Chapter 4

Chapter 4. Over-Fitting and Model Tuning

4.1 The problem of over-fitting

4.2 Model tuning

4.3 Data splitting

Pre-processing the predictor data Estimating model parameters Selecting predictors for the model Evaluating model performance Fine tuning class prediction rules (via ROC curves)

Nonrandom approaches, random sample, stratified random sampling, maximum dissimilarity sampling

4.4 Resampling techniques

K-fold cross-validation

Generalized cross-validation

Repeated training/test splits

The bootstrap

4.5 Case study: credit scoring

4.6 Choosing final tuning parameters

4.7 Data splitting recommendations

simple 10-fold cross-validation should provide acceptable variance, low bias, and is relatively quick to compute

4.8 Choosing between models

1. Start with several models that are the least interpretable and most flexible, such as boosted trees or support vector machines. Across many problem domains, these models have a high likelihood of producing the empirically optimum results (most accurate)
2. Investigate simpler models that are less opaque(not complete black boxes), such as multivariate adaptive regression splines (MARS), partial least squares, generalized addictive models, or naive bayes models.
3. Consider using the simplest model that reasonably approximates the performance of the more complex methods

Paired comparisons: a paired t-test can be used to evaluate the hypothesis that the models have equivalent accuracies (on average) or, analogously, that the mean difference in accuracy for the resampled data sets is zero

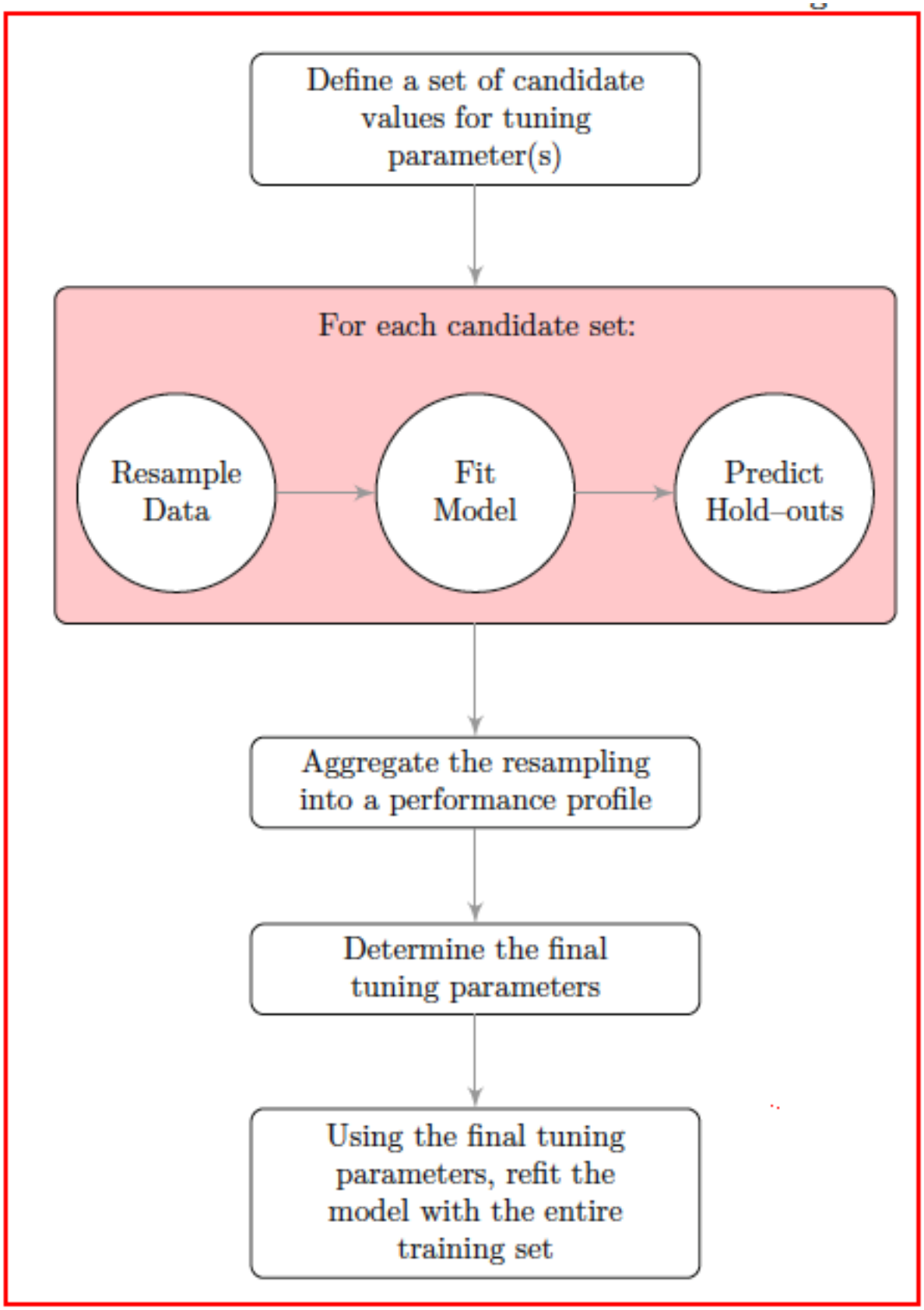

Define a set of candidate values for tuning parameter(s)
For each candidate set:
Resample Data
Fit Model
Predict Hold–outs
Aggregate the resampling into a performance profile
Determine the final tuning parameters
Using the final tuning parameters, refit the model with the entire training set

Figure 1: 1

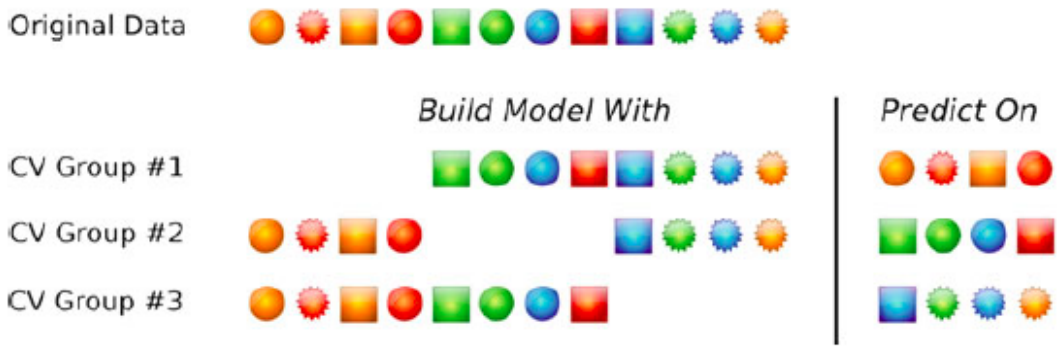


Fig. 4.6: A schematic of threefold cross-validation. Twelve training set samples are represented as symbols and are allocated to three groups. These groups are left out in turn as models are fit. Performance estimates, such as the error rate or R^2 are calculated from each set of held-out samples. The average of the three performance estimates would be the cross-validation estimate of model performance. In practice, the number of samples in the held-out subsets can vary but are roughly equal size

Figure 2: 2

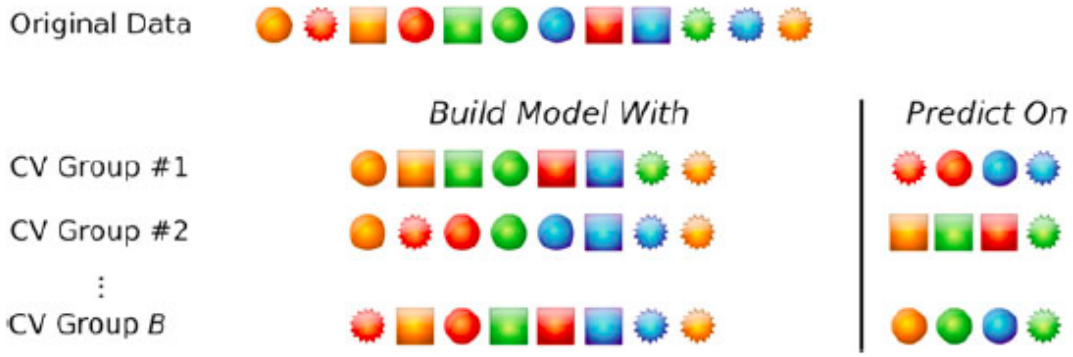


Fig. 4.7: A schematic of B repeated training and test set partitions. Twelve training set samples are represented as symbols and are allocated to B subsets that are 2/3 of the original training set. One difference between this procedure and k-fold cross-validation are that samples can be represented in multiple held-out subsets. Also, the number of repetitions is usually larger than in k-fold cross-validation

Figure 3: 3

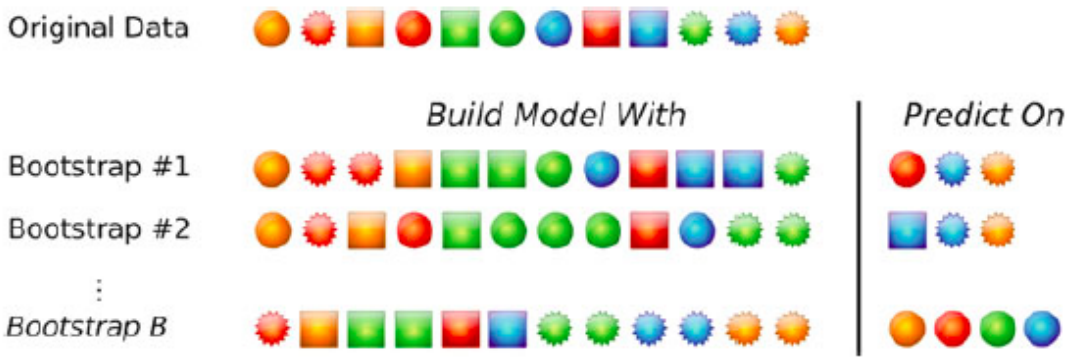


Fig. 4.8: A schematic of bootstrap resampling. Twelve training set samples are represented as symbols and are allocated to B subsets. Each subset is the same size as the original and can contain multiple instances of the same data point. Samples not selected by the bootstrap are predicted and used to estimate model performance

Figure 4: 4

4.9 Computing

```
library(AppliedPredictiveModeling)
library(caret)
```

```
## Loading required package: lattice
```

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

```
# library(Design)
library(e1071)
library(ipred)
library(MASS)
```

Data Spitting

```
data(twoClassData)
```

```
str(predictors)
```

```
## 'data.frame':    208 obs. of  2 variables:
##  $ PredictorA: num  0.158 0.655 0.706 0.199 0.395 ...
##  $ PredictorB: num  0.1609 0.4918 0.6333 0.0881 0.4152 ...
```

```
str(classes)
```

```
##  Factor w/ 2 levels "Class1","Class2": 2 2 2 2 2 2 2 2 2 2 ...
```

```
set.seed(1)
trainingRows <- createDataPartition(classes, p=.80, list = FALSE)
head(trainingRows)
```

```
##      Resample1
## [1,]         1
## [2,]         2
## [3,]         3
## [4,]         4
## [5,]         5
## [6,]         6
```

```
trainPredictors <- predictors[trainingRows,]
trainClasses <- classes[trainingRows]
testPredictors <- predictors[-trainingRows,]
testClasses <- classes[-trainingRows]
```

```
str(trainPredictors)
```

```
## 'data.frame':    167 obs. of  2 variables:
##  $ PredictorA: num  0.158 0.655 0.706 0.199 0.395 ...
##  $ PredictorB: num  0.1609 0.4918 0.6333 0.0881 0.4152 ...
```

```
str(testPredictors)
```

```
## 'data.frame':    41 obs. of  2 variables:
##  $ PredictorA: num  0.0658 0.1056 0.2909 0.4129 0.0472 ...
##  $ PredictorB: num  0.1786 0.0801 0.3021 0.2869 0.0414 ...
```

Resampling

```
set.seed(1)
repeatedSplits <- createDataPartition(trainClasses, p=.80,times=3)
str(repeatedSplits)
```

```
## List of 3
##  $ Resample1: int [1:135] 1 2 4 5 6 8 9 10 11 12 ...
##  $ Resample2: int [1:135] 2 3 4 6 7 8 9 11 14 15 ...
##  $ Resample3: int [1:135] 4 5 6 7 8 9 11 13 14 15 ...
```

```
set.seed(1)
cvSplits <- createFolds(trainClasses, k=10, returnTrain = TRUE)
str(cvSplits)
```

```
## List of 10
##  $ Fold01: int [1:151] 2 3 4 5 6 7 11 12 13 14 ...
##  $ Fold02: int [1:150] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold03: int [1:150] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold04: int [1:151] 1 2 3 4 5 7 8 9 10 11 ...
##  $ Fold05: int [1:150] 1 2 3 5 6 7 8 9 10 11 ...
##  $ Fold06: int [1:150] 1 2 3 4 5 6 8 9 10 11 ...
##  $ Fold07: int [1:150] 1 3 4 5 6 7 8 9 10 11 ...
##  $ Fold08: int [1:151] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold09: int [1:150] 1 2 4 5 6 7 8 9 10 12 ...
##  $ Fold10: int [1:150] 1 2 3 4 6 7 8 9 10 11 ...
```

```
fold1 <- cvSplits[[1]]
fold1
```

```
##   [1]   2   3   4   5   6   7  11  12  13  14  15  16  17  18  19  20  21
##  [18]  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  38  39
##  [35]  40  41  42  43  44  46  48  49  50  51  52  53  54  55  56  57  58
##  [52]  59  60  61  62  63  64  65  66  67  68  69  70  71  73  74  75  76
##  [69]  77  78  79  80  81  82  83  84  85  86  87  89  90  91  92  93  94
##  [86]  95  97  99 100 102 103 104 105 106 107 108 109 110 111 112 113 114
## [103] 115 116 117 118 119 120 121 123 124 125 126 127 128 129 130 132 133
## [120] 134 135 136 137 138 139 140 141 143 144 145 146 148 149 150 151 152
## [137] 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167
```

```
cvPredictors1 <- trainPredictors[fold1,]
cvClasses1 <- trainClasses[fold1]
nrow(trainPredictors)
```

```
## [1] 167
```

```
nrow(cvPredictors1)
```

```
## [1] 151
```

Basic Model Building in R

```
# the formula interface
modelFunction(price ~ numBedrooms + numBaths + acres, data = housingData)
```

```
## Error in modelFunction(price ~ numBedrooms + numBaths + acres, data = housingData): could not find fu
```

```
# the non-formula interface
modelFunction(x = housePredictors, y = price)
```

```
## Error in modelFunction(x = housePredictors, y = price): could not find function "modelFunction"
```

```r
# knn3
trainPredictors <- as.matrix(trainPredictors)
knnFit <- knn3(x = trainPredictors, y = trainClasses, k=5)
knnFit
```

```
## 5-nearest neighbor model
## Training set outcome distribution:
##
## Class1 Class2
##     89     78
```

```r
testPredictions <- predict(knnFit, newdata = testPredictors, type = "class")
head(testPredictions)
```

```
## [1] Class2 Class2 Class1 Class1 Class2 Class2
## Levels: Class1 Class2
```

```r
str(testPredictions)
```

```
##  Factor w/ 2 levels "Class1","Class2": 2 2 1 1 2 2 2 2 2 2 ...
```

Determination of tuning parameters

```r
library(caret)
data(GermanCredit)
```

```r
# Data splitting
GermanCredit <- GermanCredit[, -nearZeroVar(GermanCredit)]
GermanCredit$CheckingAccountStatus.lt.0 <- NULL
GermanCredit$SavingsAccountBonds.lt.100 <- NULL
GermanCredit$EmploymentDuration.lt.1 <- NULL
GermanCredit$EmploymentDuration.Unemployed <- NULL
GermanCredit$Personal.Male.Married.Widowed <- NULL
GermanCredit$Property.Unknown <- NULL
GermanCredit$Housing.ForFree <- NULL

set.seed(100)
inTrain <- createDataPartition(GermanCredit$Class, p = .8)[[1]]
GermanCreditTrain <- GermanCredit[ inTrain, ]
GermanCreditTest  <- GermanCredit[-inTrain, ]
```

```r
set.seed(1056)
svmFit <- train(Class ~ .,
                data = GermanCreditTrain,
                method = "svmRadial")
```

```r
set.seed(1056)
svmFit <- train(Class ~ .,
                data = GermanCreditTrain,
                method = "svmRadial",
                preProc = c('center','scale'))
```

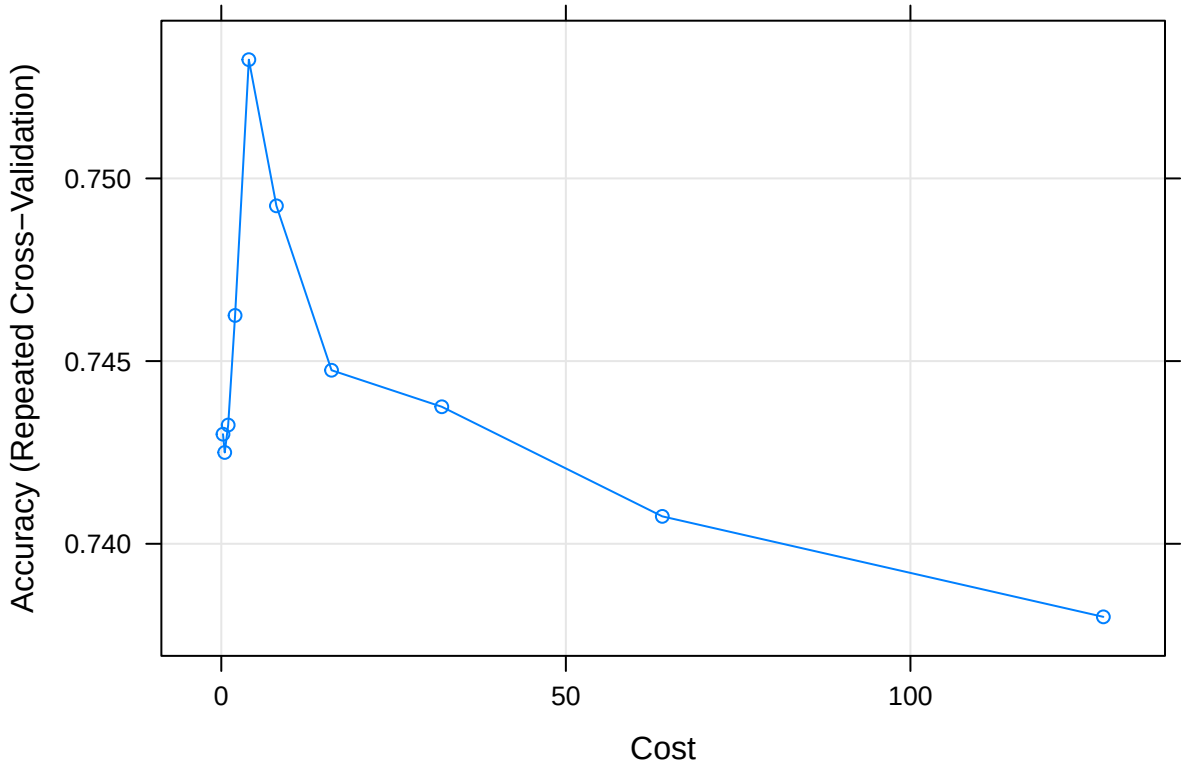
```r
set.seed(1056)
svmFit <- train(Class ~ .,
                data = GermanCreditTrain,
                method = "svmRadial",
                preProc = c('center','scale'),
                tuneLength = 10)
```

```r
set.seed(1056)
svmFit <- train(Class ~ .,
                data = GermanCreditTrain,
                method = 'svmRadial',
                preProc = c('center','scale'),
                tuneLength = 10,
                trControl = trainControl(method = 'repeatedcv',repeats = 5, classProbs =  TRUE))
```

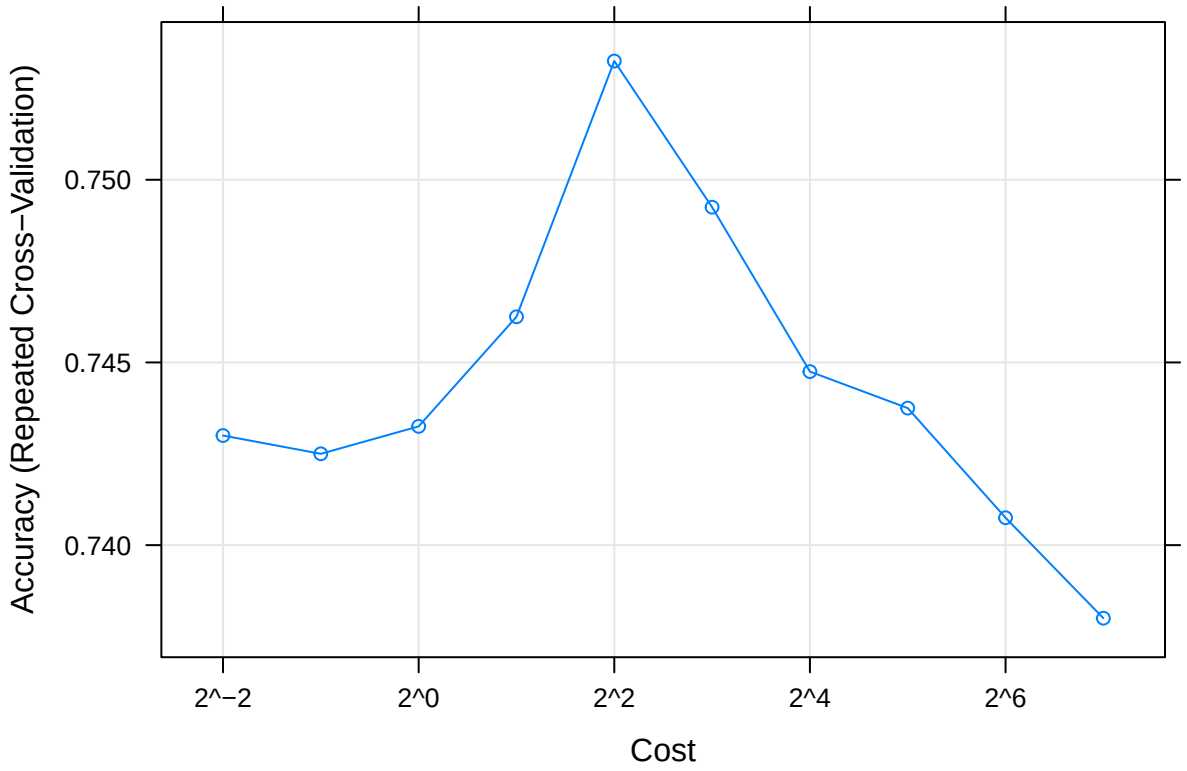
```r
svmFit
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 800 samples
##  41 predictor
##   2 classes: 'Bad', 'Good'
##
## Pre-processing: centered (41), scaled (41)
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 720, 720, 720, 720, 720, 720, ...
## Resampling results across tuning parameters:
##
##   C       Accuracy  Kappa
##     0.25  0.74300   0.3542454
##     0.50  0.74250   0.3542944
##     1.00  0.74325   0.3292659
##     2.00  0.74625   0.3309138
##     4.00  0.75325   0.3320549
##     8.00  0.74925   0.3181382
##    16.00  0.74475   0.2953169
##    32.00  0.74375   0.2984285
##    64.00  0.74075   0.2929082
##   128.00  0.73800   0.2841027
##
## Tuning parameter 'sigma' was held constant at a value of 0.01418087
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.01418087 and C = 4.
```

```r
plot(svmFit)
```




```
plot(svmFit, scales = list(x=list(log = 2)))
```



```
predictedClasses <- predict(svmFit, GermanCreditTest)
str(predictedClasses)
```

```
##  Factor w/ 2 levels "Bad","Good": 1 1 2 2 1 2 2 2 1 1 ...
```

```
predictedProbs <- predict(svmFit, newdata = GermanCreditTest, type = "prob")
head(predictedProbs)
```

```
##         Bad      Good
## 1 0.5712094 0.4287906
## 2 0.5030683 0.4969317
## 3 0.3377723 0.6622277
## 4 0.1025021 0.8974979
## 5 0.6227635 0.3772365
## 6 0.1498071 0.8501929
```

Between model comparisons

```
set.seed(1056)
logisticReg <- train(Class ~ .,
                     data = GermanCreditTrain,
                     method = "glm",
                     trControl = trainControl(method = "repeatedcv", repeats = 5))
logisticReg
```

```
## Generalized Linear Model
##
## 800 samples
##  41 predictor
##   2 classes: 'Bad', 'Good'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 720, 720, 720, 720, 720, 720, ...
## Resampling results:
##
##   Accuracy  Kappa
##   0.748     0.3573223
```

```
resamp <- resamples(list(SVM = svmFit, Logistic = logisticReg))
summary(resamp)
```

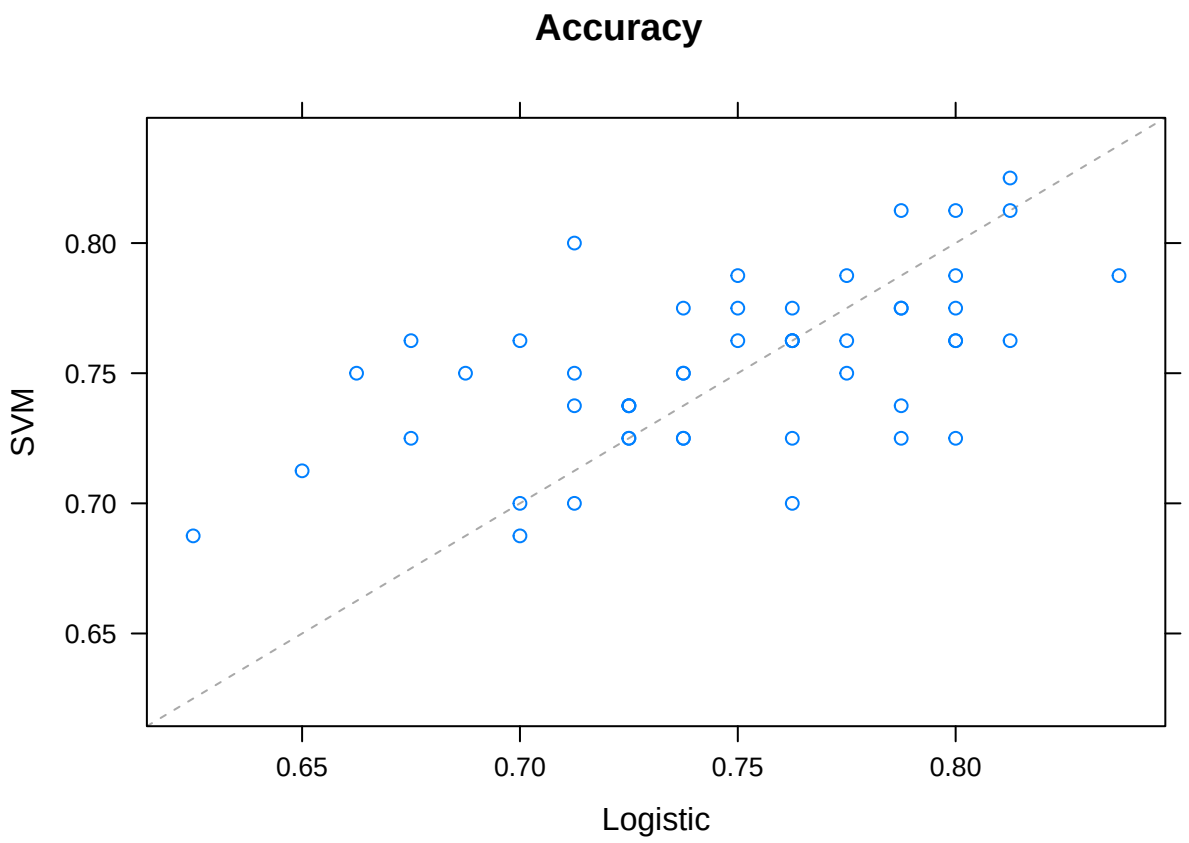
```
##
## Call:
## summary.resamples(object = resamp)
##
## Models: SVM, Logistic
## Number of resamples: 50
##
```

```
## Accuracy
##            Min.  1st Qu.  Median    Mean 3rd Qu.   Max. NA's
## SVM      0.6875 0.725000 0.75625 0.75325  0.7750 0.8250    0
## Logistic 0.6250 0.715625 0.75000 0.74800  0.7875 0.8375    0
##
## Kappa
##                 Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## SVM       0.11971831 0.2774423 0.3436717 0.3320549 0.3729167 0.5270270
## Logistic -0.01351351 0.2797677 0.3670683 0.3573223 0.4471501 0.5886076
##          NA's
## SVM         0
## Logistic    0
```

```
modelDifferences <- diff(resamp)
summary(modelDifferences)
```

```
##
## Call:
## summary.diff.resamples(object = modelDifferences)
##
## p-value adjustment: bonferroni
## Upper diagonal: estimates of the difference
## Lower diagonal: p-value for H0: difference = 0
##
## Accuracy
##          SVM    Logistic
## SVM              0.00525
## Logistic 0.3474
##
## Kappa
##          SVM     Logistic
## SVM              -0.02527
## Logistic 0.08531
```

```
xyplot(resamp)
```


Accuracy
SVM
Logistic
0.65
0.70
0.75
0.80
0.65
0.70
0.75
0.80