Support Vector Machines

36-290 – Statistical Research Methodology

Week 9 Thursday – Fall 2021

The Short Version

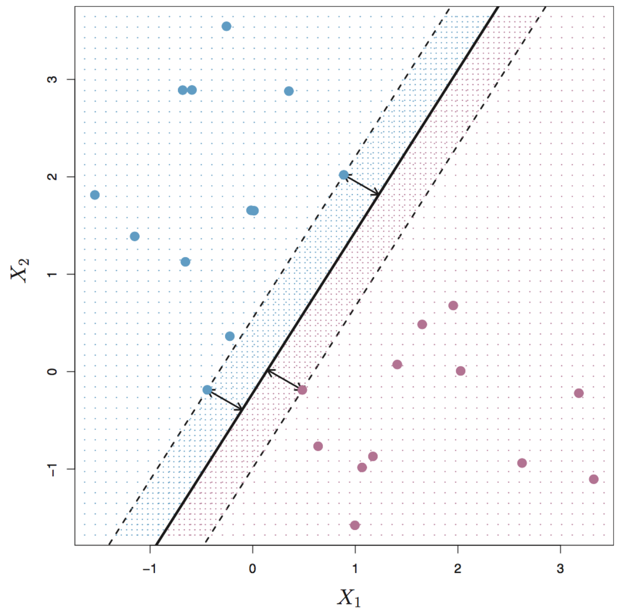
A *support vector machine* is an enigmatically named machine learning algorithm for classification (although some use a regression variant of SVM as well).

SVM transforms predictor data into a *higher-dimensional space* and in that space constructs a linear boundary (i.e., a hyperplane) that optimally separates instances of two classes. (Note that SVM is not designed to tackle analyses in which the response has more than two classes!)

- Like a KNN model, an SVM model is a purely predictive model, useless for inference…
- …and like a KNN model, an SVM model utilizes distances between data, so scaling the predictor data is required…
- …and like a KNN model, an SVM model works only with quantitative predictor variables.

Let's build up SVM qualitatively, one layer at a time…

Maximum Margin Classifier

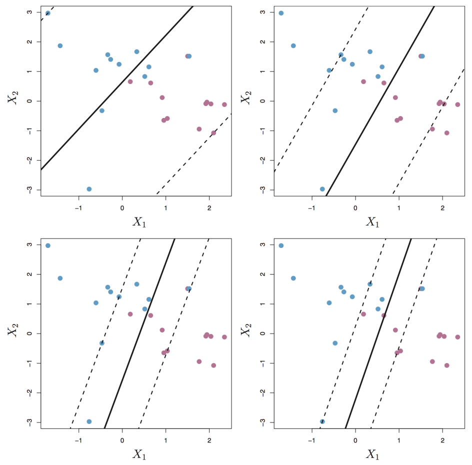


(Figure 9.3, *Introduction to Statistical Learning* by James et al.)

The *maximum margin classifier* determines the linear boundary (or "separating hyperplane") in the native space of the predictor data that has the largest *minimum* distance to a training datum. The MMC is very sensitive to the choice of training data, and in the end is not useful in real-life classification settings, as it requires a complete separation between instances of the two classes.

(But while we are here: look at the three short line segments that are perpendicular to the boundary. These are *support vectors*: they "hold up," or support, the linear boundary.)

Support Vector Classifier



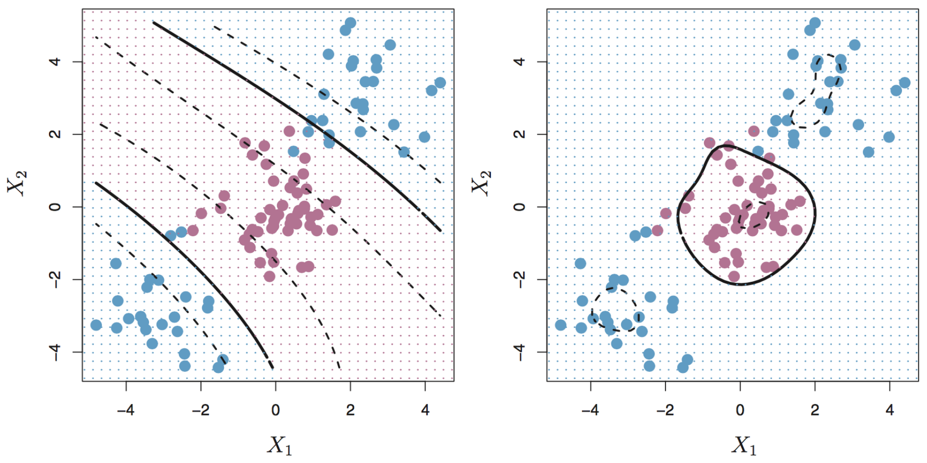
(Figure 9.7, *Introduction to Statistical Learning* by James et al.)

The *support vector classifier* improves upon the MMC by allowing instances of the two classes to overlap. (Good!) It still determines a linear boundary in the native space of the predictor data, and adds to MMC a tuning parameter (conventionally C or "cost") that controls for the rate of boundary violations. As $C \to \infty$, the SVC becomes more tolerant to violations.

The figure above shows modeled linear boundaries given four different values of C, from more tolerant of violations at upper left (high C) to less tolerant at lower right (low C). Determining the appropriate value of C requires splitting training data into training and validation datasets, or applying cross-validation with the training set data.

(However, the `e1071` package in `R` has functions that does cross-validation on the training set for you, to find the optimal value of C.)

Support Vector Machine



(Figure 9.9, *Introduction to Statistical Learning* by James et al.)

And finally, the *support vector machine*. SVM is an extension to SVC that, like SVC, utilizes a linear boundary that can be violated, but that defines that boundary in a higher-dimensional space.

Example: SVM with a polynomial kernel of degree 2 transforms a space where $p = 2$ to one where $p = 5$:

$$X_1, X_2 \rightarrow X_1, X_2, X_1^2, X_2^2, X_1X_2 \, .$$

Support Vector Machine: Kernel

What is a kernel, you ask. The mathematical details are complex are will not be recreated here, particularly as they don't particularly help one build intuition about what SVM does. (However, feel free to look in ISLR for details!) Ultimately, SVM:

- determines an optimal linear boundary in a space of dimensionality $> p$;
- uses inner (or dot) products to solve for that linear boundary; and
- exploits the fact that evaluating kernel functions in the native p-dimensional space of the data is equivalent (and far less computationally intensive!) to explicitly transforming the data vectors to a higher-dimensional space and computing inner products there. (This is the so-called "kernel trick.")

Different kernels encapsulate different mappings to higher-dimensional spaces and thus lead to the creation of *different boundaries*. (Hence SVM with different kernel choices will yield different test-set MCR values!) Common kernel choices are `linear` (which is actually equivalent to SVC…hence no kernel trick here), `polynomial`, and `radial`. Note that each has tuning parameters. For instance, for the `polynomial` kernel, the tuning parameters are the cost C and the polynomial degree d.

SVM: Example

We'll begin by importing data on 500 stars and 500 quasars. (**Note:** like KNN, SVM only works with quantitative predictor variables.)

```
##      col.ug             col.gr              col.ri              col.iz              mag.r         class    
##  Min.   :-4.2274   Min.   :-2.98092   Min.   :-0.40610   Min.   :-3.69967   Min.   :14.43   QSO :500  
##  1st Qu.: 0.6613   1st Qu.: 0.09591   1st Qu.: 0.02866   1st Qu.: 0.02976   1st Qu.:17.95   STAR:500  
##  Median : 1.1102   Median : 0.26471   Median : 0.12162   Median : 0.14411   Median :18.75             
##  Mean   : 1.3196   Mean   : 0.37682   Mean   : 0.21581   Mean   : 0.18544   Mean   :18.66             
##  3rd Qu.: 1.7465   3rd Qu.: 0.51801   3rd Qu.: 0.25169   3rd Qu.: 0.29248   3rd Qu.:19.47             
##  Max.   : 6.2807   Max.   : 2.68311   Max.   : 3.39274   Max.   : 4.04392   Max.   :24.82             
```

We will use the functions of the `e1071` package below, after scaling the predictors (!) and performing a 70-30 data split.

SVM: Example With Linear Kernel

```
library(e1071)

set.seed(202) # reproducible cross-validation
tune.out = tune(svm,class~.,data=df.train,kernel="linear",ranges=list(cost=10^seq(-2,2,by=0.2)))
cat("The estimated optimal value for C is ",as.numeric(tune.out$best.parameters),"\n")
```

```
## The estimated optimal value for C is  1.584893
```

```
resp.pred = predict(tune.out$best.model,newdata=df.test)
mean(resp.pred!=df.test$class) ; table(resp.pred,df.test$class)
```

```
## [1] 0.1433333
```

```
##
## resp.pred QSO STAR
##      QSO  138   29
##      STAR  14  119
```

Compare with…

```
log.out = glm(class~.,data=df.train,family=binomial)
log.prob = predict(log.out,newdata=df.test,type="response")
log.pred = ifelse(log.prob>0.5,"STAR","QSO")
mean(log.pred!=df.test$class) ; table(log.pred,df.test$class)
```

```
## [1] 0.1766667
```

```
##
## log.pred QSO STAR
##     QSO  132   33
##     STAR  20  115
```

SVM: Example With Polynomial Kernel

```
set.seed(202)
tune.out = tune(svm,class~.,data=df.train,kernel="polynomial",
                ranges=list(cost=10^seq(0,4,by=0.5),degree=2:4))
cat("The estimated optimal values for C and degree are ",as.numeric(tune.out$best.parameters),"\n")
```

```
## The estimated optimal values for C and degree are  1000 3
```

```
resp.pred = predict(tune.out$best.model,newdata=df.test)
(svm.poly.mcr = mean(resp.pred!=df.test$class))
```

```
## [1] 0.1033333
```

```
table(resp.pred,df.test$class)
```

```
##          
## resp.pred QSO STAR
##      QSO  130    9
##      STAR  22  139
```

Note: if the optimal value is a boundary value (for instance, the optimal value of cost is 1 when the minimum value considered is 1), change the `ranges` and rerun the code. Also, note the very sparse nature of the `ranges` grid: this is to ensure a relatively small computation time. You can iterate if you want to hone in more precisely on the tuning parameter optimal values.

SVM: Example With Radial Kernel

```
set.seed(202)
tune.out = tune(svm,class~.,data=df.train,kernel="radial",
                ranges=list(cost=10^seq(-1,1,by=0.5),gamma=10^seq(-1,1,by=0.4)))
cat("The estimated optimal values for C and gamma are ",as.numeric(tune.out$best.parameters),"\n")
```

```
## The estimated optimal values for C and gamma are  1 0.6309573
```

```
resp.pred = predict(tune.out$best.model,newdata=df.test)
(svm.poly.mcr = mean(resp.pred!=df.test$class))
```

```
## [1] 0.05
```

```
table(resp.pred,df.test$class)
```

```
##
## resp.pred QSO STAR
##      QSO  142    5
##      STAR  10  143
```

If you go back a few slides, you will see that a radial-kernel SVM model does **much** better than a logistic regression model, in terms of misclassification rate.

Note that in our examples, we assumed that the class-separation threshold is 0.5 (which is OK because we had balanced classes). If you wish to create a ROC curve, you'd pass the argument `probability=TRUE` to *both* `tune()` and `predict()`; the output of the latter will be a matrix, and you'd extract the second column. You'd then pass that vector of numbers into `roc()`.