Homework 1

Joe Diaz

8/23/2019

Assignment 1

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.

For context, I work in a weekly subscription based company. One key problems subscription based companies have is customer churn (customer unsubscribing). Classificaiton can be used to predict if a given customer will unsubscribe. This can help the business by catching those who are likely to unsubscribe. by giving them offers/discounts before they get the chance to unsubscribe.

It will be a classification model with 2 groups: kept subscription and unsubscribed.

Some key predictors I would use for this model are:

1. Order history
2. Average price order basket
3. Average # of items per order
4. How long has the customer been subscribed
5. Customer demographics if available (age, income, etc.)

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.)

Using the Provided Code

```
#install.packages('kernlab')
library('kernlab')


data <- as.matrix(read.table('credit_card_data.txt', sep = '\t'))


# call ksvm. Vanilladot is a simple linear kernel.
model <- ksvm(data[,1:10],data[,11],type="C-svc",kernel="vanilladot",C=100,scaled=TRUE)
```

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

```
# calculate a1...am
a <- colSums(model@xmatrix[[1]] * model@coef[[1]])
#a
# calculate a0
a0 <- -model@b
#a0
# see what the model predicts
pred <- predict(model,data[,1:10])
pred
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [211] 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [246] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
## [281] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0
## [316] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [351] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [386] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [421] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [456] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1
## [561] 1 1 1 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
## [596] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
## [631] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
# see what fraction of the model's predictions match the
# actual classification
sum(pred == data[,11]) / nrow(data)
```

```
## [1] 0.8639144
```

With the code provided, we are able to get about **86%** of the classifications correct. We can further improve this by changing the kernel used and by manipulating the C parameter.

The equation's coefficients and intercept is found below in the variable a for the coefficients and a0 for the intercept

```
a
```

```
##            V1            V2            V3            V4            V5
## -0.0010065348 -0.0011729048 -0.0016261967  0.0030064203  1.0049405641
##            V6            V7            V8            V9           V10
## -0.0028259432  0.0002600295 -0.0005349551 -0.0012283758  0.1063633995
```

```
a0
```

```
## [1] 0.08158492
```

Generally speaking, assuming the data is linearly separable, an increase in C would mean the model will want to avoid misclassifications (of the training data) at a greater rate. We can easily make an SVM model to be amazing at classifying this specific data set with a high enough C.

note: in this case, the data is not easily seperable by the ***vanilladot*** *kernel, so a high C will actually decrease accuracy. After some trial and error,* ***rbfdot*** *seems to be the best kernel for this dataset.*

```
# call ksvm. Vanilladot is a simple linear kernel.
model <- ksvm(data[,1:10],data[,11],type="C-svc",kernel="rbfdot",C=10000000,scaled=TRUE)

# calculate a1...am
a <- colSums(model@xmatrix[[1]] * model@coef[[1]])

# calculate a0
a0 <- -model@b

# see what the model predicts
pred <- predict(model,data[,1:10])

# see what fraction of the model's predictions match the
# actual classification
sum(pred == data[,11]) / nrow(data)
```

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

As you can see, we get a perfect classification with an extremely large C and the proper kernel selected. While, an accuracy of 100% might seem good, in practice, this is not the case. What we have here is a model that is overfitted. In other words, it's really good at classifying the points we have trained it with, but not so much as a general model.

To make a more general model, we can lower C, but without an actual test, we will only be guessing on how good the model actually is. One simple solution is to have a holdout data set. Let's try making a better general model by training on a random 80% of the dataset, while testing on the other 20%.

I will also be performing some parameter tuning in this step. For now, I will only tune C.

```
set.seed(0)

train_ind <-sample(1:nrow(data),floor(.8*nrow(data)) ,replace = TRUE)

train <- data[train_ind,]
test <- data[-train_ind,]

currentAccuracy <- 0
bestAccuracy <- 0
bestC <- 0

# seq(0.01,1.0001)
for (c in seq(0.065,0.065,.0001)){
```

```
  model <- ksvm(train[,1:10],train[,11],type="C-svc",kernel="rbfdot",C=c,scaled=TRUE)

  # calculate a1...am
  a <- colSums(model@xmatrix[[1]] * model@coef[[1]])

  # calculate a0
  a0 <- -model@b

  # see what the model predicts
  pred <- predict(model,test[,1:10])

  # see what fraction of the model's predictions match the
  # actual classification
  currentAccuracy <- sum(pred == test[,11]) / nrow(test)

  if( currentAccuracy > bestAccuracy){

    bestAccuracy <- currentAccuracy
    bestC <- c
    bestModel <- model
  }
}
cat('Achieved best accuracy of',bestAccuracy, 'witk a C of',bestC)
```

```
## Achieved best accuracy of 0.8704319 witk a C of 0.065
```

NOTE: First, I'm iterating through .01 to 1 as I've tried going from 1 to 10000 and the best C was still 1. Also, I set it to only try .65 to save time during execution as it takes a while to run this portion of the code

```
besta <- colSums(bestModel@xmatrix[[1]] * bestModel@coef[[1]])
besta0 <-bestModel@b

besta
```

```
##         V1         V2         V3         V4         V5         V6
##  0.2731385  1.3895525  1.3740521  6.1365227 12.5703655 -5.4749477
##         V7         V8         V9        V10
##  6.5616553 -0.4648938 -2.2706620  5.0107011
```

```
besta0
```

```
## [1] -0.3198568
```

K-Nearest-Neighbors Implementation

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)

I'll do a kknn with the train test methodology in mind. We'll only do some parameter tuning with k to find the best k possible.

Some other ideas to tune parameters:

1. distance parameter in the kknn model
2. the boundary where each prediction becomes a 1 or 0 (right now I've set it if the prediciton is .5 or higher, it is a 1)

```
library(kknn)
data <- read.table('credit_card_data.txt', sep = '\t')
set.seed(0)

train_ind <-sample(1:nrow(data),floor(.8*nrow(data)) ,replace = TRUE)

train <- data[train_ind,]
test <- data[-train_ind,]

bestAccuracy <- 0
bestK <-1

for(k in 1:20){

  knnModel <- kknn(V11 ~ .,train, test, k = k, kernel = "rectangular", scale=TRUE)

  pred <- c()
  for (i in knnModel$fitted.values){

    if(i>=.5){
      pred<-append(pred,1)
    } else {
      pred<-append(pred,0)
    }
}

  accuracy <- sum(pred == test[,11]) / nrow(test)

  if(accuracy>bestAccuracy){
    bestAccuracy<-accuracy
    bestK <- k
  }
}
cat('Achieved best accuracy of',bestAccuracy, 'witk a k of',bestK)
```

```
## Achieved best accuracy of 0.8637874 witk a k of 4
```

Take-aways

1. In SVM, C is essentially a way to control overfitting.
2. The assumption that if C increases, in-sample fit increases and vice versa is only true if the data is linearly seperable.
3. Overfitting is when a model is fitted way too closely on a specific dataset resulting in a bad general model.
4. Using a hold-out dataset can be used to counter-act overfitting
5. parameter-tuning can be used to find the best K and the best C