Exercising Right: Using sensors to ensure proper form

michael downs

synopsis

Many people measure the amount they exercise (miles run, weights lifted, etc.). Few measure the quality of their exercise, their "form". Using data from sensors placed on the belts, forearms, arms, and dumbells of six participants, I used four algorithms to classify barbell lifts. Classifications range from correct form (class A) to four incorrect forms including throwing the elbows (class B), throwing the hips (class E), etc..

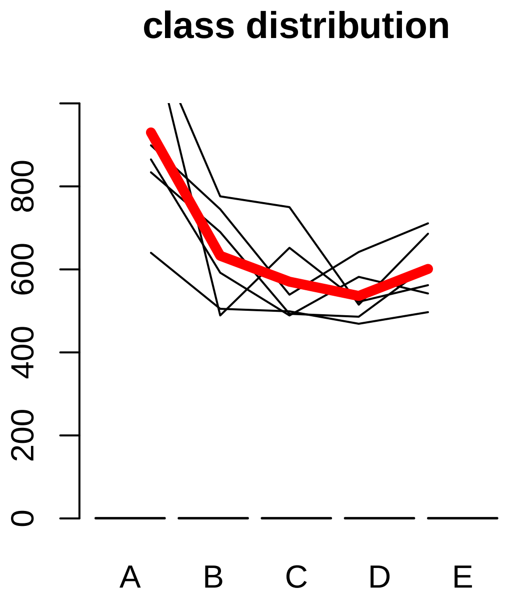
I found random forest (rf) and boosting (gbm) to be effective at classifying activites based on sensor output. Both algorithms achieved 97%+ sensitivity and 99%+ specificity scores on cross validation. (The random forest model scored 20/20 on the project test set.) Further, both models maintained high cross validation scores: 1. across four train/test datasets that were constructed using different methodologies and 2. across a range of variables from over 100 down to as small as 10. K-nearest neighbors (knn) and basic regression trees (rpart) were less successful with sensitivity scores of $\approx$ 88% and $\approx$ 53% on cross validation, respectively.

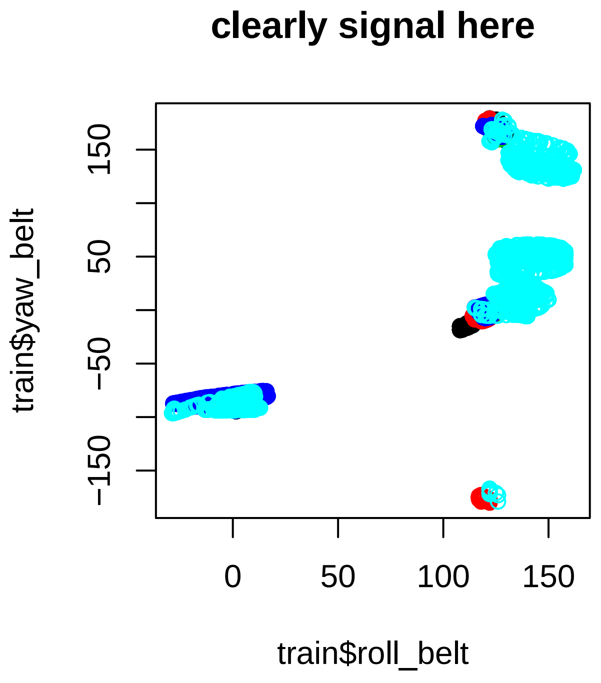
I walk thru model construction below including use of cross validation, estimates of out-of-sample error rates and the rationale for decisions made along the way.

the model

getting data

The training data set included 160 variables and about 20,000 records. Each record contained a subset of sensor readings sampled during the exercise period for each of the six particpants. Basic exploratory analysis showed that 67% of the columns contained 97% NA values which would have to be cleaned. That analysis also identified a roughly even distribution of observations among the three classes and signficant signal that could be seen when fields were plotted against each other (below).



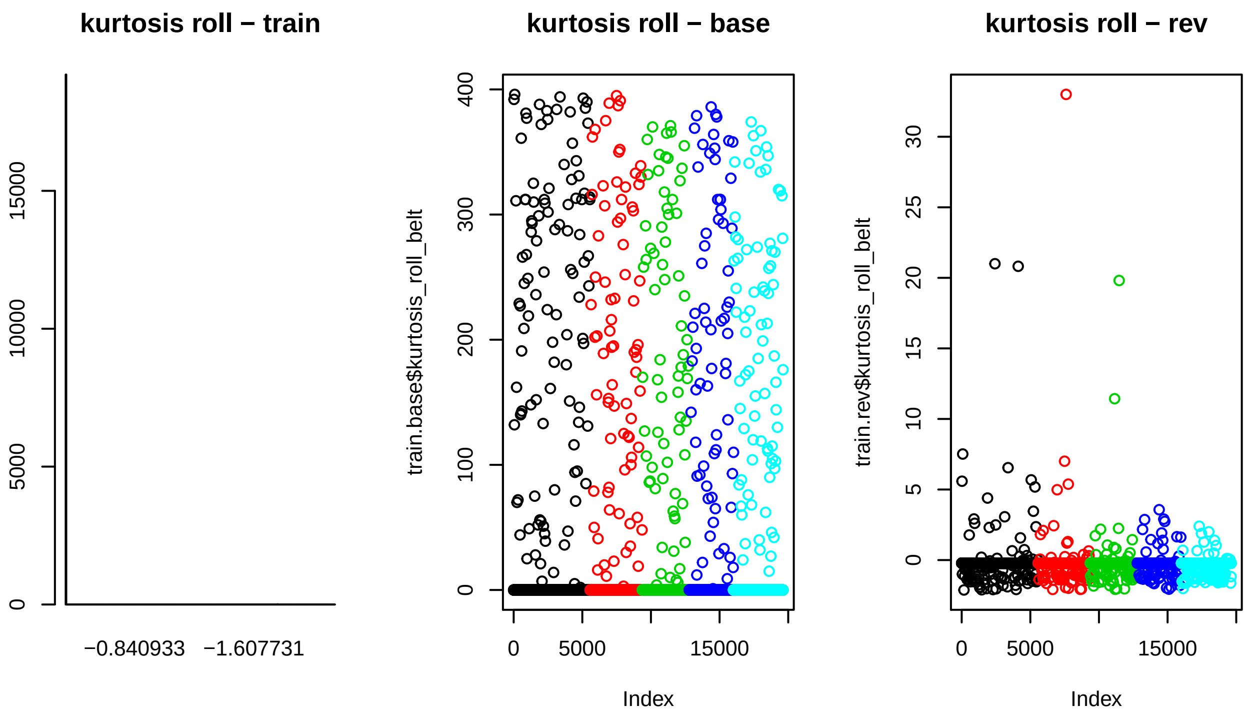


cleaning data

I generated four train/test datasets. While I deleted timestamp and window columns for all datasets (to the extent exercisers are following a script, these columns provide information that wouldn't be available otherwise), each dataset was developed according to its own methodology. Highlights include: 1. $*train.base:* I replaced NA's by the means for their respective columns. Given most algorithms use column variance for predictions, this approach elimnates NAs while not changing column variance. 2. $*train.nzv:* I deleted rows with near zero varaiance (nzv()) then set NAs in remaining columns to the respective column mean. 3. $*train.na:* I deleted columns with greater than 97% NA. 4. $*train.rev:* I converted factor to numeric using $levels(train.rev[, i]))[train.rev[, i]]$, then set any remaining NAs to the respective column means.

The effect of these changes and others not covered is illusrated in the kurtosis graphic below. From left to right I show how how the data was imported, what the data looke like using for train.base and what the data looked like for train.rev.

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

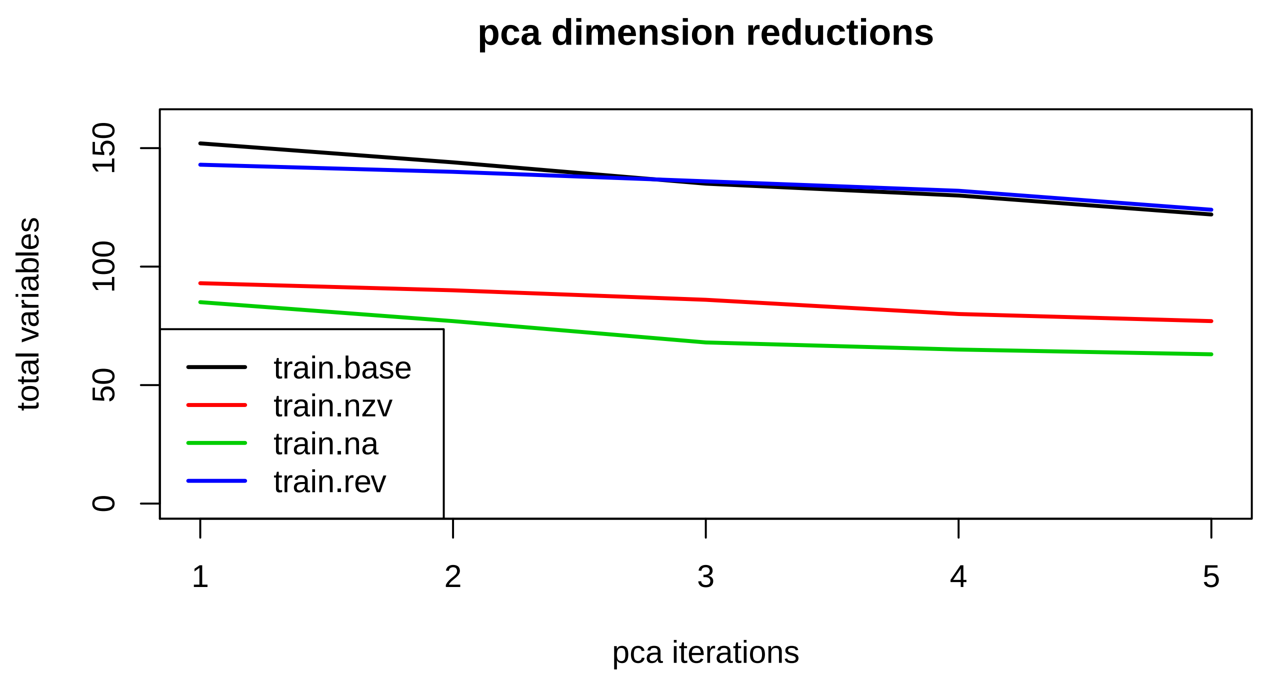


pca pre-processing

I performed feature selection in two steps. The first was pca analysis. I based my approach on the lecture where pca was used only for highly correlated variables (rather than to reduce dimensions and regularize all variables). Specifically, I looped over the data sets five times replacing highly correlated variables with their principal components. As the approach was a bit involved, I've included the code below. This approach reduced variables between 25% and 30% across data sets.

```
dset=list(ds=list("train.base","train.nzv","train.na","train.rev"),
          trnx=list(train.base,train.nzv,train.na,train.rev),
          trny=train$classe)
```

```r
par(mfrow=c(1,1))
plot(1:5,type="n",main="pca dimension reductions",ylim=c(0,160),xlab="pca iterations",ylab="total varial

for(k in 1:length(dset$trnx)){
     target=dset$trnx[[k]][,c(2:(dim(dset$trnx[[k]])[2]-1))]
     gph.line=NULL
     cat("********** new file: ",dset$ds[[k]],"\n")
     for(i in 1:5){ # was 7
          cat("1. start: dim(target)",dim(target),"\n")
          gph.line[i]=dim(target)[2]
          cor.decs=matrix(NA,dim(target)[2],2)
          M=abs(cor(target)) # find predictors w/ high correlation
          diag(M)=0 # set cor for vars w/ themselves to 0

          for(j in 1:dim(target)[2]){
               cor.decs[j,1]=j
               cor.decs[j,2]=length(which(M[,j]>max(0.5,(1-0.1*i)),arr.ind=T))
          }
          clean=target[,c(which(M[,which.max(cor.decs[,2])]>max(0.5,(1-0.1*i)),arr.ind=T),
                          which.max(cor.decs[,2]))]

          cat("2. consolidating: ",cor.decs[which.max(cor.decs[,2]),][2]+1,"\n")
          comps=prcomp(clean);print(summary(comps))

          target=target[,-c(which(M[,which.max(cor.decs[,2])]>max(0.5,(1-0.1*i)),arr.ind=T),
                            which.max(cor.decs[,2]))]

          tmp=as.matrix(summary(comps)$importance);pc.cut=which(tmp[3,]>0.95)[1]
          comps=comps$x[,1:pc.cut];comps=as.data.frame(comps)
          for(z in 1:pc.cut){
               colnames(comps)[z]=c(sprintf("%s_%s_%s_%s","iter",i,"pc",z))}
          cat("3. into: ",dim(comps)[2],"\n")

          target=cbind(target,comps)
     }
     dset$trnx[[k]]=target
     lines(1:5,gph.line,col=k,lwd=2)
}
legend("bottomleft",c("train.base  ","train.nzv  ","train.na  ","train.rev  "),col=c(1,2,3,4),lwd=2)
```



```
rm(tmp,clean,target)
for(i in 1:4){dset$trnx[[i]]$user_name=train$user_name
              dset$trnx[[i]]$classe=train$classe}
```

models

I built a two loop model training structure to iterate thru the datasets outlined above and a set of core algorithms including regression trees (rpart), k-nearest neighbors (knn), random forest (rf) and boosting (gbm). Each iteration of the internal loop fit the model, made predictions on a cross validation set and stored off the fit for subsequent processing.

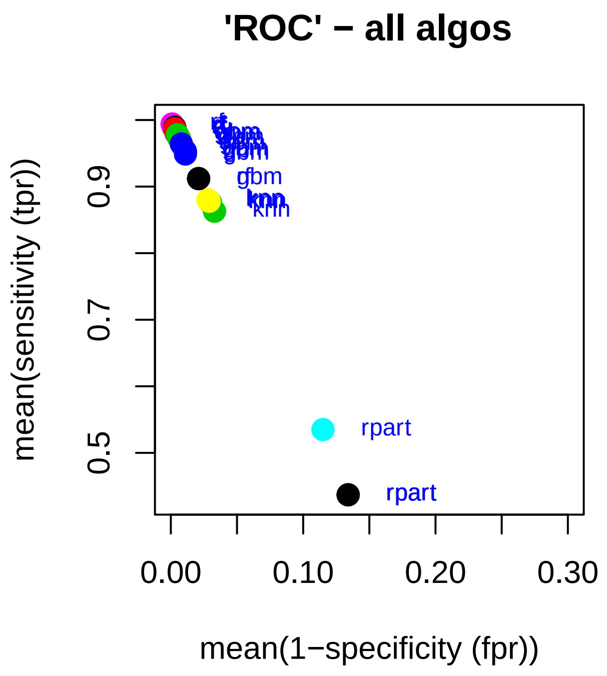
Getting thru the dataset required careful resource management. While data cleaning / pca eliminated $\frac{1}{3}$ to $\frac{1}{2}$ of the columns, early iterations from 70 to 115 columns. Accordingly, I trimmed train/test database by up to 50%. I also enabled parallel processing.

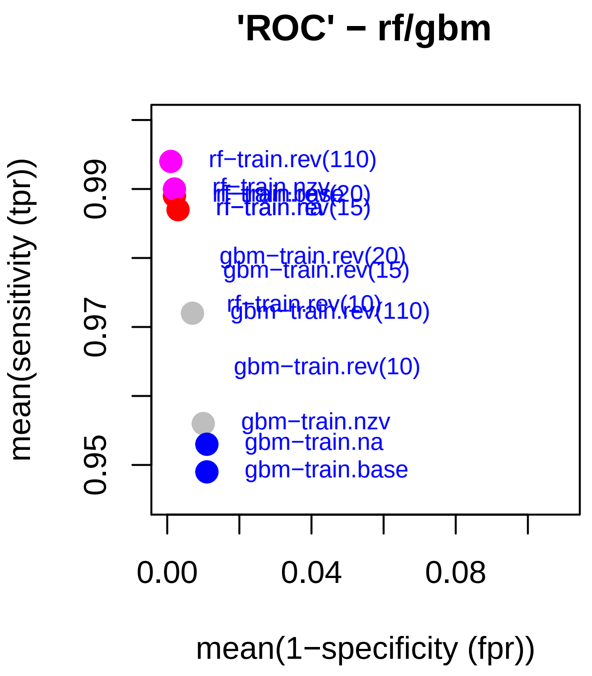
Beyond cross vaidation performed within the Caret function which showed 95-99% accuracy (1%-5% OOB error rates), I separately **cross validated** each dataset / algorithm combination using a 70 / 30 train / holdout test set cross validation.

As the approach was fairly involved, I've provided code below. Below that are the cross validated prediction sensitivity and specificity for each model. The left graphic shows all the models inclding the outperformance of the rf and gbm models. The right looks more closely at the rf and gbm results.

```
# controller for algo iterations
algos=list(algo=list("rpart","rf","knn","gbm")) ## logistic, lda failed.

# structures to hold rf fits
fset=list(data=list("train.base","train.nzv","train.na","train.rev"),
          rpart=list(NA,NA,NA,NA),
          rf=list(NA,NA,NA,NA),
          knn=list(NA,NA,NA,NA),
          gbm=list(NA,NA,NA,NA))
```

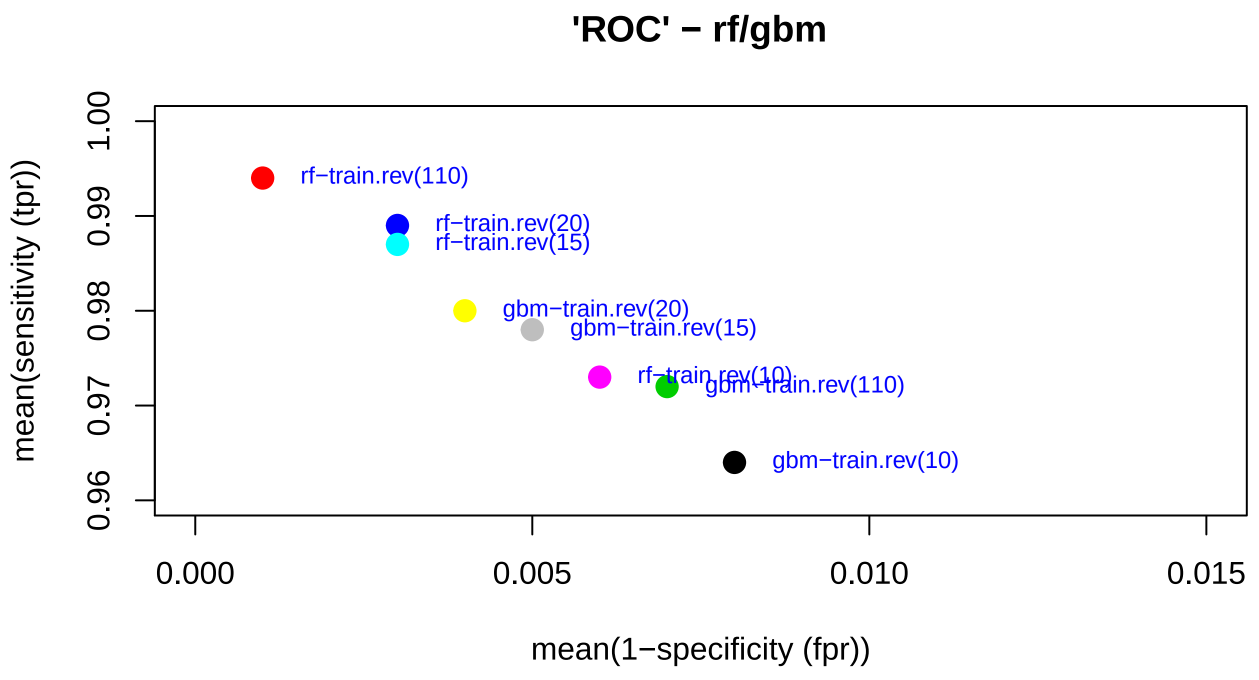
```r
# process data sets, algos, folds
# rm(res,res.mstr)

## enable parallel processing
library(parallel); library(doParallel)
registerDoParallel(clust <- makeForkCluster(detectCores()))

for(i in 1:length(dset$ds)){
     # cut data in half
     train.cv=dset$trnx[[i]][train$user_name==unique(train$user_name)[1] |
                                  train$user_name==unique(train$user_name)[2] |
                                  train$user_name==unique(train$user_name)[3],]

     for(k in 1:length(algos$algo)){
               # train and test sets
               set.seed(123)
               inTrain=createDataPartition(y=train.cv$classe,p=0.7,list=FALSE)
               training=train.cv[inTrain,]
               testing=train.cv[-inTrain,]

               cat("******* FITTING: algo:",algos$algo[[k]],", data dim: ",dim(training),"\n")
               # fit and predict
               fit=train(classe~.,training,method=algos$algo[[k]])

               # store and print fit
               fset[[k+1]][[i]]=fit;print(fit)

               # predict
               preds=predict(fit,testing)

               # create output table
               res=as.data.frame(t(as.matrix(c(algos$algo[[k]],dset$ds[[i]]))))
               colnames(res)=c("algo","data set")
               sense=round(t(confusionMatrix(preds,testing$classe)$byClass[,1]),3)
               spec=round(t(confusionMatrix(preds,testing$classe)$byClass[,2]),3)
               res=cbind(res,sense,round(mean(sense),3),spec,round(mean(spec),3))
               if(exists("res.mstr")){res.mstr=rbind(res.mstr,res)}else{res.mstr=res}
               print(res.mstr)
     }
}
stopCluster(clust)
```





top 20 vars

Finally, I re-ran the rf and gbm algorithms using the train.rev database and just their top variables. I compared four sets of results for different numbers of top variables including 100, 20, 15 and 10. Interestingly the performance did not degrade significantly for either algorithm. While rf moved from 99.5% to 97.5% accuracy between the 100 variable and 10 variable models, gbm actually increased its accuracy from 97% to 98% using as variables decreased from 100 to 15.


'ROC' − rf/gbm
mean(sensitivity (tpr))
mean(1−specificity (fpr))
rf−train.rev(110)
rf−train.rev(20)
rf−train.rev(15)
gbm−train.rev(20)
gbm−train.rev(15)
rf−train.rev(10)
gbm−train.rev(110)
gbm−train.rev(10)