Practical session: Using user-defined kernels

Jean-Philippe Vert

In this session you will

- Learn how to use precomputed kernels
- Learn how to define your own kernel function

Sometimes you want to use a kernel that is not already implemented in kernlab, e.g., because you use a specific kernel that you can implement in R or is obtained from another problem. It is then useful to be able to implement your own kernel or directly use a kernel matrix instead of x to train and test a SVM.

1 Using precomputed kernels

Suppose you have your own $n \times n$ Gram matrix $K$, and a label matrix $y$. Then you can train a SVM by the command

```r
m <- ksvm(as.kernelMatrix(K),y,type="C-svc",kernel='matrix')
```

**Question 1** Check on a toy dataset that the using a precomputed kernel gives the same result as using the normal formulation, for a Gaussian or linear kernel.

To make predictions with precomputed kernel, you need to be a bit careful. You can use the following command:

```r
ypred <- predict(m,as.kernelMatrix(testK))
```

where `testK` is the kernel matrix between the test points (in rows) and the support vectors (in columns).

**Question 2** Split the toy dataset in a training set (80%) and a test set (20%). Train a SVM with precomputed Gaussian kernel on the training set, predict the labels on the test set. Compare the predictions with the true label.

**Question 3** Write a function `cv.precomp.ksvm(K,y,folds=3,...)` which returns a vector `ypred` of predicted decision scores for all points by $k$-fold cross-validation.

2 User-defined kernel function

kernlab offers the possibility to define kernel functions by yourself. This may be preferable to precomputed kernels, since often we do not need to know the full Gram matrix to train a SVM.

Creating a kernel means creating an object of class `kernel`, which is basically a function with an additional slot to hold kernel parameters. By default, in kernlab we construct such objects by calling functions like `rbfdot` or `vanilladot`. For example:

```r
# Create a RBF kernel
rbf <- rbfdot(sigma=1)

# Look at what it contains
```
Once we have a kernel (like the object \texttt{rbf} here), we can do several things:

\begin{verbatim}
# Compute kernel between two vectors
rbf(x[1,], x[2,])

# Compute the kernel matrix
K <- kernelMatrix(rbf, x[1:5,], x[6:n,])
dim(K)
K <- kernelMatrix(rbf, x)
dim(K)

# Train a SVM
m <- ksvm(x, y, kernel=rbf, scale=c())

# Look at the points with kernel PCA
kpc <- kpca(x, kernel=rbf, scale=c())
plot(rotated(kpc), col=ifelse(y>0,1,2))
\end{verbatim}

To define our own kernel functions, we can just do the same: define a function of class \texttt{kernel} in the correct format, following the examples of \texttt{rbfdot} or \texttt{vanilladot}.

\textbf{Question 4} Implement your own linear kernel and check that it does the same as the original one.

\textbf{Question 5} Implement the kernel $K(x, x') = \sum_{i=1}^{P} \min(x_i, x'_i)$, for $x, y \geq 0$, and test it on the toy dataset.

\textbf{Question 6} Implement a function \texttt{precomp <- function(K=matrix())} which takes as input a square kernel Gram matrix $K$ and creates a kernel function such that \texttt{precomp(i,j) = K[i,j]}. Show that it is equivalent to using $K$ as a precomputed kernel, although with a different syntax. Train a SVM with this formulation and check that the results are coherent with the use of precomputed kernels.