Homework1

Michael Ahn

1/13/2019

```r
library(kernlab)
set.seed(42)
```

R Markdown

Question 2.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a classification model would be appropriate. List some (up to 5) predictors that you might use.

A classification model could help determine whether or not federal government closes due to the weather, i.e. snow. Predictors may include:

1. The snow accumulation - The total accumulation may affect the decision differently than the max accumulation since trucks must clear the roads continuously

2. Timing of max accumulation - Overnight will be easier to clean than in the morning during rush hour

3. Temperature - Freezing snow is harder to clean than soft snow

4. Time of year - First snow storm may be harder to clean as the employees are not as trained or rehearsed in protocol

Question 2.2

The files credit_card_data.txt (without headers) and credit_card_data-headers.txt (with headers) contain a dataset with 654 data points, 6 continuous and 4 binary predictor variables. It has anonymized credit card applications with a binary response variable (last column) indicating if the application was positive or negative. The dataset is the "Credit Approval Data Set" from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Credit+Approval) without the categorical variables and without data points that have missing values.

1. Using the support vector machine function ksvm contained in the R package kernlab, find a good classifier for this data. Show the equation of your classifier, and how well it classifies the data points in the full data set. (Don't worry about test/validation data yet; we'll cover that topic soon.)
   Notes on ksvm

- You can use scaled=TRUE to get ksvm to scale the data as part of calculating a classifier.

```r
# Build ksvm model
#   @param data (String): filename for data to be imported
#   @param lambda (vector(int)): C values of ksvm model to be looped through and tried
#   @return rel.error (float): relative error using test set

find_lambda <- function(txt, lambda=100, model="vanilladot") {
  # Import Data
  df <- read.table(txt, header=FALSE, stringsAsFactors=FALSE)

  # Split train, test
  N <- nrow(df)
  n <- floor(nrow(df)*0.75)

  df <- df[sample(N),]
  df.train <- df[1:n,]
  df.test <- df[(n+1):N,]

  # Format train, test
  df.train.x <- as.matrix(df.train[,1:10])
  df.train.y <- as.factor(df.train[,11])
  df.test.x <- as.matrix(df.test[,1:10])
  df.test.y <- as.factor(df.test[,11])

  # errorList will store all the relative errors in each loop iteration
  errorList = c()

  # Loop through each C (lambda) value
  for (i in 1:length(lambda)) {
    model.ksvm <- ksvm(x = df.train.x,
                       y = df.train.y,
                       type="C-svc", kernel=model, C=lambda[i], scaled=TRUE)
    # Predict on testing set
    ans = predict(model.ksvm, df.test.x)

    # Calculate relative error
    error = sum(ans != df.test.y)
    rel.error = error/length(df.test.y)
    # Append to errorList
    errorList[i] = rel.error
  }
```

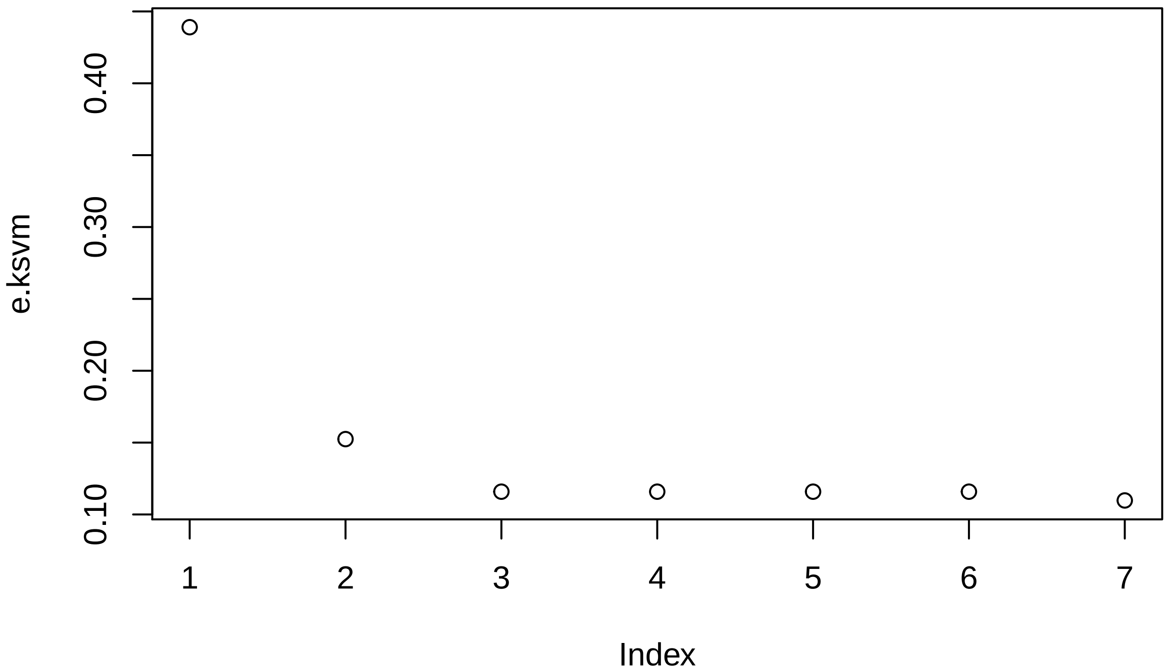
```r
  return(errorList)
}


# Trains model using inputted data
build_model <- function(data, lambda=100, model="vanilladot") {
  df <- data
  x.train <- as.matrix(df[,1:10])
  y.train <- as.factor(df[,11])

  model <- ksvm(x=x.train,
                y=y.train,
                type="C-svc",
                kernel=model,
                C=lambda,
                scaled=TRUE)
  return(model)
}
```

Using the first function, we can create a loop to try out multiple lambda values.
The outputs will show us which of the lambdas provide the most "accurate" model.

```r
d = "credit_card_data.txt"
L = c(10^(-5), 10^(-3), 0.1, 1, 10, 10^3, 10^5)
e.ksvm <- find_lambda(txt=d, lambda=L)
```

```
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
```

```r
plot(e.ksvm)
```



It looks like there is not much variation in the relative errors after the first indexed lambda, at 10^(-5). Let us build a model using our other function, build_model.

```
# Get data
df = read.table(d, header=FALSE, stringsAsFactors=FALSE)
# Randomly resort data
df <- df[sample(nrow(df)),]

# Split training and testing sets
n <- floor(nrow(df)*0.75)
df.train = df[1:n,]
df.train.x = as.matrix(df.train[,1:10])
df.train.y = as.matrix(df.train[,11])

df.test = df[(n+1):nrow(df),]
df.test.x = df.test[,1:10]
df.test.y = df.test[,11]

# Build a model using the best lambda value
e <- which(e.ksvm == min(e.ksvm))
model.ksvm <- build_model(df.train,
                          lambda = L[e])
```

```
##  Setting default kernel parameters
```

Now that we have our model, it is time to both test and evaluate it.

```
# Get coefficients
xmatrix <- model.ksvm@xmatrix[[1]]
xcoef <- model.ksvm@coef[[1]]

a <- colSums(xmatrix * xcoef)
a0 <- -model.ksvm@b
```

```r
# See how model does
# Predict on test.x, and check using test.y
df <- read.table("credit_card_data.txt", stringsAsFactors=FALSE, header=FALSE)
ans.ksvm <- predict(model.ksvm, df.test.x)
ans.ksvm.rel <- sum(ans.ksvm == df.test.y) / length(ans.ksvm)
```

```
## [1] "Coefficients: "
```

```
##           V1           V2           V3           V4           V5
## -0.002848742  0.580705608  0.419950321  0.392126618  1.010800250
##           V6           V7           V8           V9          V10
## -0.054598583  0.076459793 -0.065937051  0.231605184  0.150103661
```

```
## [1] "Intercept: "
```

```
## [1] 0.07473675
```

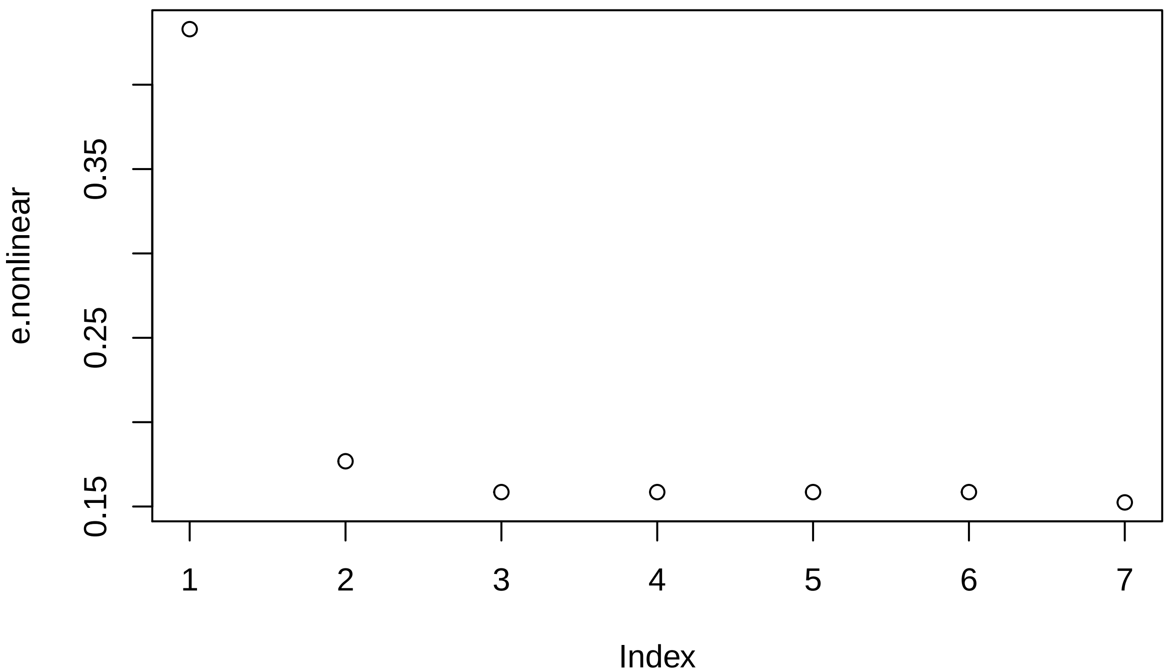
```
## [1] "Model predicts the data set with 0.86 percent accuracy"
```

2. You are welcome, but not required, to try other (nonlinear) kernels as well; we're not covering them in this course, but they can sometimes be useful and might provide better predictions than vanilladot.

```
# Same method as before, but with different "model" argument
# We have d, L saved as variables from our linear model
e.nonlinear <- find_lambda(txt=d,
                           lambda=L,
                           model="polydot")
```

```
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
```

```
plot(e.nonlinear)
```




```
# Build a model
# Let's use the same training and testing data as before
e.nl <- min(which(e.nonlinear == min(e.nonlinear)))
model.nonlinear <- build_model(df.train,
                               lambda = L[e.nl],
                               model="polydot")
```

```
##  Setting default kernel parameters
```

```
# See how model does
ans.nonlinear <- predict(model.nonlinear, df.test.x)
ans.nonlinear.rel <- sum(ans.nonlinear == df.test.y) / length(ans.nonlinear)
ans.nonlinear.rel
```

```
## [1] 0.8902439
```

Both have the same accuracy, but a quick check shows that the predictions themselves were slightly different.

```r
sum(ans.ksvm != ans.nonlinear)
```

```
## [1] 7
```

3. Using the k-nearest-neighbors classification function kknn contained in the R kknn package, suggest a good value of k, and show how well it classifies that data points in the full data set. Don't forget to scale the data (scale=TRUE in kknn).

```
# Install dependencies
library(kknn)
```

```
data <- read.table("credit_card_data-headers.txt", stringsAsFactors=FALSE, header=TRUE)
sample <- sample(1:nrow(data), floor(nrow(data)*0.6))


scores <- c(1, 1, 1, 1, 1, 1)
kvalue <- c(1,3,5,7,9,11)
for (j in 1:length(kvalue)) {

  i=1:654
  ansList = c()
  # loop through each row of the data
  for (i in 1:654){

    # build model
    kknn <- kknn(R1~.,
                  data[-i,], data[i,],
                  k=kvalue[j],
                 kernel="optimal",
                 distance=2,
                  scale=TRUE)
    # predict and round
    ans <- round(fitted.values(kknn))
    ansList[i] = (ans == data[i,11])
  }
  scores[j] <- sum(ansList)/ length(ansList)
}
best_k <- kvalue[which(scores==max(scores))[1]]
```

```
## [1] "Best accuracy score: "
```

```
## [1] 0.851682
```

```
## [1] "Best k-value: "
```

```
## [1] 5
```

We have the accuracy score and the "good" k-value associated with it.