SVLAB User's Manual

Alexandros Karatzoglou[*] and Alex J. Smola[†]

February 1, 2005

Abstract

SVLAB is an extensible, object oriented, library for kernel based learning in MATLAB. It contains various dot product primitives, quadratic optimizers for classification, regression, and a general purpose qp solver. Moreover, it provides optimizers of the SMO family (sequential minimal optimization), online training algorithms, algorithms for sparse greedy matrix approximation, kernel pca and kernel feature analysis, Laplacian SVM, and Gaussian Processes.

The library is extensible and available for public use under the restrictions of the GNU Public License.[1]

[*]Vienna University of Technology, Department of Statistics, Wiedner Hauptstraße 8-10, A-1040 Wien, Austria; `Alexandros.Karatzoglou@ci.tuwien.ac.at`

[†]Australian National University, Department of Engineering and RSISE, Canberra, 0200 ACT, Australia; `Alex.Smola@anu.edu.au`

[1]The latest version of this documentation can be found on the Kernel Machines Website at `http://www.kernel-machines.org/svlab`.

Contents

1 Introduction

The purpose of the SVLAB toolbox for MATLAB is to provide a rapid prototyping framework which will enable the user to test existing algorithms and implement new algorithms in a very short time. We hope that it will prove useful as a toolkit for further development in kernel algorithms.

Whenever possible we chose MATLAB .m over .mex files with a C backend. This does not always lead to the highest performance but makes the library very portable and easy to install. In particular this overcomes the problem with libc and glibc development systems for MATLAB 5 under Linux. For large scale production environments users may consider using `libkbl`, a C/C++ library for kernel based learning.[2]

For further information about kernel methods see the following books Herbrich [2002], Schölkopf and Smola [2003], Cristianini and Shawe-Taylor [2000], Joachims [2002], Vapnik [1995, 1998] and the tutorials by Burges [1998], Smola and Schölkopf [1998], Müller et al. [2001], Frieß et al. [1998].

1.1 Getting Started

On Windows unzip the file `svlab.zip` in a location where you would like the the library to reside.

```
c:\>mkdir svlab
c:\>cd svlab
c:\>unzip svlab.zip
```

Likewise, on Unix systems untar the library `svlab.tgz` by

```
gunzip svlab.tgz
tar -xvf svlab.tar
```

This completes all the installation needed for the library. To use it two paths have to be set, namely to the `svlab` directory itself, and also to the `svlab/common` subdirectory containing several auxiliary routines. This is best done (we assume that svlab is installed in `/usr/local`)by

```
addpath('/usr/local/svlab')
addpath('/usr/local/svlab/common')
```

The settings in Windows are analogous.

1.2 Reporting Bugs and Improvements

Please send e-mail to `Alex.Smola@anu.edu.au` or `alexis@ci.tu-wien.at` for bug reports, preferably with a patch how to fix it. If you have a module that you would like to be added to the library, please do not hesitate to send the classes plus the corresponding documentation (in LaTeX, following the same style guidelines as this user's manual) to the same address.

[2] `http://www.kernel-machines.org/libkbl`

1.3 Strategy

Presently there exist three main building blocks which comprise the library.

- The first are the actual dot products (`vanilla_dot, poly_dot, rbf_dot, tanh_dot, laplace_dot` ...) which are fast implementations of the kernel functions $k(x, x')$ as used in many learning algorithms.
- Secondly general purpose helper routines such as a cache for kernel evaluations, timing routines, file I/O, etc. are useful tools for data handling.
- Finally, there are several algorithms, e.g. for Support Vector Machines, Kernel PCA, or Sparse Matrix Approximations, which are built on top of the first two columns. These routines are fairly independent from each other but make heavy use of the underlying kernel routines for their work. In particular, they allow easy modifications from one kernel function ot another by simply passing a differed kernel object.

The library uses the object oriented features of MATLAB 5 an later. Hence it will not work with MATLAB 4.2 and earlier. Unfortunately the standard OO features provided by The MathWorks are far from being complete or useful. In particular, the concept of private and public variables does not exist. However, it is possible to fix some of these deficiencies without a large performance loss. A serious concern, however, is that function calls in MATLAB are call-by-value. To avoid bad performance some modifications are therefore needed, which would not arise in a proper programming language. We will explain the general ideas in the subsequent section.

1.4 Class Interfaces

The main idea is that all algorithms should be represented as classes. This allows the designer to set useful initial parameters for most algorithms in a manner that is more intuitive than the parameter vector technique as often used in the built-in MATLAB routines (e.g. the eigensolvers `eig` and `eigs` or the function minimizers `fmins`, which make extensive use of the options object). Sometimes this will cause some minor overhead in coding, however it adds consistency to the library. Below we list the common features a class interface should have. We use `vanilla_dot` as an example.

Constructor: By default class constructors should be callable without the need for any further arguments, e.g. we should be able to construct a kernel object by calling

```
>> kernel = vanilla_dot
Dot product
Type             : vanilla_dot
Block size       : 128
```

Besides that, it should also be possible to call the constructor with whatever parameters can be modified in the algorithm. In the present case this is the block size of a multiplication operation. In other words, we should be able to call

```
>> kernel = vanilla_dot(256)
Dot product
Type            : vanilla_dot
Block size      : 256
```

Multiple arguments and variable argument lengths are allowed. They are up to the discretion of the implementer. Finally there exists a third way of calling a constructor, namely with pre-formatted configuration strings. The latter are formed by keywords and underscores '_'. In the present case `blocksize` would be such a keyword and hence we obtain

```
>> kernel = vanilla_dot('blocksize_42')
Dot product
Type            : vanilla_dot
Block size      : 42
```

In particular, several keywords and configurations may be concatenated in one string. This is useful for automatic generation of filenames (we will explain the converse mapping via `display.m` below) and reconstruction of experiments.
It may appear as if this functionality would require a large coding overhead. The contrary is true, though. For demonstration purposes see the code section of the constructor `vanilla_dot.m`.

```
function d = vanilla_dot(a)

d.name      = 'vanilla_dot';
if nargin == 0
  d.blocksize = 128;                    % block size for sv_mult
elseif (nargin == 1) & isa(a, 'char')
  token = read_token(a, 'blocksize');
  d.blocksize = str2num(token);
elseif (nargin == 1) & isa(a, 'double')
  d.blocksize = a;
else
  error('wrong type of arguments');
end
d = class(d, 'vanilla_dot');            % make it a class
```

Display and Printing (`display.m`): This function is responsible for displaying objects and can be overloaded from the standard MATLAB defaults. We have to distinguish two different cases:

1. No return argument is required. In this case we want to display an object, e.g. after its construction in human readable form. With the kernel as defined above we obtain

   ```
   >> display(kernel)
   Dot product
   Type            : vanilla_dot
   Block size      : 42
   ```

2. `display.m` is called with one return argument. In this case it will generate a configuration string which fully describes the object (if possible) and can be used as a file name or later for reconstruction of the experiment via the constructor (see previous section). It is best described by an example:

```
>> x = display(kernel)

x =

vanilla_dot_blocksize_42
```

   The code to generate these outputs can be found here:

```
function x = display(d)

if nargout == 0
  tmp = sprintf('Dot product\nType       \t: %s ', ...
        '\nBlock size \t: %d', d.name, d.blocksize);
  disp(tmp);
else
  x = sprintf('%s_blocksize_%d', d.name, d.blocksize);
end
```

   It checks the number of output arguments and depending on their value returns one of the two desired answers.

Member Variables: In almost all object oriented languages one may access (public) member variables by a method similar to `object.variable = 5` or via `x = object.variable`. Unfortunately, MATLAB does not provide this functionality without additional code provided by the user. The methods `subsref.m` and `subsasgn.m` fix this deficiency. When **creating a new class**, all that needs to be done is to **copy these two files into the corresponding subdirectory**. This allows us to set member variables just in the same way as in normal MATLAB structures.[3] The following example shows us how. Assume we generated `vanilla_dot` by

```
>> kernel = vanilla_dot
Dot product
Type            : vanilla_dot
Block size      : 128
```

Then we may change the blocksize by

```
>> kernel.blocksize = 99
Dot product
Type            : vanilla_dot
Block size      : 99
```

[3]The downside is that there is no protection in terms of private and public variables. This, however, is a small price to pay in comparison to having to code every variable access explicitly. This problem could be overcome by initializing the constructors with a public list of variables and restricting the parsing of `subsref.m, subsasgn,m` to this subset of variables.

The blocksize can be obtained by

```
>> kernel.blocksize

ans =

    99
```

Methods: The access to methods of a class follows the default (inelegant) calling conventions of MATLAB. Hence, we access, e.g. `sv_dot` by calling

```
dotproduct = sv_dot(kernel, train_patterns, test_patterns);
```

This means that the first argument of a method has to be the object of the class the method is associated with.

Inheritance: Unfortunately inheritance in MATLAB can produce unexpected results (e.g. masking of member variables, etc.). Therefore we make as little use of it as possible (and have to duplicate some code).

1.5 Coding Style

The coding style follows closely the suggestions by The MathWorks. We begin with an example, namely `@vanilla_dot/sv_dot.m`.

```
function x = sv_dot(d, a, b)
%SV_DOT Basic template for the Dot Product
%
%       X = SV_DOT(D, A, B) returns the scalar product of A and B
%       (for three input arguments) or of A'*A (for two input
%       arguments) where D is the type of dot product used
%
%       see also DISPLAY, SV_DOT, SV_MULT, SV_POL

% File:         @vanilla_dot/sv_dot.m
%
% Author:       Alex J. Smola
% Created:      01/12/98
% Updated:      05/08/00
%
% This code is released under the GNU Public License
% Copyright by GMD FIRST and The Australian National University

if (nargin < 2) | (nargin > 3)
  error('wrong number of arguments');
elseif nargin == 2
  x = a' * a;
else
  x = a' * b;
end;
```

One can identify four units in the code:

Function header: This describes the interface of the code. If possible, do not change the header unless needed. See also Section 1.6 for details on calling conventions, variable names, and ordering of the arguments.

Help Information: The comments immediately following the function header will be returned by the MATLAB help system. If possible, the information provided there should be sufficient to understand the functions without the need for any further documentation.

Copyright Header: It contains information about the creation history of the file (slightly redundant with CVS) and the copyright under which the file has been released.

Code: If possible use comments, meaningful variable names and proper indentation. In Emacs the `matlab.el` mode is very useful to generate readable code. It can be obtained from the The MathWorks website at `http://www.mathworks.com/support/ftp/emacs_add_ons/matlab.el`. Besides that, the usual strategies for creating readable code apply.

1.6 Variable Names and Order

Because of the way Matlab works with matrices, it is better to represent datasets as colum-major matrices (i.e. matrices whose columns are the vectorial representations of datapoints). The interface to each implemented module can be seen by using the appropriate Matlab `help` command, but some basic rules/suggestions are as follows:

Objects need to be passed to the method which uses them (thanks to Matlab's weird implementation of oject orientation) as the first parameter.

The Kernel used by a function should be passed next.

Training Sets are often denoted as X in the literature, and most modules adopt this terminology and use the row-major representation.

Other variables which are members of classes are called obvious things like `sigfig`, `maxiter`, `blocksize` etc. No rules here.

2 Dot Products

A fast library for kernel functions [Vapnik, 1995, Wahba, 1990, Cristianini and Shawe-Taylor, 1999, Williams, 1998, Girosi et al., 1995, Mangasarian, 2000, Schölkopf and Smola, 2003] is at the heart of any any algorithm for kernel based learning. While the matrix multiplication routines in Matlab may not always lead to peak performance on all architectures, the recent releases (MATLAB 5.3 with a library update and also MATLAB 6) provide floating point power which is typically faster than C code which has not optimized for caching effects. This led to the conclusion that fast kernel functions under MATLAB are possible.

In order to allow users to modify existing algorithms easily for different kernel functions, the latter are represented as objects with their corresponding methods for computing dot products and making predictions. This means that in order to adapt the currently existing set of algorithms to a new problem all that needs to be done is to write an efficient dot product library for it, be it in MATLAB or C (such kernels can be used in conjunction with the new datatypes). Moreover, it allows the transparent use of a caching module on top of the actual kernels which automatically will benefit all algorithms (as far as they are affected by caching).

One of the benefits of the dot products in the library is that they work both on individual vectors (patterns) which in the following we will denote by x, and also on matrices of such vectors, where the x_i are arranged as **column vectors**.[4] Secondly, when computing

$$f(x_i) = \sum_{j=1}^{m} \alpha_i k(\bar{x}_j, x_i) \text{ for } 1 \leq i \leq n \tag{1}$$

it is wasteful (and usually impossible) to compute the full matrix $K_{ji} = k(\bar{x}_j, x_i)$ in memory beforehand and perform the matrix multiplication with $\alpha \in \mathbb{R}^m$ afterwards since this will require a large amount of memory. Hence, block-wise computation is needed. The `blocksize` member variable of all dot product classes handles exactly that. The default blocksize is set to 128.

The kernels implemented (polynomial, rbf, vanilla, tanh) in the library serve as basic templates for further kernels of type $k(x, x') = k(\|x - x'\|)$ and $k(x, x') = k(\langle x, x' \rangle)$. It is very easy to derive further kernel functions from these libraries simply by replacing the corresponding polynomial and exponential functions.

Kernels using invariances, such as the pyramidal kernels by Schölkopf [1997] will require further work and they are probably best implemented in C with additional glue linking the files to MATLAB. Section **??** contains an example of such a kernel function for which no fast MATLAB implementation can be given since it involves convolutions of the patterns on the pixel level.

[4]The latter is due to the fact that MATLAB uses column ordering of matrices as commonly used in FORTRAN, hence, transversal of a matrix in column-first order is fast and will not incur a large number of cache misses.

2.1 Classes and Constructors

2.1.1 vanilla_dot

Function Definition This class implements the simplest of all kernel functions, namely

$$k(x, x') = \langle x, x' \rangle. \tag{2}$$

Still, it may be useful, in particular when dealing with large sparse data vectors x, as is usually the case in text categorization.

Calling Convention

```
kernel = vanilla_dot(blocksize)
```

Member Variables

blocksize determines the size of the blocks used in (1).

2.1.2 poly_dot

Function Definition This class implements both homogeneous and inhomogeneous polynomial kernels via the following function

$$k(x, x') = \left(\text{scale} \cdot \langle x, x' \rangle + \text{offset}\right)^{\text{degree}}. \tag{3}$$

It is mainly used for classification on images such as handwritten digits [Schölkopf et al., 1997] and pictures of three dimensional objects [Schölkopf, 1997].

Calling Convention

```
kernel = poly_dot(degree, offset, scale, blocksize)
```

Default values are 1 for degree, offset, and scale, and 128 for the blocksize.

Member Variables

degree The degree of the polynomial. This variable has to be an integer. **Otherwise the kernel will return complex values** and it clearly will cease being a Mercer kernel Smola et al. [2001].

offset If the offset is set to 0, we obtain homogeneous polynomial kernels [Poggio, 1975], for positive values, we have inhomogeneous kernels. Note that for negative values the kernel does not satisfy Mercer's condition and thus the optimizers may fail.

scale The scaling parameter is a convenient way of normalizing patterns without the need to modify the data itself directly.

blocksize It determines the size of the blocks used in (1).

2.1.3 rbf_dot

Function Definition This class implements Gaussian radial basis function kernels via the following function

$$k(x, x') = \exp(-\sigma \|x - x'\|^2). \tag{4}$$

It is a general purpose kernel and is typically used when no further prior knowledge is available. **Note the sigma in the numerator.**

Calling Convention

```
kernel = rbf_dot(sigma, blocksize)
```

Default values are 1 for sigma and 128 for the blocksize.

Member Variables

sigma This is the inverse kernel width. Typically one would denote (4) by $\exp\left(-\frac{\|x-x'\|^2}{2\bar{\sigma}^2}\right)$. However, it is much more convenient (an numerically efficient since it avoids divisions) to define a kernel as in (4). Only strictly positive values are allowed.

blocksize It determines the size of the blocks used in (1) and also the blocking update in the dot product computations.

2.1.4 bessel_dot

Function Definition This class implements Bessel functions of the first kind via the following function

$$k(x, x') = \frac{\text{Bessel}^n_{(\nu+1)}(\sigma\|x - x'\|)}{(|x - x'\|)^{-n(\nu+1)}}. \tag{5}$$

It is a general purpose kernel and is typically used when no further prior knowledge is available and mainly popular in the Gaussian Process community.

Calling Convention

```
kernel = bessel_dot(sigma, nu, n, blocksize)
```

Default values are 1 for sigma, an automatic choice for ν, namely $\nu = \frac{d-1}{2}$, where d is the (effective) dimensionality of the data, and 128 for the blocksize.

Member Variables

sigma This is the inverse kernel width. Only strictly positive values are allowed.

nu The order of the Modified Bessel function of the first kind. Only adjust this if you know what you are doing!

n This is the power of the Bessel function

blocksize It determines the size of the blocks used in (1) and also the blocking update in the dot product computations.

2.1.5 tanh_dot

Function Definition This class implements hyperbolic tangent kernels via the following function

$$k(x, x') = \tanh\left(\text{scale} \cdot \langle x, x' \rangle + \text{offset}\right) \tag{6}$$

It is mainly used as a proxy for Neural Networks. Note that there is no guarantee (for any choice of parameters) that the resulting kernel matrix will be positive definite [**?**].

If you are not convinced, try the following:

```
kernel = tanh_dot;
x = randn(10, 100);
dpt = sv_dot(x, kernel);
min(eig(dpt))
```

Calling Convention

```
kernel = tanh_dot(offset, scale, blocksize)
```

Default values are 1 for offset and scale, and 128 for the blocksize.

Member Variables

offset If the offset is negative the likelihood of obtaining a kernel matrix that is not positive definite is much higher (since then even some diagonal elements may be negative), hence if this kernel has to be used, the offset should always be positive. Note, however, that this is no guarantee that the kernel will be positive.

scale The scaling parameter is a convenient way of normalizing patterns without the need to modify the data itself directly.

blocksize It determines the size of the blocks used in (1).

2.1.6 laplace_dot

Function Definition This class implements a Laplacian radial basis function kernels via the following function

$$k(x, x') = \exp(-\sigma\|x - x'\|). \tag{7}$$

It is a general purpose kernel and is typically used when no further prior knowledge is available.

All further particulars are identical with `rbf_dot` of Section 2.1.3.

2.2 Methods

As mentioned above the methods of the kernel library are written in such a way that they take both vectors (i.e. single patterns) and also matrices (i.e. vectors of patterns) as arguments. Moreover, in the case of symmetric matrices (e.g. the dot product matrix of a Support Vector Machine) they only require one argument rather than having to pass the same matrix twice (for rows and columns). The computations are vectorized whenever possible which guarantees good performance and acceptable memory requirements.

2.2.1 sv_dot

This is the most commonly used function. It computes $k(x, x')$, i.e. it computes the matrix K where $K_{ij} = k(x_i, x_j)$ and x is a **column** vector. In particular

```
K = sv_dot(kernel, X);
```

computes the matrix $K_{ij} = k(x_i, x_j)$ where the x_i are the columns of X and

```
K = sv_dot(kernel, X1, X2);
```

computes the matrix $K_{ij} = k(x1_i, x2_j)$. Both `X1` and `X2` may be matrices and this way of computing dot product matrices is highly preferred over looping through individual kernel computations. Hence, the above method (`K = sv_dot(kernel, X)`)is preferred over

```
for i=1:m; for j=1:m;
   K(i,j) = sv_dot(kernel, X(:,i), X(:,j));
end; end
```

2.2.2 sv_dot_fast

This method is only available for `rbf_dot`, `bessel_dot`, and `laplace_dot`, which are all RBF kernels. It is identical to `sv_dot`, except that it also requires the squared norm of the first argument as additional input.

It is mainly used in incomplete Cholesky decompositions, where columns of the kernel matrix are computed per invocation. In these cases, evaluating the norm of each row-entry over and over again would cause significant computational overhead, hence `sv_dot_fast`. Its invocation is via

```
K = sv_dot_fast(kernel, X1, X2, DOTX1)
```

Here `DOTX1` is a vector containing the squared norms of `X1`.

2.2.3 sv_pol

This method is very similar to `sv_dot` with the only difference that rather than computing $K_{ij} = k(x_i, x_j)$ it computes $K_{ij} = y_i y_j k(x_i, x_j)$. This means that

```
K = sv_pol(kernel, X, Y);
```

computes the matrix $K_{ij} = y_i y_j k(x_i, x_j)$ where the x_i are the columns of X and y_i are elements of the vector Y. Moreover

```
K = sv_pol(kernel, X1, X2, Y1, Y2);
```

computes the matrix $K_{ij} = k(x1_i, x2_j)$. Both `X1` and `X2` may be matrices and `Y1` and `Y2` vectors.

2.2.4 sv_mult

`sv_mult` is a convenient way of computing kernel expansions at many locations simultaneously. It returns the vector $(f(x_1), \ldots, f(x_m)$ where

$$f(x_i) = \sum_{i=1}^{m} \alpha_i k(x_i, x_j), \text{ hence } f = K\alpha. \tag{8}$$

The need for such a function arises from the fact that K may sometimes be larger than the memory available. Therefore it is convenient to compute K only in stripes and discard the latter after the corresponding part of $K\alpha$ has been computed. The variables `blocksize` determines the number of rows in the stripes. In particular

```
f = sv_mult(kernel, X, alpha)
```

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

```
f = sv_mult(kernel, X1, X2, alpha)
```

computes $f_i = \sum_{j=1}^{m} k(x1_i, x2_j)\alpha_j$.

3 Interior Point Codes

For general purpose quadratic programming, and also as the core engine to various chunking codes a robust QP engine is needed. We choose to use a primal-dual interior point path following method, as described by Vanderbei [1992] and Schölkopf and Smola [2002], with several modifications to cater for rank-deficient kernel matrices, i.e. where $K = ZZ^\top$ with $Z \in \mathbb{R}^{m\times n}$ and $n \ll m$.

3.1 Generic

3.1.1 intpoint

Function Definition The intpoint class implements the generic QP engine available in svlab which solves the following quadratic problem :

$$\begin{array}{ll} \text{minimize} & c^\top x + \frac{1}{2} x^\top H x \\ \text{subject to} & b \leq Ax \leq b + r \\ & l \leq x \leq u \end{array} \tag{9}$$

Calling Convention To initialize a optimizer object :

```
optimizer = intpoint(sigfig, maxiter, margin, bound)
```

Default values: sigfig = 7, maxiter = 50, margin = 0.05, bound = 10

Member Variables

sigfig the number of significant decimal digits

maxiter maximum number of iteration.

margin how close we want to get to the constrains. Note that this has no relation with the margin of a classifier

bound clipping bound for the variables. Should be adjusted on a per problem basis.

3.1.2 optimize

Function Definition optimize solves the quadratic problem and returns the primal, dual solution along with information on convergence.

Calling Convention `[primal, dual, how] = optimize(optimizer, c, H, A, b, r, l, u)`

Member Variables

optimizer an optimizer object

c vector appearing in the quadratic function.

H matrix appearing in the quadratic form. This can also be a non-square matrix Z or Hmn (where $H = Z^\top Z$ or $H = H_{mn}H_{nn}^{-1}H_{nm}$)in this case the Sherman Morrisonn Woodbury formula is used.

A matrix defining the constrains under which we minimize the quadratic function

b vector or number defining the constrains

r vector or number defining upper bound constrains

l lower bound on solution

u upper bound on solution

Hnn this is an optional parameter in case the $H = H_{mn}H_{nn}^{-1}H_{nm}$ low rank aproximation is used.

3.2 Pattern Recognition

3.2.1 intpoint_pr

Along with the QP engine we also provide a tailored version of the solver adapted to the support vector classification ($C - svm$,$nu - svm$) optimization problems. The following optimization problem, solved by this optimizer, is general enough to cover classification, regression, and novelty detection.

$$\begin{array}{ll} \text{minimize} & c^\top x + \frac{1}{2}x^\top Hx \\ \text{subject to} & Ax = b \\ & l \leq x \leq u \end{array} \tag{10}$$

Calling Convention To initialize a PR optimizer object :

```
optimizer = intpoint_pr(sigfig, maxiter, margin, bound)
```

Default values: sigfig = 7, maxiter = 50, margin = 0.05, bound = 10

Member Variables

sigfig the number of significant decimal digits
maxiter maximum number of iteration.
margin how close we want to get to the constrains. Note that this has no relation with the margin of a classifier
bound clipping bound for the variables. Should be adjusted on a per problem basis.

3.2.2 optimize

Optimize solve the pattern recognition QP problem and returns primal and dual solution along with convergence information.

```
[primal, dual, how] = optimize(optimizer, c, H, A, b, l, u)
```

Parameters

optimizer an optimizer object
c vector appearing in the quadratic function.
H matrix appearing in the quadratic form. This can also be a non-square matrix Z or Hmn (where $H = Z^\top Z$ or $H = H_{mn}H_{nn}^{-1}H_{nm}$)in this case the Sherman Morrisonn Woodbury formula is used.
A matrix defining the constrains under which we minimize the quadratic function
b vector or number defining the constrains
l lower bound on solution
u upper bound on solution
Hnn this is an optional parameter in case the $H = H_{mn}H_{nn}^{-1}H_{nm}$ low rank aproximation is used.

primal a vector containing the primal solution to the problem
dual the dual solution to the problem
how a character string describing the type of convergence

3.3 Regression

3.3.1 intpoint_re

The intpoint_re is a quadratic problem solver specially tailored to handle the optimization problem of support vector regression (see Schölkopf and Smola [2002] section 10.3.3 for details). It solves the following optimization problem :

$$\begin{array}{ll} \text{minimize} & c^\top x + \frac{1}{2} x^\top H x \\ \text{subject to} & Ax = b \\ & l \leq x \leq u \end{array} \tag{11}$$

where H has a special form: $H = \begin{matrix} (H2 + H1) & (H2 - H1) \\ (H2 - H1) & (H2 + H1) \end{matrix}$ $H1$is assumed to be positive semidefinite full$H$

(12)

Calling Convention

```
optimizer = intpoint_pr(sigfig, maxiter, margin, bound)
```

Default values: sigfig = 7, maxiter = 50, margin = 0.05, bound = 10

Member Variables

sigfig the number of significant decimal digits
maxiter maximum number of iteration.
margin how close we want to get to the constrains. Note that this has no relation with the margin of a classifier
bound clipping bound for the variables. Should be adjusted on a per problem basis.

3.3.2 optimize

optimize solves the QP returning the primal, dual solution along with convergence information.

Calling Convention `[primal, dual, how] = optimize(optimizer, c, H1, H2, A, b, l, u)`

Parameters
c vector appearing in the quadratic function.
H1 first part of the matrix appearing in the quadratic form.
H2 second part of the matrix appearing in the quadratic form. $H2$ is diagonal hence its passed as a vector.
A matrix defining the constrains under which we minimize the quadratic function
b vector or number defining the constrains
l lower bound on solution
u upper bound on solution

3.3.3 optimize_smw

Again we provide a version of the optimizer which only requires a low rank approximation of the $H1$ matrix $H1_{mn}$ and H_{nn}.

Calling Convention

```
[primal, dual, how] = optimize(optimizer, c, H1mn,H1nn, H2, A, b, l, u)
```

Parameters

c vector appearing in the quadratic function.

H1mn first part of the low rank approximation appearing in the quadratic form.

H1nn second part of the low rank approximation appearing in the quadratic form.

H2 second part of the matrix appearing in the quadratic form. $H2$ is diagonal hence its passed as a vector.

A matrix defining the constrains under which we minimize the quadratic function

b vector or number defining the constrains

l lower bound on solution

u upper bound on solution

4 Sequential Minimal Optimization

The Sequential Minimal Qptimazation algorithm (SMO)] solves the SVM quadratic problem by putting chunking to the extreme and iteratevly selecting subsets of size 2 for optimizating the target function with respect to them as described in **?**. The key point is that for a working set of 2 the optimization subproblem can be solved analytically without explicity invoking a quadratic optimizer. The amount of memory requiered for SMO is linear in the training set size, which allows SMO to handle very large training sets. Because large matrix computation is avoided SMO, scales somewhere between linear and quadratic in the training set size for various test problems.

4.1 Pattern Recognition

`svlab` includes a version of the SMO algorithm for classification with SVM. This version of the algorithm is tailored for the Support Vector Classification problem.

4.1.1 smo_pr

Creates an smo object for classification.

Calling Convention

```
smo = smo_pr(C, epsilon, tol, numchanged, examineall, verbose, counter, filename)
```

Default values: C = 1, epsilon = 1e-8, tol = 0.01, numchanged = 0, examineall = 1, verbose = 0, counter = 0, filename = 'smo_state'

Member Variables

C the SVM cost parameter
epsilon
tol tolerance of the termination criterion
verbose be verbose (default 0, no verbosity)
counter save algoritm state along with current results every counter steps (default 0, no saving done)
filename file to save results in (only relevant if counter is nonzero)

4.1.2 optimize

optimize solves the C-SVM QP problem for the classification problem given the data and the kernel and returns the α vector containing the solution, the offset b along with the training error.

Calling Convention `[alpha, b, error] = optimize(d, kernel, x, y)`

Parameters
d an smo_pr object
kernel a kernel object
x the input patterns
y the binary target values

Results
alpha the α vector containing the result
b the offset
error the training error

4.2 Regression

`svlab` includes a version of the SMO algorithm for regression with SVM. This version of the algorithm is tailored for the Support Vector regression problem.

4.2.1 smo_re

Creates an smo object for regression.

Calling Convention

```
smo = smo_pr(C, epsilon, tol, numchanged, examineall, verbose, counter, filename)
```

Default values: C = 1, epsilon = 1e-8, tol = 0.01, numchanged = 0, examineall = 1, verbose = 0, counter = 0, filename = 'smo_state'

Member Variables
C the SVM cost parameter
eps the SVM ϵ parameter
epsilon
tol tolerance of the termination criterion
verbose be verbose (default 0, no verbosity)
counter save algoritm state along with current results every counter steps (default 0, no saving done)
filename file to save results in (only relevant if counter is nonzero)

4.2.2 optimize

optimize solves the ϵ-SVM QP problem for the regression problem given the data and the kernel and returns the α vector containing the solution, the offset b along with the training error.

Calling Convention `[alpha, b, error] = optimize(d, kernel, x, y)`

Parameters
d an smo_pr object
kernel a kernel object
x the input patterns
y the target values
Results
alpha the α vector containing the result
b the offset
error the training error

4.2.3 smo_re

4.2.4 optimize

5 On-line Algorithms

The on-line algorithms which are included in `svlab` are all part of the NORMA family of on-line algorithms Kivinen et al. [2003], Schölkopf and Smola [2002] section 10.6 .

The nature of on-line algorithms requires a different aproach compared to algorithms that run in batch setting. The algorithm takes one pattern at the time and tries to optimize a cumulative loss function. The algorithms implemented here are classical stochastic gradient descent with respect to the instantaneous risk which is an aproximation of the regularized risk. `svlab` includes implementations of the algorithm for classification regression and novelty detection.

The on-line algorithms are constracted in a sligtly different way than the batch algorithms. An object is used recursevly to store the current state of the algorithm along with the current set of parameters, the object is returned after each "learned" pattern and has to be passed to the train function again along with the next pattern.

5.1 Pattern Recognition

The online algorithm for binary classification uses the soft margin loss function and returns a online_pr object along with an estimatin of f for that pattern. A predict function is also included for patterns without class labels.

5.1.1 online_pr

Creates an online object for classification.

Calling Convention

```
on = online_pr(bufsize = 1000)
```

Default values: bufsize = 1000

Member Variables

bufsize the size of the buffer containing the model parameters.

5.1.2 train

The train function takes a single pattern along with its label and the parameters of the algorithm and trys to minimize the soft margin loss function returning the f value along with a online_pr object.

Calling Convention

```
[phi, on] = train(on, k, x, y, lambda, nu)
```

Parameters

on an online_pr object
kernel a kernel object
x a single input pattern
y the target value (binary -1 or 1)
lambda the learning rate for the stochastic gradient algorithm

nu the ν parameter (similar to the *nu* parameter in the ν-SVM)
Results
phi the f value of the estimated function sign(f) giving the class the patternn belongs
on the online_pr object containing

5.1.3 predict

The train function takes a single pattern along with the kernel object and a trained online_pr object and returns the estimated function f giving the class of the pattern.

Calling Convention

```
phi = predict(on, k, x)
```

Parameters
on an online_pr object
kernel a kernel object
x the input pattern
Results
phi the f value of the estimated function, sign(f) giving the class the pattern belongs

5.2 Regression

The online algorithm for regression uses Huber's robust loss function and returns a online_re object along with an estimatin of f for that pattern.

5.2.1 online_re

Creates an online object for regression.

Calling Convention

```
on = online_pr(bufsize = 1000)
```

Default values: bufsize = 1000
Member Variables

bufsize the size of the buffer containing the model parameters.

5.2.2 train

The train function takes a single pattern along with its label and the parameters of the algorithm and trys to minimize the soft margin loss function returning the f value along with a online_re object.
Calling Convention

```
[phi, on] = train(on, k, x, y, lambda, nu)
```

Parameters

on an online_re object
kernel a kernel object
x a single input pattern
y the target value
lambda the learning rate for the stochastic gradient algorithm
nu the ν parameter (similar to the *nu* parameter in the ν-SVR)
Results
phi the f value of the estimated function
on the online_re object containing

5.3 Novelty Detection

The online algorithm for novelty detection returns a online_novelty object along with an estimate of the decision function f for that pattern, a negative f marks a "novel" pattern.

5.3.1 online_pr

Creates an online object for classification.

Calling Convention

```
on = online_novelty(bufsize = 1000)
```

Default values: bufsize = 1000
Member Variables

bufsize the size of the buffer containing the model parameters.

5.3.2 train

The train function takes a single pattern along with its label and the parameters of the algorithm and tries to minimize the loss function returning the f value along with a online_novelty object.
Calling Convention

```
[phi, on] = train(on, k, x, lambda, nu)
```

Parameters
on an online_novelty object
kernel a kernel object
x a single input pattern
lambda the learning rate for the stochastic gradient algorithm
nu the ν parameter (similar to the *nu* parameter in the ν-SVM)
Results
phi the f value of the estimated function sign(f) giving the class the patternn belongs
on the online_novelty object containing

6 Lagrangian SVM

The Lagrangian support vector machine] is a particularly simple learning algorithm which deals with classification problems involving squared slacks. Compared to the clasical SVM algorithms with the margin it aslo regularizes the constant offset β thus loosing translation invariance in feature space. It still an open question if this has detrimental effect in generalization ability.

6.1 Pattern Recognition

We include an implmentation of lsvm for classification using an incomplete cholesky decomposition.

6.1.1 lsvm

Creates an `lsvm` object for classification.

Calling Convention

```
lvm = lsvm(kernel, problemtype, nu, lambda, tol, itmax)
```

Default values: kernel = rbf_dot, problemtype = “classification”, nu = 0.1, lambda = 0.3, tol = 0.0001, itmax = 400

Member Variables

kernel the kernel to be used
problemtype currently only classification is supported
nu
lambda
tol the tolerance of the convergence criterium
itmax the number of maximum iterations

6.1.2 train

Calling Convention

```
d = train(lsvm, x, y)
```

Parameters
lsvm an online_novelty object
x the input patterns on which to train the `lsvm`
y the learning rate for the stochastic gradient algorithm
Results
d a `lsvm` object containing the model parameters

6.1.3 predict

Calling Convention `y = predict(l,x)`
l a trained `lsvm` object
x the input patterns
Results
y the input patterns label

7 Feature Extraction

7.1 Kernel PCA

This class implements Kernel Principal Component Analysis, as described in Ch 16 of [?]. Briefly, it uses a kernel to map the training data patterns into feature space, then computes the eigenvectors or "features" in of the feature space representation of the training patterns. The eigenvectors are described by their coefficients using the training pattern set as a basis. A test data set can then be analysed in terms of its components under the eigenvector basis.

7.1.1 kpca

Function Definition This is the constructor for the `kpca` object.
Calling Convention

```
kpca_var = kpca();
```

is the basic constructor, which defaults to extracting all the eigenvectors of the training data set mapped into feature space.

```
kpca_var = kpca(verbose, numfeatures);
```

This variation sets the `kpca` object so that only the eigenvectors corresponding to the `numfeatures` biggest eigenvectors are returned.
Member Variables

verbose Set to 0 by default, for no reports. No further verbosity implemented yet.
numfeatures Set to 0 by default. If set to 0, algorithms will extract the full set of features.

7.1.2 train

Function Definition Train computes the eigensystem of the Gram matrix of the training patterns, after it has centered the patterns in feature space.
Calling Convention

```
[e_values, e_vectors, offset] = train(kpca_var,ker, X);
```

Here `kpca_var` is the kpca object you created using the `kpca` method, `ker` is a kernel object, and `X` is a matrix containing the training patterns, with the individual vectors arranged in columns. The output `e_values` is a column of eigenvalues, `e_vectors` is a column-major matrix of eigenvectors (the size of these matrices will depend on the dimension of your input patterns and the value of `kpca.numfeatures`). The vector `offset` is used by the `extract` method to compensate for the fact that the input set may not be centered in feature space. In the terminology of [?] equation 20.32, $offset = 1'_M K - 1'_M K 1_M$. The user need not worry about these details, merely ensuring that `offset` is passed to the `extract` method.

7.1.3 extract

Function Definition Extract computes the eigenvector components of a test data set, after normalizing it in feature space according to the center of mass of the training data set.

Calling Convention

```
[F] = extract(kpca_var, ker, X, T, offset, e_vecs);
```

where F_{ij} is the ith eigenvector component of the jth normalized test pattern, where T is a matrix containing as columns the test patterns.

7.2 Kernel Feature Analysis

7.2.1 kfa

7.2.2 train

7.2.3 quad_contrast

quadratic contrast function

7.2.4 curt_contrast

curtosis contrast

7.3 Sparse Greedy Matrix Approximation

This module is based on the algorithm described in Chapter XX of [?] and in [?]. Basically, it is similar to Kernel PCA, but instead of extracting eigenvectors of the training dataset in feature space, it approximates the eigenvectors by selecting training patterns which are good basis vectors for the training set. Let us call this set of patterns $\bar{X}$. the algorithm works as follows: it chooses a fixed size subset of X, scales it to unit length (under the kernel), and then chooses the one which, when dotted with all the other vectors in $\{X - \bar{X}\}$, gives the largest result. It optimizes the search for these vectors by using several tricks described in Problems XX.X and XX.X of [?], namely the Cholesky decomposition of K, the rank-1 update, and the caching of only T and $T^{-1}(K^{mn})^\top$ instead of K, K^{-1} and K^{mn}.

7.3.1 sgma

Function Definition This is the constructor for the `sgma` object.

Calling Convention

```
sgma_var = sgma()
```

returns an sgma object with default properties listed below.

```
sgma_var = sgma(verbose,sigfig,maxiter,subsetsize,errorbound,blocksize)
```

returns an sgma object with the member variables initialized appropriately.

Member Variables

verbose If verbosity is 0 then there are no reports, 1 gives a report at the beginning and the end of the algorithm, and 2 gives a report after each iteration of the main loop, i.e. after each new vector in X has been selected.

sigfig The number of significant figures to use. Currently not implemented, defaults to 7.

maxiter The maximum number of times to run through the loop, i.e. the maximum number of vectors in X to select. Defaults to 100.

subsetsize The size of the subset of vectors in X to search for the best vector. Defaults to 59, a value which seems to work well.

errorbound Ranging between 0 and 1, this quantity is equal to $\left(1 - \frac{tr(K^{m,n}(K^{nn})^{-1}K^{nm})}{tr(K)}\right)$. If $\bar{X}$ is the subset of X selected as the basis, then $K_{i,j}^{m,n+1} = k(x_i, \bar{x}_j)$. Therefore `error` represents the difference in energy between X and the projection of X onto $\bar{X}$. If `error` is selected close to zero, the algorithm will iterate many times, whereas if it is set close to 1, it will iterate fewer times but give a poorer basis for X. Defaults to 0.01.

blocksize This is the same as the kernel function's blocksize. In fact, if set to zero (the default) the kernel will use its own blocksize, otherwise the kernel will use the blocksize specified here. It is a good idea to use a blocksize which is a power of 2.

7.3.2 train

Function Definition This actually performs the search for the subset of X with the best span. It continues searching until all vectors in X have been selected, `maxiter` vectors have been selected, or the error described above is $\leq$ errorbound.

Calling Convention `[T,basisvecsindex] = train(sgma_var, ker, X)`
Here `basisvecsindex` is an array of indices into X, for example if $basisvecsindex[2] = 5$, then the third vector selected by the algorithm is the 5th column of X. Here `T` refers not to a set of test patterns, but to the Cholesky decomposition of $\bar{K}$, i.e. $\bar{K} = TT^{-1}$.

8 Utilities and Notes for Developers

explain the read_token, loadsn, savesn functionality etc.

9 Summary and Future Work

all we didn't have time to do but thought it would be a great idea ...

Acknowledgments AS is supported by a grant of the Deutsche Forschungsgemeinschaft (SM 62/1-1). BL is supported by a grant of the Australian Research Council.

References

C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

N. Cristianini and J. Shawe-Taylor. Bayesian voting schemes and large margin classifiers. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods—Support Vector Learning*, pages 55–68, Cambridge, MA, 1999. MIT Press.

N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, Cambridge, UK, 2000.

T.-T. Frieß, N. Cristianini, and C. Campbell. The kernel adatron algorithm: A fast and simple learning procedure for support vector machines. In J. Shavlik, editor, *Proceedings of the International Conference on Machine Learning*, pages 188–196. Morgan Kaufmann Publishers, 1998.

F. Girosi, M. Jones, and T. Poggio. Regularization theory and neural networks architectures. *Neural Computation*, 7(2):219–269, 1995.

R. Herbrich. *Learning Kernel Classifiers: Theory and Algorithms*. MIT Press, 2002.

T. Joachims. *Learning to Classify Text Using Support Vector Machines: Methods, Theory, and Algorithms*. The Kluwer International Series In Engineering And Computer Science. Kluwer Academic Publishers, Boston, May 2002. ISBN 0-7923-7679-X.

J. Kivinen, A. J. Smola, and R. C. Williamson. Online learning with kernels. *IEEE Transactions on Signal Processing*, 2003. forthcoming.

O. L. Mangasarian. Generalized support vector machines. In A. J. Smola, P. L. Bartlett, B. Schölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 135–146, Cambridge, MA, 2000. MIT Press.

K.-R. Müller, S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf. An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 12(2):181–201, 2001.

T. Poggio. On optimal nonlinear associative recall. *Biological Cybernetics*, 19: 201–209, 1975.

B. Schölkopf. *Support Vector Learning*. R. Oldenbourg Verlag, München, 1997. Doktorarbeit, TU Berlin. Download: http://www.kernel-machines.org.

B. Schölkopf and A. J. Smola. *Learning with Kernels*. MIT Press, 2002.

B. Schölkopf and A. J. Smola. Support vector machines. In M. A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 1119–1125. MIT Press, 2nd edition, 2003.

B. Schölkopf, K. Sung, C. Burges, F. Girosi, P. Niyogi, T. Poggio, and V. Vapnik. Comparing support vector machines with Gaussian kernels to radial basis function classifiers. *IEEE Transactions on Signal Processing*, 45:2758–2765, 1997.

A. J. Smola, Z. L. Óvári, and R. C. Williamson. Regularization with dot-product kernels. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 308–314. MIT Press, 2001.

A. J. Smola and B. Schölkopf. A tutorial on support vector regression. NeuroCOLT Technical Report NC-TR-98-030, Royal Holloway College, University of London, UK, 1998.

R. J. Vanderbei. Primal-dual symmetric formulations of the predictor-corrector method for QP. Talk held at the fourth siam conference on optimization in chicago, il, usa, Dept. of Civil Engineering and Operations Research, Princeton University, Princeton, NJ 08544, USA, May 1992.

V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, New York, 1998.

G. Wahba. *Spline Models for Observational Data*, volume 59 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 1990.

C. K. I. Williams. Prediction with Gaussian processes: From linear regression to linear prediction and beyond. In M. I. Jordan, editor, *Learning and Inference in Graphical Models*, pages 599–621. Kluwer Academic, 1998.

10 Contributions and Copyright Owners

10.1 Copyright Notice

10.2 Contributors

Below is a list of people and institutions who have contributed to this code.

- Jyrki Kivinen, The Australian National University
- Ben O'Loghlin, The Australian National University
- Gunnar Rätsch, GMD FIRST
- Bernhard Schölkopf, GMD FIRST and Microsoft Research
- Alexandros Karatzoglou, TU Wien
- Alex Smola, GMD FIRST and The Australian National University
- Paul Wankadia, The Australian National University

10.3 Clarification of the GNU Public License

Since the GNU Public License has been written with computer programs in mind, yet the SVLAB library is both a program and a scientific work, we would like to ask researchers who derive scientific work based on the SVLAB library to **acknowledge the use of this library in their publications**. This should not pose a problem with the GNU public license. Moreover this is in agreement with the common practice in academic environments to give credit for scientific work.

10.4 GNU Public License

[5]See also Section 10.2 for a full list of contributors.

Terms and Conditions for Copying, Distribution and Modification

permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.