Predicting How Good an Excercise is Performed

Toor

March 20, 2017

0) Introduction

Using devices such as Jawbone Up, Nike FuelBand, and Fitbit it is now possible to collect a large amount of data about personal activity relatively inexpensively. These type of devices are part of the quantified self movement – a group of enthusiasts who take measurements about themselves regularly to improve their health, to find patterns in their behavior, or because they are tech geeks. One thing that people regularly do is quantify how much of a particular activity they do, but they rarely quantify how well they do it. In this project, your goal will be to use data from accelerometers on the belt, forearm, arm, and dumbell of 6 participants. They were asked to perform barbell lifts correctly and incorrectly in 5 different ways.

1) Loading, partitioning and Cleaning Data

First, we load the training and testing datasets. The related files are placed in the working directory.

```
library(parallel)
library(doParallel)
```

```
## Loading required package: foreach
## Loading required package: iterators
```

```
library(caret)
```

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

```
library(e1071)
training<-read.csv("pml-training.csv")
testing<-read.csv("pml-testing.csv")
ntrain <- length(training)
```

Now we will get rid of all the columns which are not usable in our predictions. These columns include the first 7 ones which are names and time stamps,. . . . we also delete the emty columns and the oned with number of NAs more than 20% of totall dataset size.

```
#partitioning data
set.seed(100)
inTrain <- createDataPartition(training$classe,p=.7)
#cleaning
trainset <- training[inTrain[[1]],8:160]
testset <- training[-inTrain[[1]],8:160]
testing <- testing[,8:160]
for (i in 152:1){
  if (sum(is.na(trainset[,i])) >= ntrain*0.2 | is.na(mean(trainset[,i],na.rm = TRUE))){
    trainset <- trainset[,-i]
    testset <- testset[,-i]
    testing <- testing[,-i]
  }
}
```

notice that we have two test sets:

- testset : which we created by subsetting (30%) the original training set. We use this for cross validation
- testing : which is provided by the problem, and our task is to predict it.

3) Preprocessing

After cleaning the data, we are left with 52 features (predictors). These 52 features might be correlated. let's do some principle component analysis:

```
#preprocessing
summary(prcomp(trainset[,-53]))
```

```
## Importance of components:
##                             PC1      PC2      PC3      PC4       PC5
## Standard deviation     597.8636 534.9895 470.5745 379.0005 354.68883
## Proportion of Variance   0.2624   0.2101   0.1626   0.1055   0.09237
## Cumulative Proportion    0.2624   0.4726   0.6352   0.7406   0.83298
##                              PC6       PC7       PC8       PC9      PC10
## Standard deviation     255.29968 201.36475 174.48565 158.26241 118.13884
## Proportion of Variance   0.04785   0.02977   0.02235   0.01839   0.01025
## Cumulative Proportion    0.88084   0.91061   0.93296   0.95135   0.96160
##                            PC11    PC12     PC13     PC14     PC15
## Standard deviation     96.83667 89.6318 75.82031 68.35399 62.57110
## Proportion of Variance  0.00688  0.0059  0.00422  0.00343  0.00287
## Cumulative Proportion   0.96848  0.9744  0.97860  0.98203  0.98491
##                            PC16     PC17    PC18     PC19    PC20     PC21
## Standard deviation     56.83343 53.20356 49.5286 48.97429 42.0358 37.66180
## Proportion of Variance  0.00237  0.00208  0.0018  0.00176  0.0013  0.00104
## Cumulative Proportion   0.98728  0.98936  0.9912  0.99292  0.9942  0.99526
##                            PC22    PC23     PC24     PC25     PC26
## Standard deviation     34.88813 33.0258 30.65451 25.58821 23.65613
## Proportion of Variance  0.00089  0.0008  0.00069  0.00048  0.00041
## Cumulative Proportion   0.99615  0.9970  0.99764  0.99812  0.99853
##                            PC27     PC28     PC29     PC30     PC31
## Standard deviation     21.67869 20.86223 17.35979 15.19438 14.02283
## Proportion of Variance  0.00035  0.00032  0.00022  0.00017  0.00014
## Cumulative Proportion   0.99888  0.99920  0.99942  0.99959  0.99973
##                           PC32    PC33    PC34    PC35    PC36    PC37
## Standard deviation     9.97055 7.77641 7.27721 6.67035 6.21030 4.77763
## Proportion of Variance 0.00007 0.00004 0.00004 0.00003 0.00003 0.00002
## Cumulative Proportion  0.99980 0.99985 0.99989 0.99992 0.99995 0.99997
##                           PC38    PC39    PC40  PC41  PC42  PC43   PC44
## Standard deviation     3.80338 3.55293 3.37238 1.954 1.522 1.089 0.4657
## Proportion of Variance 0.00001 0.00001 0.00001 0.000 0.000 0.000 0.0000
## Cumulative Proportion  0.99998 0.99999 0.99999 1.000 1.000 1.000 1.0000
##                          PC45   PC46   PC47   PC48   PC49   PC50   PC51
## Standard deviation     0.3961 0.3595 0.3158 0.2395 0.2027 0.1857 0.1044
## Proportion of Variance 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
## Cumulative Proportion  1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
##                           PC52
## Standard deviation     0.03667
```

```
## Proportion of Variance 0.00000
## Cumulative Proportion  1.00000
```

as the pca shows, the first 10 PCs captures almost %96 percent of the variance. So we toss the remaining PCs. The data is also centered and scaled.

```
train_pre_obj <- preProcess(trainset[,-ncol(trainset)],method = c("center","scale","pca"),pcaComp = 10)
trainset.pre <- predict(train_pre_obj, newdata = trainset[,-ncol(trainset)])
testset.pre <-  predict(train_pre_obj, newdata = testset[ ,-ncol(testset) ])
testing.pre <-  predict(train_pre_obj, newdata = testing[ ,-ncol(testing) ])
train.classe <- trainset$classe
test.classe <- testset$classe
```

4) Tranining Multiple Models

Now we are ready to model our data. Since at this stage we are not sure that what is the best model for our data, We fit different models and later will compare their performance.

```
#training different models:
nc <- detectCores()
fitControl <- trainControl(method="cv", number=3, allowParallel = TRUE)
#Random Forest:
cluster <- makeCluster(nc)
registerDoParallel(cluster)
m.rf <- train(x=trainset.pre, y=train.classe, method="rf", trControl = fitControl)
```

```
## Loading required package: randomForest

## randomForest 4.6-12

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
stopCluster(cluster)
registerDoSEQ()
#linear discriminant analysis:
m.lda <- train(x=trainset.pre, y=train.classe, method="lda")
```

```
## Loading required package: MASS
```

```
#Gradient Boosting Algorithm:
cluster <- makeCluster(nc)
registerDoParallel(cluster)
m.gbm <- train(x=trainset.pre, y=train.classe, method="gbm", trControl = fitControl)
```

```
## Loading required package: gbm

## Loading required package: survival

##
## Attaching package: 'survival'

## The following object is masked from 'package:caret':
##
```

```
##     cluster
## Loading required package: splines
## Loaded gbm 2.1.1
## Loading required package: plyr
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1        1.6094             nan     0.1000    0.0989
##      2        1.5506             nan     0.1000    0.0856
##      3        1.4992             nan     0.1000    0.0619
##      4        1.4627             nan     0.1000    0.0483
##      5        1.4330             nan     0.1000    0.0383
##      6        1.4097             nan     0.1000    0.0406
##      7        1.3850             nan     0.1000    0.0323
##      8        1.3652             nan     0.1000    0.0325
##      9        1.3452             nan     0.1000    0.0281
##     10        1.3279             nan     0.1000    0.0299
##     20        1.2064             nan     0.1000    0.0123
##     40        1.0730             nan     0.1000    0.0047
##     60        0.9881             nan     0.1000    0.0042
##     80        0.9208             nan     0.1000    0.0038
##    100        0.8649             nan     0.1000    0.0023
##    120        0.8150             nan     0.1000    0.0028
##    140        0.7742             nan     0.1000    0.0026
##    150        0.7518             nan     0.1000    0.0024
```

```r
stopCluster(cluster)
registerDoSEQ()
#Support Vector Machine:
m.svm <- svm(x=trainset.pre, y=train.classe, method="svm")
```

In the above code, note that we used parallel computing for rf and gbm methods. The reason is training these models are computationally hard and takes a long time. So by using all the cores of the cpu we can reduce the running time.

5) Comparing Performance of the Models

let's see which model is doing better on the preprocessed **testset**:

```r
#performance:
rf.predicted <- predict(m.rf,testset.pre)
lda.predicted <- predict(m.lda,testset.pre)
gbm.predicted <- predict(m.gbm,testset.pre)
svm.predicted <- predict(m.svm,testset.pre)
confusionMatrix(test.classe,rf.predicted)$overall
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##   9.553101e-01   9.434552e-01   9.497158e-01   9.604472e-01   2.881903e-01
## AccuracyPValue  McnemarPValue
##   0.000000e+00   2.361257e-08
```

```r
confusionMatrix(test.classe,lda.predicted)$overall
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##   4.276975e-01   2.680909e-01   4.150108e-01   4.404560e-01   3.717927e-01
```

```
## AccuracyPValue  McnemarPValue
##   7.986912e-19   7.974398e-66
```

```
confusionMatrix(test.classe,gbm.predicted)$overall
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##   7.208156e-01   6.462325e-01   7.091619e-01   7.322501e-01   3.029737e-01
## AccuracyPValue  McnemarPValue
##   0.000000e+00   1.027620e-34
```

```
confusionMatrix(test.classe,svm.predicted)$overall
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##   8.083263e-01   7.565412e-01   7.980329e-01   8.183131e-01   3.211555e-01
## AccuracyPValue  McnemarPValue
##   0.000000e+00   4.411658e-63
```

Well, rf's performance is quite impresssive with %95 accuracy. The runner-ups are gbm, and svm. The lda method (with default setting) is not suitable for our data.

6) stacking predictors

Now, that we know rf, gbm, and svm are doing good on our testset, we can try stacking up these methods to have even better performance:

```
#stacking predictors
combined<-cbind(rf.predicted,gbm.predicted,svm.predicted)
mc<-train(x=combined,y=test.classe,method="rf", trControl = fitControl)
```

```
## note: only 2 unique complexity parameters in default grid. Truncating the grid to 2 .
```

```
combined.predicted<-predict(mc,newdata=combined)
confusionMatrix(test.classe,combined.predicted)$overall
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##   9.558199e-01   9.441008e-01   9.502541e-01   9.609279e-01   2.881903e-01
## AccuracyPValue  McnemarPValue
##   0.000000e+00   2.064830e-08
```

the confusion matrix shows that the combined model is not better than the original rf on the testset. But no harm in keeping it.

7) Results:

The following code will apply the combined model(rf+gbm+svm) to the testing set which consists of 20 instances:

```
#output
newdata<-cbind(predict(m.rf,testing.pre),predict(m.gbm,testing.pre),predict(m.svm,testing.pre))
output<-predict(mc,newdata = newdata)
print(output)
```

```
##  [1] B A A A A E D B A A A C B A E E A B B B
## Levels: A B C D E
```