The Design and Analysis of Benchmark Experiments,” *Journal of Computational and Graphical Statistics*, Forthcoming.