Support Vector Machines and Gaussian Processes

ACM

2023-10-10

Support Vector Machines

Application to Gene Expression Data

The `Khan` dataset from the `ISLR` package is a high-dimensional dataset from a study that involves classifying cancer patients based on gene expression levels.

```
library(ISLR)
library(e1071)
?Khan

# Check the structure of the dataset
str(Khan)
```

```
## List of 4
##  $ xtrain: num [1:63, 1:2308] 0.7733 -0.0782 -0.0845 0.9656 0.0757 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:63] "V1" "V2" "V3" "V4" ...
##   .. ..$ : NULL
##  $ xtest : num [1:20, 1:2308] 0.14 1.164 0.841 0.685 -1.956 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:20] "V1" "V2" "V4" "V6" ...
##   .. ..$ : NULL
##  $ ytrain: num [1:63] 2 2 2 2 2 2 2 2 2 2 ...
##  $ ytest : num [1:20] 3 2 4 2 1 3 4 2 3 1 ...
```

```
dim(Khan$xtest)
```

```
## [1]   20 2308
```

```
dim(Khan$xtrain)
```

```
## [1]   63 2308
```

```
length(Khan$ytest)
```

```
## [1] 20
```

```
length(Khan$ytrain)
```

```
## [1] 63
```

```
table(Khan$ytest)
```

```
##
## 1 2 3 4
## 3 6 6 5
```

```
table(Khan$ytrain)
```

```
##
##  1  2  3  4
##  8 23 12 20
```

```
dat = data.frame(x=Khan$xtrain,y=as.factor(Khan$ytrain))
```

Now we'll fit a standard svm model with a linear kernel and fixed cost.

```
out = svm(y~.,data=dat,kernel="linear",cost=10)
summary(out)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##
## Number of Support Vectors:  58
##
##  ( 20 20 11 7 )
##
##
## Number of Classes:  4
##
## Levels:
##  1 2 3 4
```

```
table(out$fitted,dat$y)
```

```
##
##      1  2  3  4
##   1  8  0  0  0
##   2  0 23  0  0
##   3  0  0 12  0
##   4  0  0  0 20
```

```
dat.ts = data.frame(x=Khan$xtest,y=as.factor(Khan$ytest))
pred = predict(out,newdata=dat.ts)
table(pred,dat.ts$y)
```

```
##
## pred 1 2 3 4
##    1 3 0 0 0
##    2 0 6 2 0
##    3 0 0 4 0
##    4 0 0 0 5
```

Another option is to use the `rminer` package, which uses functions in other packages such as `kernlab` and `miner` to fit various machine learning methods. See https://repositorium.sdum.uminho.pt/bitstream/1822/36210/1/rminer-tutorial.pdf for details on the functionality of `rminer`.

An asset of this package is that it has a nice variable importance measure for support vector machines.

```
library(rminer)
```

```
## Warning in rgl.init(initValue, onlyNULL): RGL: unable to open X11 display
```

```
## Warning: 'rgl.init' failed, running with 'rgl.useNULL = TRUE'.
```

```
?fit
```

```
## Help on topic 'fit' was found in the following packages:
##
##   Package               Library
##   modeltools            /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/library
##   generics              /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/library
##   rminer                /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/library
##
##
## Using the first match ...
```

```
M <- fit( y ~ ., data = dat, model = "svm", task = "class",
          search = "heuristic", scale = "inputs")
M2<- fit( y ~ ., data = dat, model = "svm", task = "prob",
          search = "heuristic", scale = "inputs")

P = predict(M, dat.ts)
P2= predict(M2, dat.ts)

print( mmetric(dat.ts$y, P, c("ACC","CONF")))
```

```
## $res
## ACC
##  60
##
## $conf
##      pred
```

```
## target 1 2 3 4
##      1 2 0 0 1
##      2 0 4 0 2
##      3 0 0 1 5
##      4 0 0 0 5
##
## $roc
## NULL
##
## $lift
## NULL
```

```
print( mmetric(dat.ts$y, P2, c("ACC","CONF")))
```

```
## $res
## ACC
##  65
##
## $conf
##       pred
## target 1 2 3 4
##      1 2 1 0 0
##      2 0 6 0 0
##      3 0 1 1 4
##      4 0 1 0 4
##
## $roc
## NULL
##
## $lift
## NULL
```

```
print( mmetric(dat.ts$y, P2, c("AUC")))
```

```
## [1] 0.9116667
```

Cross-validation (CV) can be done directly in the `fit` function via the search argument. There is also another `crossvaldata` function in this package.

In this function, we're going to search over a grid of tuning parameters using CV.

```
M.CV <- fit( y ~ ., data = dat, model = "svm", task = "prob",
         search = list( smethod = "grid", search= list(
           sigma = c(1e-05, 5e-04, 1e-04, 5e-03, 1e-03, 0.01),
           C = c(1,1.5,2,3)), convex = 0,
           method = c("kfold", 10, 498),
           metric="CE"), scale = "inputs")
M.CV@mpar
```

```
## $kpar
## $kpar$sigma
## [1] 1e-04
```

```
##
##
## $C
## [1] 1.5
##
## $task
## [1] "prob"
```

Now, let's see how well can predict to the test data.

```
P.CV = predict( M.CV, dat.ts)
print( mmetric( dat.ts$y, P.CV, c("ACC","CONF")))
```

```
## $res
## ACC
##  80
##
## $conf
##       pred
## target 1 2 3 4
##      1 3 0 0 0
##      2 0 6 0 0
##      3 0 0 3 3
##      4 0 1 0 4
##
## $roc
## NULL
##
## $lift
## NULL
```

```
print( mmetric( dat.ts$y, P.CV, c("AUC")))
```

```
## [1] 0.9333333
```

See the documentation of "Importance" for the many options. I'm doing a single (i.e., no interactions) importance measure due to the number of features (use `method = "DSA"` for interactions).

```
Impor <- Importance(M.CV, data = dat)
colnames(dat)[ Impor$imp>quantile(Impor$imp,probs = 1-10/2309)]
```

```
##  [1] "x.229"  "x.407"  "x.566"  "x.1003" "x.1238" "x.1799" "x.1954" "x.1994"
##  [9] "x.2050" "x.2275"
```

Note: when I've used rminer in the past the results were not consistent from run to run even if a seed was set.

Gaussian Processes Classification

To demonstrate the application of Gaussian Processes (GPs) to this dataset, we'll need to use the `kernlab` package in R, which provides support for kernel-based machine learning methods, including GPs.

Here's an example of how you might use Gaussian Processes for classification with the `Khan` dataset:

```
library(kernlab)
?gausspr

# Make factor versions of the outcome
ytrain <- factor(Khan$ytrain, levels = c(1:4))
ytest <- factor(Khan$ytest, levels = c(1:4))

# Create a Gaussian Processes model
# The 'rbfdot' is the Radial Basis Function (RBF) kernel which common choice for the kernel
# You can experiment with different kernels
# Train the model with the training dataset
gp.model <- gausspr(Khan$xtrain, ytrain, kernel="rbfdot")
```

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

```
gp.model
```

```
## Gaussian Processes object of class "gausspr"
## Problem type: classification
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.000234514681507366
##
## Number of training instances learned : 63
## Train error : 0
```

Notice that there is an automatic estimation of the `sigma` parameter.

Now let's predict on the test dataset:

```
predictions <- predict(gp.model, Khan$xtest, type="probabilities")
round(head(predictions),2)
```

```
##         1    2    3    4
## [1,] 0.21 0.28 0.27 0.25
## [2,] 0.22 0.28 0.25 0.25
## [3,] 0.19 0.28 0.26 0.28
## [4,] 0.14 0.42 0.25 0.19
## [5,] 0.25 0.27 0.23 0.25
## [6,] 0.16 0.26 0.32 0.25
```

```
# Export which probability is the maximum
# Turn it into a factor with the same levels
pred_y <- factor(apply(predictions,1,which.max), levels = c(1:4))
# Evaluate the model's performance
library(caret)
```

```
## Loading required package: ggplot2
```

```
##
## Attaching package: 'ggplot2'

## The following object is masked from 'package:kernlab':
##
##     alpha

## Loading required package: lattice
```

```
confusionMatrix(pred_y, ytest )
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction 1 2 3 4
##          1 0 0 0 0
##          2 3 6 4 4
##          3 0 0 2 0
##          4 0 0 0 1
##
## Overall Statistics
##
##                Accuracy : 0.45
##                  95% CI : (0.2306, 0.6847)
##     No Information Rate : 0.3
##     P-Value [Acc > NIR] : 0.1133
##
##                   Kappa : 0.2171
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3 Class: 4
## Sensitivity             0.00   1.0000   0.3333   0.2000
## Specificity             1.00   0.2143   1.0000   1.0000
## Pos Pred Value           NaN   0.3529   1.0000   1.0000
## Neg Pred Value          0.85   1.0000   0.7778   0.7895
## Prevalence              0.15   0.3000   0.3000   0.2500
## Detection Rate          0.00   0.3000   0.1000   0.0500
## Detection Prevalence    0.00   0.8500   0.1000   0.0500
## Balanced Accuracy       0.50   0.6071   0.6667   0.6000
```

The `gausspr` function includes many other parameters that you might need to tune depending on the dataset's characteristics, such as the kernel parameter, regularization parameter, and others. The performance of the GP model can be highly sensitive to these parameters.

Additionally, GPs can be computationally intensive for large datasets due to the inversion of a large matrix, which has a computational complexity of $O(n^3)$ with n being the number of samples. The Khan dataset is quite high-dimensional, and depending on the computational resources, handling such a dataset with GPs might require methods specifically designed for large datasets, such as sparse GPs, or utilizing a subset of the data for training.

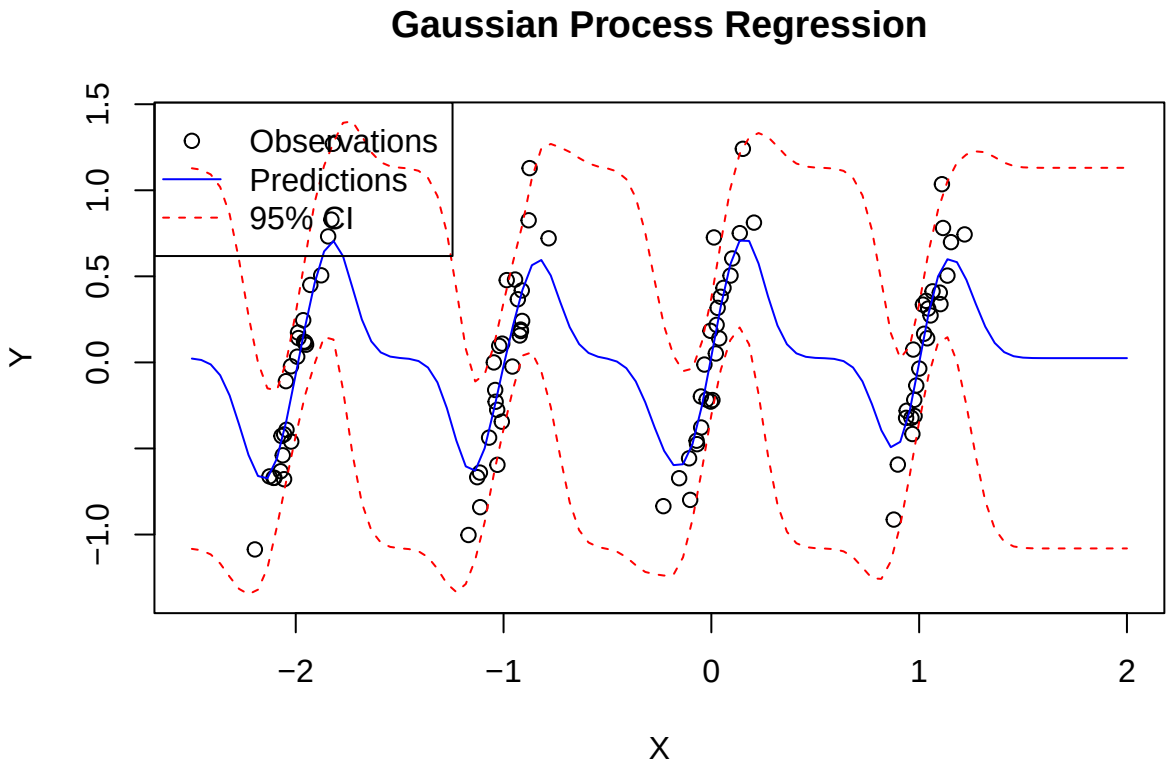
Gaussian Processes Regression

Here, I will provide an example of Gaussian process regression using R and the `kernlab` package to some simulated data.

```
# Simulate some data
set.seed(123)
n <- 100  # number of data points
x <- sort(c(rnorm(n/4, -2, 0.1),
            rnorm(n/4, -1, 0.1),
            rnorm(n/4, 0, 0.1),
            rnorm(n/4, 1, 0.1)))  # predictors
y <- sin(2 * pi * x) + rnorm(n, sd = 0.2)  # responses with some added noise

# Create a Gaussian process model with Radial Basis Function (RBF) kernel
gp_model <- gausspr(x, y, kernel="rbfdot", variance.model = TRUE)
```

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

```
# Predict for new data points
x_new <- seq(-2.5, 2, length.out = 100)
y_pred <- predict(gp_model, x_new)

# Get standard deviations of predictions to understand uncertainty
sdevs <- predict(gp_model, x_new, type="sdeviation")

# Plot the results
plot(x, y, main = "Gaussian Process Regression", xlab = "X", ylab = "Y",
     ylim = range(c(y_pred + 2*sdevs,y_pred - 2*sdevs)),
     xlim = range(x_new))
lines(x_new, y_pred, col = "blue")  # regression line
lines(x_new, y_pred + 2*sdevs, col = "red", lty = 2)  # 95% CI upper bound
lines(x_new, y_pred - 2*sdevs, col = "red", lty = 2)  # 95% CI lower bound
legend("topleft", legend=c("Observations", "Predictions", "95% CI"),
       col=c("black", "blue", "red"), lty=c(NA,1,2), pch=c(1,NA,NA))
```



This script performs Gaussian process regression on a simulated dataset and plots the resulting predictions with a 95% confidence interval. The `gausspr` function from `kernlab` is used for the Gaussian process, and an RBF kernel is specified, though other kernels can be used as well.

The prediction results include not just the predicted values but also the variances of the predictions, which can be used to understand the model's certainty about different points. The plot visualizes the true data, the model's predictions, and the 95% confidence interval around those predictions.

Variable Importance

- Determining variable importance is not as straightforward as in some other machine learning models, like random forests or gradient boosting, which have built-in methods for computing feature importance.
- There are several methods to evaluate feature importance in the context of Gaussian Processes.
- One common approach is to use the sensitivity analysis which measures how the prediction will vary with changes in the input features. This can give a sense of which features the model is most sensitive to, and therefore which are the most important.