Support vector machines

STAT30270 – Statistical Machine Learning

Contents

1 Kernels **1**

2 Support vector machines **2**
2.1 A toy data example 2
2.2 Task 4
2.3 Tuning of SVM 4
2.4 Task 6

1 Kernels

Kernel functions can be used to map data into high-dimensional spaces with particular characteristics. The main functionalities to operate with kernels in R are available in the package `kernlab`. The package vignette contains a lot of information and technical details about the package and the methods, and it can be accessed via the command `vignette("kernlab")`. Take some time to read the documentation of the package.

Here we will see how to compute some basic kernels. First, we compute the simple linear kernel for two bi-dimensional vectors `x` and `y`.

```
# load the package
library(kernlab)

x <- c(1, 2)
y <- c(3, 4)

kern <- vanilladot()   # the 'vanilla' kernel corresponds to the inner product

kern(x, y)
t(x) %*% y
```

Suppose we have a data matrix x. Then we can use the function `kernelMatrix` to compute the kernel between observations (which are rows in a data matrix). The function takes in input the data matrix and the kernel function.

```
# create matrix of 5 observations on 2 dimensions
x <- matrix( rnorm(5*2), 5, 2 )

# compute kernel for each observation
kernelMatrix(x, kernel = kern)
#
# equivalent to the matrix
x %*% t(x)
```

Note that by applying the kernel we mapped our data from the original 2-dimensional space to a space with 5 dimensions. Different kernel functions can be considered, see `?dots` for a complete list and details.

```r
# polynomial kernel with offset
kern <- polydot(degree = 2, offset = 1)
kernelMatrix(x, kernel = kern)
```

2 Support vector machines

Support vector machines (SVM) are an effective family of supervised classifier for data with complex non-linear structures. SVM can be implemented in R using the functionalities of the package `kernlab`. In addition to the available vignette, the document at the following link provides extensive information about SVM in R. Take some time to read this document.

https://www.jstatsoft.org/article/view/v015i09

SVM are implemented using function `ksvm`. The function allows to implement different kernel-based support vector classifiers for different purposes, giving the option of selecting among different kernels and setting hyperparameters - See `?ksvm`.

2.1 A toy data example

We consider application of SVM to investigate the effect of different kernel choices and hyperparameters on the inferred decision boundaries. For this purpose, we will look at a classic dataset for testing classifiers with non-linear boundaries available in the file `data_toy_svm.txt` (see paper here https://link.springer.com/chapter/10.1007/11590316_1). We load the data and plot it. Note that we standardize the input feature data to have mean 0 and variance 1 - This is a standard procedure when working with SVM (and it is actually done by default in `ksvm`).

```r
data <- read.table("data_toy_svm.txt", header = TRUE)

x <- scale( as.matrix(data[,1:2]) )
y <- data[,3]

pch = c(19, 17)
cols <- c("forestgreen", "purple3")
plot(x, pch = pch[y], col = adjustcolor(cols[y], 0.7))
```

We now use the function `ksvm` to fit different SVM specified using various kernels. The function takes in input the matrix of predictors and the vector of class labels. The type of kernel can be specified using the argument `kernel`, while the argument `kpar` allows to input in a list the hyperparameters of the kernel. The argument `type` specifies for which task we use the SVM, in this case we set it to `C-svc` for soft-margin classification. The cost C is controlled through the same argument `C`, which by default is set to 1. See `?ksvm` for more info.

```r
# polynomial - degree 2
svm_poly_1 <- ksvm(x, y, type = "C-svc",
                   kernel = "polydot", kpar = list(degree = 2))

# polynomial - degree 3
svm_poly_2 <- ksvm(x, y, type = "C-svc",
                   kernel = "polydot", kpar = list(degree = 3))

# GRBF - sigma 0.5
svm_grbf_1 <- ksvm(x, y, type = "C-svc",
                   kernel = "rbfdot", kpar = list(sigma = 0.5))

# GRBF - sigma 1.5
svm_grbf_2 <- ksvm(x, y, type = "C-svc",
                   kernel = "rbfdot", kpar = list(sigma = 1.5))
```

```
# LRBF - sigma set via heuristic - see '?sigest'
svm_lrbf <- ksvm(x, y, type = "C-svc",
                 kernel = "rbfdot", kpar = "automatic")

# sigmoid - scale = 0.5
svm_tanh <- ksvm(x, y, type = "C-svc",
                 kernel = "tanhdot", kpar = list(scale = 0.5))
```

By calling the `ksvm` object, some useful information is printed on the console, including number of support vectors and training predictive performance. For example:

```
svm_poly_1
svm_grbf_1
```

More in general, the output from `ksvm` contains a lot of useful information. The object in output is of class `S4`, take some time to explore the slots of a few of the trained SVM models with the aid of the documentation in `?ksvm-class`. For example, the slot `xmatrix` contains the coordinates in the input space identifying the support vectors.

We can look at the data points defining the support vectors. We now visualize the decision boundaries of the different SVM. To do so, we create a function which takes in input a `ksvm` object. The function creates a grid of values over the input feature space and computes the values of the fitted SVM classifier, the output values are then used to produce a contour plot. We include the contour levels corresponding to 0, -1, and +1 (the separating hyperplane and the support vectors in the kernel feature space). The coordinates identifying the support vectors are highlighted.

```
plot_svm <- function(svm, x, grid_lenght = 100,
                     pal = function(n) hcl.colors(n, palette = "BrBg"),
                     pch = c(19, 17),
                     level = c(-1, 0, 1),
                     show_support_vectors = TRUE, ...)
{
  x <- as.matrix(x)
  y <- svm@ymatrix

  # set ranges for contour plot and crate grid of values
  r1 <- range(x[,1]) + c(-0.1, 0.1)
  r2 <- range(x[,2]) + c(-0.1, 0.1)
  xx <- seq(r1[1], r1[2], length = grid_lenght)
  yy <- seq(r2[1], r2[2], length = grid_lenght)
  grid <- expand.grid(xx, yy)

  # obtain values of the SVM classifier over grid values
  pred <- predict(svm, newdata = grid, type = "decision")

  # produce contour plot
  z <- matrix(pred, nrow = grid_lenght)
  filled.contour(xx, yy, z, color.palette = pal,
                 plot.axes = {
                   points(x, col = adjustcolor("black", 0.7), pch = pch[y])
                   if ( show_support_vectors ) points(svm@xmatrix[[1]][,1], svm@xmatrix[[1]][,2],
                                                      col = "#ff4d5c", pch = 0, cex = 1)
                   contour(xx, yy, z = z,
                           levels = level, add = TRUE, lty = 2, col = "black")
                 }, ...)
}
```

```
# plots
par(mar = c(0, 3, 3, 5))  # for margins (you may want to change these)
```

```r
plot_svm(svm_poly_1, x, main = "Poly 2")
plot_svm(svm_poly_2, x, main = "Poly 3")
plot_svm(svm_grbf_1, x, main = "GRBF 0.5")
plot_svm(svm_grbf_2, x, main = "GRBF 2")
plot_svm(svm_lrbf, x, main = "LRBF")
plot_svm(svm_tanh, x, main = "Sigmoid")
```

2.2 Task

- Explore the content of an object of class `"ksvm"` in output from the function `ksvm`, also with the aid of the documentation for `ksvm-class`. How can you extract the support vectors? Print and plot the support vectors for the SVM classifiers above.
- Read the documentation of `ksvm` function. The argument `prob.model = TRUE` (in conjunction with predict) allows to extract posterior class probabilities from the SVM classifier. Extract these probabilities from the different SVMs and compare and inspect them.
- What differences do you notice between the decision boundaries of the different kernels?
- Investigate the effect of the σ hyperparameter, the degree, and the cost C on the decision boundaries for the GRBF, the LRBF and the polynomial kernels.
- The function `fitted` can be used to extract the predicted values. What is the training predictive performance of the different kernels? Which one do you think performs better?

2.3 Tuning of SVM

The performance of support vector machine classifiers can be sensitive to the choice of kernel function and hyperparameters (like the cost C, the degree, and the scaling coefficient). Selection of the appropriate hyperparameters and kernels is in practice a model selection problem, since different SVM are defined according to different choices of hyperpameters and settings, therefore cross validation can be employed. In machine learning, choosing the hyperparameter combinations which leads to the best performance for a fixed class of models is denoted *tuning*.

In this section we implement a k-fold cross validation procedure to tune the cost C and the coefficient σ of a SVM with GRBF kernel (note that we could also use other alternative cross-validation procedures). We consider the `spam` dataset available in the `kernlab` package.

We set aside a portion of the data (20%) that will be used to test the final selected model. The remaining portion of the data is used as the base data for training and validation within th K-fold cross-validation procedure. To compare the models, we consider a 4-fold cross-validation (a 75%/25% split between training and validation set at each step of the procedure). We will replicate the cross-validation procedure $R = 10$ times.

```r
# load the data and prepare
data(spam)

# with SVM is always better to standardize the data
x <- spam[,-58]
y <- as.numeric(spam$type)

# set aside test data
N <- nrow(x)
set.seed(2025)
test <- sample(1:N, N*0.2)
x_test <- x[test,]
y_test <- y[test]

# select data for training and validation
train <- setdiff(1:N, test)
```

```r
x <- x[train,]
y <- y[train]
N_train <- nrow(x)

# we use this function to compute classification accuracy
class_acc <- function(y, yhat) {
  tab <- table(y, yhat)
  return( sum(diag(tab))/sum(tab) )
}
```

We consider a set of values for the cost equal to $C = \{1, 2, 5, 10, 20\}$, while for the hyperparameter of the GRBF kernel we consider the set of values $\sigma = \{0.010, 0.015, 0.020, 0.025, 0.030\}$. Since we want to check how the SVM performs for various combinations of C and σ, we create a grid with all possible combinations between these values.

```r
C <- c(1, 2, 5, 10, 20)
sigma <- c(0.010, 0.015, 0.020, 0.025, 0.030)
grid <- expand.grid(C, sigma)
colnames(grid) <- c("C", "sigma")
#
# check
head(grid, 10)

# total size of the grid
n_mod <- nrow(grid)
```

We now implement the cross-validation procedure. For each replication, we generate training and validation folds, then fitting the SVM for the different combinations of hyperparameters (C, σ) present in the grid. When using SVMs, is always good practice to standardize the data; we do so taking care of avoiding data leakage between validation and training set at each iteration of the cross-validation process.

```r
K <- 4      # set number of folds
R <- 10     # set number of replicates -- you could reduce this if too slow

out <- vector("list", R)    # store accuracy output
# out is a list, each slot of this list will contain a matrix where each column
# corresponds to the accuracy of each AVM classifier in the K folds

# running this cross-validation may take some time
# go out for a walk, or read a book, or catch up with your favorite tv series...
for ( r in 1:R ) {

  acc <- matrix(NA, K, n_mod)        # accuracy of the classifiers in the K folds

  folds <- rep( 1:K, ceiling(N_train/K) )
  folds <- sample(folds)             # random permute
  folds <- folds[1:N_train]          # ensure we got N_train data points

  for ( k in 1:K ) {
    train_fold <- which(folds != k)
    validation <- setdiff(1:N_train, train_fold)

    # standardize
    x_train_fold_sc <- scale( x[train_fold,] )
    x_val_sc <- scale(x[validation,], center = attr(x_train_fold_sc, "scaled:center"),
                                scale = attr(x_train_fold_sc, "scaled:scale"))
```

```
    # fit SVM on the training data and assess on the validation set
    for ( j in 1:n_mod ) {
      fit <- ksvm(x_train_fold_sc, y[train_fold], type = "C-svc",
                  kernel = "rbfdot",
                  C = grid$C[j], kpar = list(sigma = grid$sigma[j]))

      pred <- predict(fit, newdata = x_val_sc)
      acc[k,j] <- class_acc(pred, y[validation])
    }
  }

  out[[r]] <- acc
  # print(r)   # uncomment to print iteration number
}
```

The object `out` is a list where the slots are the replications of the cross-validation procedure. Each slot contains the accuracy of the SVM classifiers for different combinations of C and σ across the folds. For example, `out[[7]]` contains the output of the 7th replication of the procedure, with the values corresponding to the accuracy over the different combinations of hyperparameters.

```
out[[7]]
```

We can calculate the average fold accuracy for each one of the SVM (C, σ) combinations in all replications.

```
avg_fold_acc <- t( sapply(out, colMeans) )
head(avg_fold_acc, 3)
```

We can calculate the mean classification accuracy for each combination of C and σ over all the replications.

```
avg_acc <- colMeans(avg_fold_acc)      # estimated mean accuracy
grid_acc <- cbind(grid, avg_acc)
grid_acc
```

The optimal (C, σ) combination (**among those considered!**) is the one corresponding to the maximum mean classification accuracy.

```
best <- which.max(grid_acc$avg_acc)
grid_acc[best,]
```

2.4 Task

- Plot the average accuracy as a function of the cost hyperparameter and of the sigma hyperparameter in order to explore their effect on the validation accuracy. What is the effect of increasing C and σ on the validation data classification performance in general?
- How many classifiers have been trained in total?
- What are the limitations of using such grid search approach to tune the hyperparameters? What would you do if one of the selected hyperparameters corresponded to one of the boundaries of the grid (for example `C = 20` and/or `sigma = 0.010`)?
- What is the test data predictive performance of the SVM corresponding to the best hyperparameter (C, σ) combination? Note: You can train the selected model on the full data used for training + validation and apply it on the test data (remember to standardize the data appropriately …).
- Re-run this cross-validation by tuning the SVM using area-under-the-curve metrics (remember that you need to extract posterior class probabilities in this case).