module4_part2

Andrew Estes

4/13/2022

Assignment

Revisit the credit default data from the Module 3, Part 1 homework. Copy over your old code if you like.

The goal here is to add a tuned RBF network classification method to your comparison of the other methods.

New code should train the RBF network, make a plot of accuracy against its single tuning parameter, state the best tuning parameter and the cross-validated accuracy that it gave, add the RBF network model to your ROC curve, and revise your concluding remarks to take this new model into consideration.

Comment on how well it did compared to the others on this data set.

```
set.seed(3287)
# Skip the first line of the file.
df.raw <- read.csv("default_of_credit_card_clients.csv", skip=1)
# We'll take out SEX and MARRIAGE, which seem like things you shouldn't base
# a decision on. If we don't take out ID, then we'll be training a model
# to basically use your "name" to predict your likelihood to default.
# Renaming and changing Default to a factor to make other code you might
# write happier later on.
df.raw <- df.raw %>%
  rename(Default = default.payment.next.month) %>% select(-ID) %>%
  select(-SEX, -MARRIAGE) %>%
  mutate(Default=factor(if_else(Default==1, "Yes", "No")))
# First, taking random 20% of entire data set, mostly to speed up your
# training runs. My test show we still have enough to get good prediction
# even with fewer data points.
df <- df.raw[sample(1:nrow(df.raw), round(0.2*nrow(df.raw))), ]
# The `downSample()` command from `caret` balances the data.
df <- downSample(x=select(df, -Default), y=df$Default, yname="Default")
# Then we make a training/testing split. You can do the whole assignment just
# with the training data set and cross-validation, but I put this in just in
# case you like to do one more verification of model accuracy.
ind <- createDataPartition(df$Default, p=0.70, list=FALSE)
df.train <- df[ind, ]
df.test <- df[-ind, ]
```

New Model

Running the Radial Basis method.

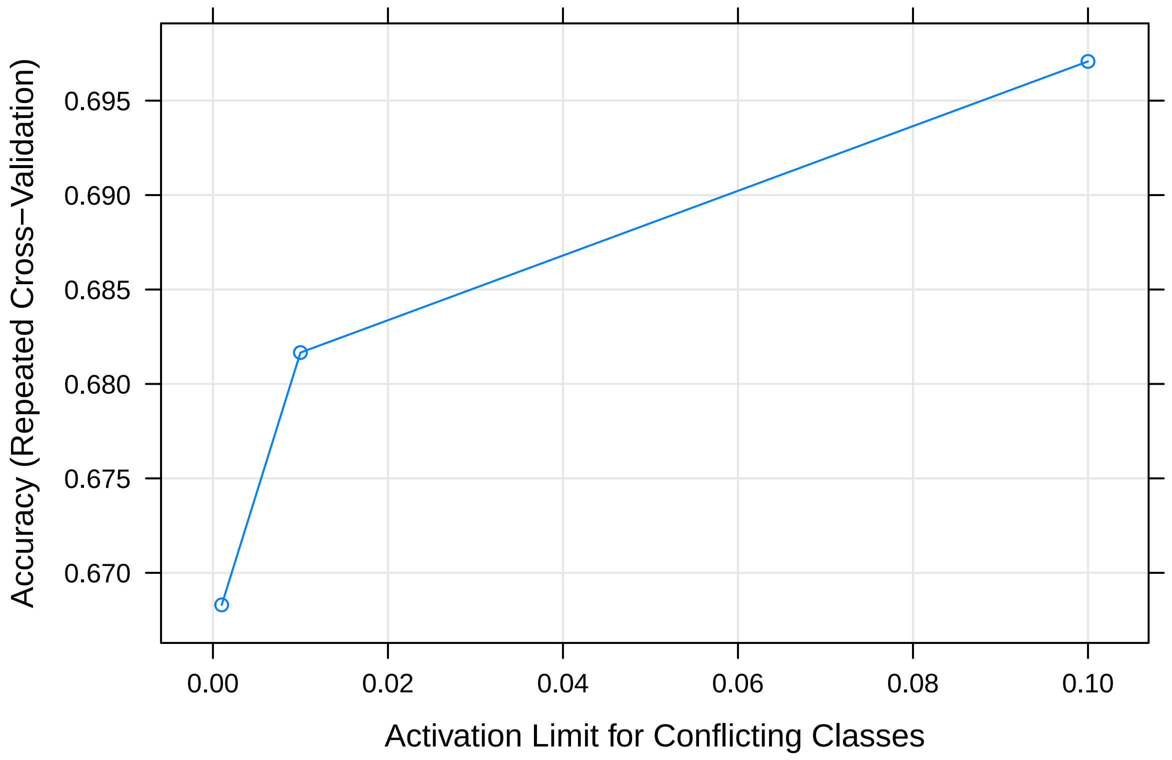
```r
fit.rbf.train <-
  train(Default ~ .,
        data=df.train,
        method="rbfDDA",
        trControl=trainControl(method="repeatedcv",
                               number=3,
                               repeats = 2,
                              savePredictions="final",
                              classProbs = TRUE),
        preProcess=c("scale", "center"),
        trace=FALSE
  )


gg.fit.rbf <- ggplot(fit.rbf.train$results) +
  geom_point(aes(x=negativeThreshold, y=Accuracy)) +
  geom_line(aes(x=negativeThreshold, y=Accuracy)) +
  theme_bw()

fit.rbf.train$bestTune
```

```
##   negativeThreshold
## 3               0.1
```

```r
max(fit.rbf.train$results$Accuracy)
```

```
## [1] 0.6970773
```

```r
plot(fit.rbf.train)
```



The best tune was of a negative Threshold value of 0.1 It lead to a maximum accuracy of 69%.

1

Use the train command with method="glm" to create a logistic regression model using your credit.train data set. Logistic regression is fast, so you can use repeated cross-validation to get a fairly reliable accuracy estimate. What is the accuracy you obtained? (Note: Since our SVM and neural network methods are, in a sense, both elaborations on what logistic regression does, this result is useful as a baseline to see whether those methods offer any benefit.)

```
fit.glm.train <-
  train(Default ~ .,
        data=df.train,
        method="glm",
        trControl=trainControl(method="repeatedcv",
                               number=100,
                               savePredictions="final",
                               classProbs=TRUE)
  )

#fit.glm.train$finalModel
fit.glm.train$results$Accuracy
```

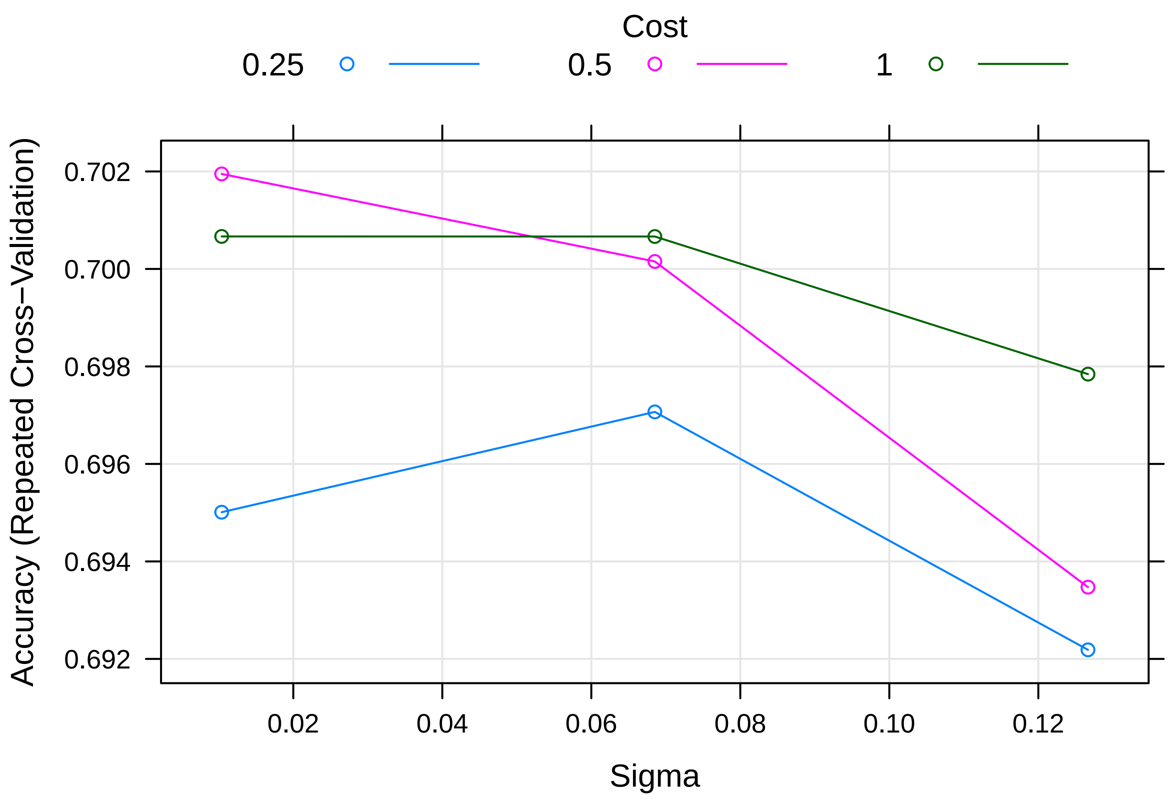
```
## [1] 0.6760965
```

Accuracy is 67%.

2

Use the train command with method="svmRadialSigma" to tune a support vector machine model. You may need to try several runs, but try to end up with a tuning grid that seems to capture a set of optimal parameters and shows drop-off on either side. Because of variation in cross-validation, you may not every find "exactly" perfect optimal parameters, but try to get a range that shows significant drop-off from an optimal zone. Output should include a plot of accuracy vs. tuning parameters as well as a statement of which tuning parameters give the best accuracy and what that accuracy was. (Note: To get ready for Question 4 below, you might want to add the appropriate options to trControl at this stage.)

```r
fit.svm.train <-
  train(Default ~ .,
        data=df.train,
        method="svmRadialSigma",
        trControl=trainControl(method="repeatedcv",
                               number=3,
                               repeats=2,
                               savePredictions="final",
                               classProbs=TRUE)
  )

plot(fit.svm.train)
```



```r
fit.svm.train$bestTune
```

```
##        sigma   C
## 2 0.01038929 0.5
```

```
#fit.svm.train$finalModel
```

The best tuning parameters was at a cost of 0.50 and a sigma between .069. The actual number depends on how the CV divvied up the dataset. The accuracy was 70.9%.

This represents a 3% increase in accuracy over the GLM method even though the GLM had 50-fold CV repeated 50 times, while the SVM has 5-fold CV repeated 3 times.

3

Repeat the process above using method="nnet" and appropriate tuning parameters.

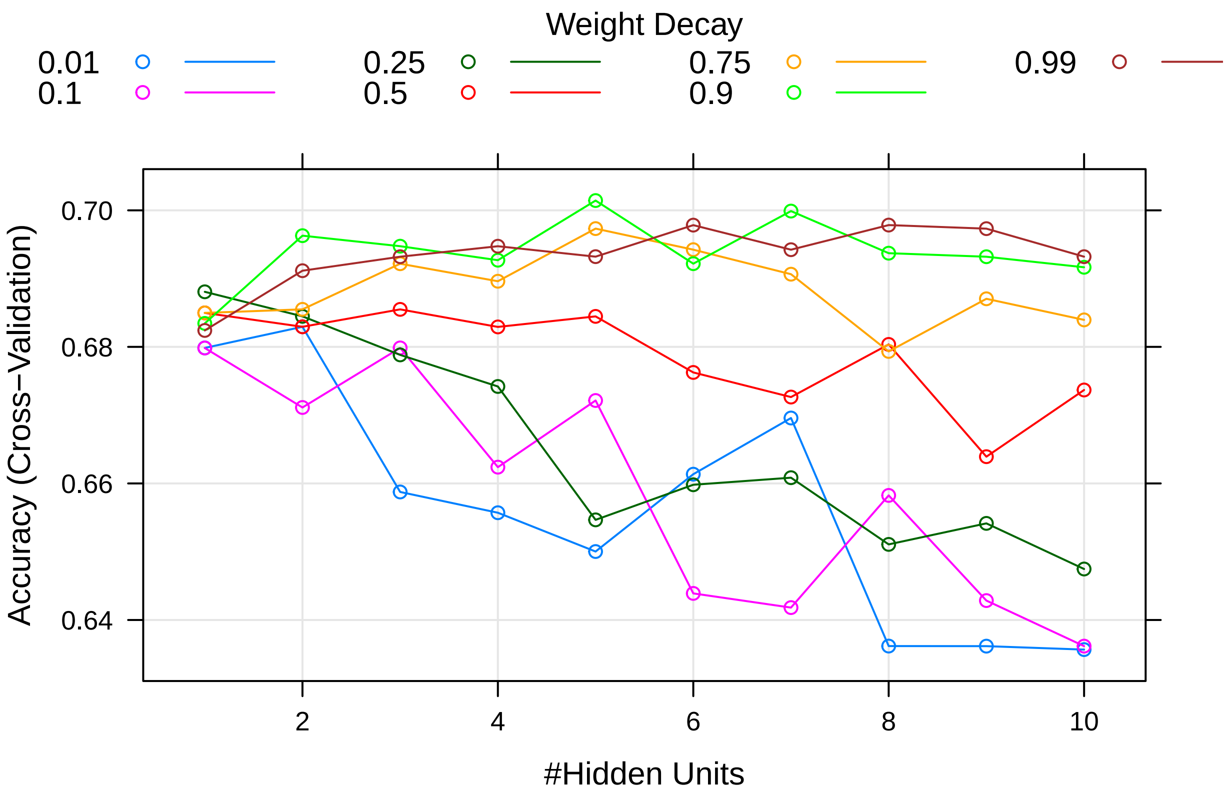
```
fit.nnet.train <-
  train(Default ~ .,
        data=df.train,
        method="nnet",
        trControl=trainControl(method="cv",
                               number=2,
                               savePredictions="final",
                               classProbs=TRUE),
        preProcess=c("scale", "center"),
        tuneGrid=expand.grid(size=c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                    decay=c(.01, .1, .25, .5, .75, .9, .99)),
        trace=FALSE
  )

fit.nnet.train$bestTune
```

```
##    size decay
## 34    5   0.9
```

```
max(fit.nnet.train$results$Accuracy)
```

```
## [1] 0.7014327
```

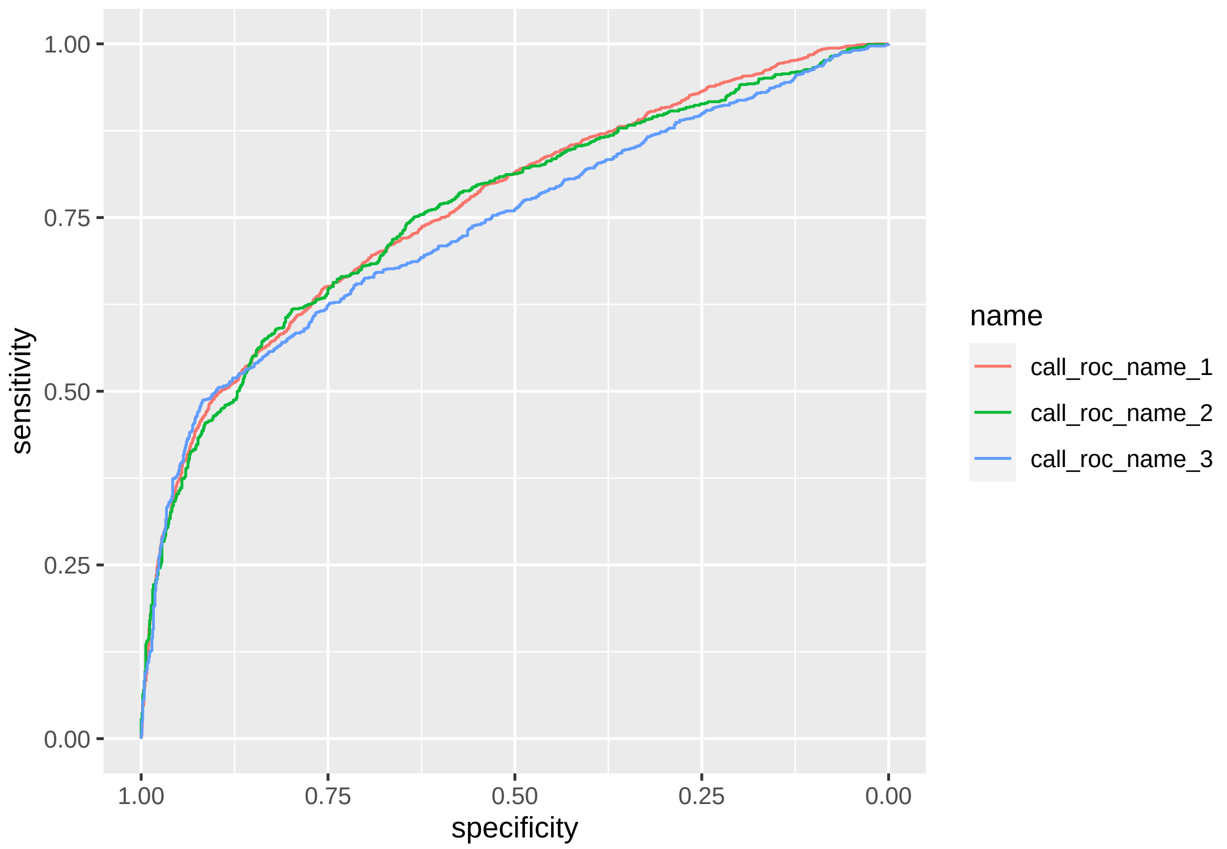
```
plot(fit.nnet.train)
```


Weight Decay
0.01
0.1
0.25
0.5
0.75
0.9
0.99
Accuracy (Cross-Validation)
0.70
0.68
0.66
0.64
2
4
6
8
10
#Hidden Units

4

Use the roc() command from the pROC package (along with plot() command) to create plots of the ROC curves for each model. (You can use the add=TRUE option to the plot command to superimpose the curves on each other. There are also ggplot-based solutions if you want to find them.)

```
glm.plot.black <- roc((fit.glm.train$pred$obs ~ fit.glm.train$pred$Yes),
                      print.auc=TRUE)

svm.plot.blue <- roc((fit.svm.train$pred$obs ~ fit.svm.train$pred$Yes),
                     print.auc=TRUE,
                     col = "blue")

nnet.plot.red <- roc((fit.nnet.train$pred$obs ~ fit.nnet.train$pred$Yes),
                     print.auc=TRUE,
                     col = "red")

#rbf.plot.green <- plot(roc(fit.rbf.train$pred$obs ~ fit.rbf.train$pred$obs),
#                       print.auc=TRUE,
#                       col = "green")

library(pROC)
ggroc(list(call_roc_name_1 = svm.plot.blue, call_roc_name_2 = nnet.plot.red, call_roc_name_3 = glm.plot
```



GLM 73.8% AUC SVM 76.0% AUC NNet 76.9% AUC The RBF did not plot due to predictions not being saved so the ROC could not run the calculations needed. I'm not sure what the issue is but I am running

out of time so am submitting as-is.