Tarleton State University

Data Mining 1

Human Activity Recognition with Smartphone Data

Authors:
David Ebert
Parker Rider

December 8, 2015

1 Introduction

With the increase in the number of smartphone users in the past several years, it is easier than ever for people to capture data while humans perform day-to-day activities. For example, Google Fit tracks and classifies users' activities throughout the day in order to estimate distance traveled, calories burned, and active time. [1] Other wearable devices have found a market selling devices which track human activity and generate data in order to promote good health.

In this following paper we set out to understand the effectiveness of applying classification techniques to a data set generated by cell phone users. We aim to test the effectiveness of classification techniques and apply some of our own questions to the data set.

2 Smartphone Data Set

We retrieved our data from the UCI Machine Learning Repository. The data is titled "Human Activity Recognition Using Smartphones Data set." [2] The research was conducted with 30 participants between the ages of 19 and 48 years old. Participants completed the 6 following activities while wearing a smartphone with an accelerometer and gyroscope, which collected data. The six activities were numbered 1 to 6 as follows:

1. Walking
2. Walking upstairs
3. Walking downstairs
4. Sitting
5. Standing
6. Laying

[1] For a discussion of several of these devices and their effectiveness, see `http://well.blogs.nytimes.com/2015/11/16/assessing-the-fitness-of-wearable-tech/`

[2] The data set is available from `https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones`

The accelerometer and gyroscope "captured 3-axial linear acceleration and 3-axial angular velocity at a constant rate of 50Hz" and data "were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 sec and 50% overlap (128 readings/window). The sensor acceleration signal, which has gravitational and body motion components, was separated using a Butterworth low-pass filter into body acceleration and gravity. The gravitational force is assumed to have only low frequency components, therefore a filter with 0.3 Hz cutoff frequency was used. From each window, a vector of features was obtained by calculating variables from the time and frequency domain."

There are a total of 561 "attributes" or X variables. The classification factor is our Y. The data had already been randomly split, with 21 walkers (70%) comprising the training data and the other 9 forming the test data.

The initial goal of our project was to accurately determine the classification activity of the volunteers in the testing set based on the attributes from the training data. The smartphones supplied 10,299 rows of records, which averages to 340 observations for each of the 30 participants. The training set indicated by the data includes all observations from 21 of the participants, while the testing set includes the data from the remaining 9 volunteers. A target vector of length 10,299 indicates which of the 6 activities are performed at each observation, and an additional target vector indicates which of the 30 participants created each observation.

3 Predicting Activity

3.1 Classification Methods

3.1.1 Trees

For our first attempt at classifying activity we used `rpart` trees, shown in table 1, which yielded an accuracy of 84%. Though the confusion matrix is far from perfect, there are two 3×3 submatrices of zeros on the off-diagonal, indicating that our 6-node tree distinguishes between walking and inactive states with 100% accuracy. Further investigation shows that even a 1-node tree (see figure 3) achieves 99.9% accuracy! Figures 1 and 2 hint that the data are linearly separable over this binary classification, but not necessarily the 6-case classification.

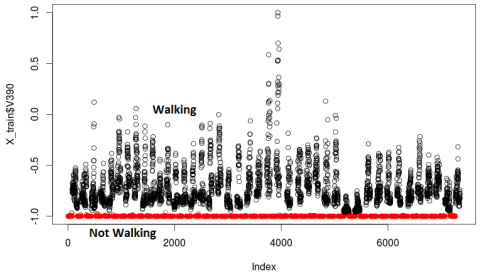


Figure 1: The walking vs inactive classification is linearly separable

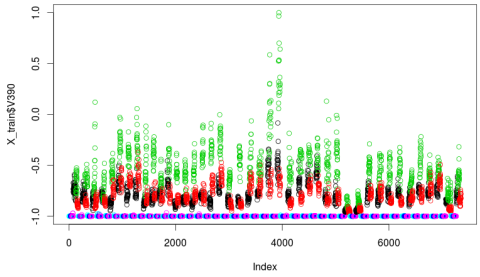
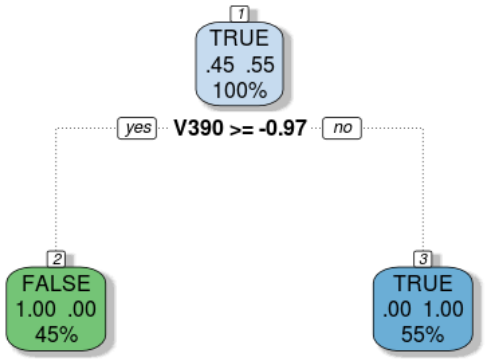


Figure 2: The 6-state classification is not linearly separable

Table 1: `rpart` tree confusion matrix, with submatrices of 0 on the off diagonals. Accuracy: 84.0%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	430	55	11	0	0	0
	2	61	402	8	0	0	0
	3	40	98	282	0	0	0
	4	0	0	0	400	91	0
	5	0	0	0	107	425	0
	6	0	0	0	0	0	537

Figure 3: A one-node `rpart` tree achieves 99.9% accuracy distinguishing between walking and inactive states.



3.1.2 Naive Bayes

With an accuracy of only 77% (see table 2), the `naiveBayes` model yielded the worst accuracy of all the models used. In numerous cases the model confused walking upstairs with the three inactive states.

Table 2: Naive Bayes confusion matrix. Accuracy: 77%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	416	38	42	0	0	0
	2	9	451	11	0	0	0
	3	80	83	257	0	0	0
	4	0	7	0	369	110	5
	5	0	15	0	55	454	8
	6	0	3	0	210	0	324

3.1.3 Support Vector Machines

A basic implementation of support vector machine using the built in one-against-rest approach of the `e1071` package yielded a classificatin accuracy of 95.5% as shown in table 3. Seeming to hold much promise for high accuracy, we made an effort to tune the SVM model using a grid search. So far the best result is returned when `cost` $= 10000$ and `gamma` $= 10^{-4}$. This is summarized in table 4. Though more work is required to tune the SVM model further,

this approach produced a higher accuracy than all other models attempted so far. Interestingly, even the tuned SVM model ran into one case in which an inactive state was confused with a walking state. We might avoid this problem by using a broad model to determine between walking and inactive states, then two narrower models to distinguish among walking and inactive states.

Table 3: SVM confusion matrix. Accuracy: 95.1%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	482	6	8	0	0	0
	2	14	456	1	0	0	0
	3	6	29	385	0	0	0
	4	0	1	0	441	47	2
	5	0	0	0	29	503	0
	6	0	0	0	0	0	537

Table 4: Tuned SVM confusion matrix. Accuracy: 96.3%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	494	0	2	0	0	0
	2	16	453	2	0	0	0
	3	5	18	397	0	0	0
	4	0	1	0	437	53	0
	5	0	0	0	12	520	0
	6	0	0	0	0	0	537

3.1.4 k-Nearest Neighbors

In order to run the k-nearest neighbors algorithm, we first standardized the data. Since the data was already divided prior to our manipulation with it, the training and test data were standardized separately using the following procedure. First, we used the `apply` function to determine the mean of the columns of the set. Second, we made an $n \times 1$ vector with the mean for each element. Third, a similar approach was used to store the standard deviations

in an $n \times 1$ vector. Finally, we subtracted the mean vector from the columns of the data and then divided the columns by the standard deviation vector.

Upon standardizing the data, we first ran the k-nearest neighbors algorithm with $k = 1$. The resulting accuracy was 85.7%. Next, we used a `for` loop to recover the optimal k for K-nearest neighbors, which yielded $k = 16$, as shown in figure 4. Using the optimal k of 16, we reran `knn`, resulting in an accuracy of 89.8%, as shown in table 5.

Figure 4: Accuracy of KNN plotted against k

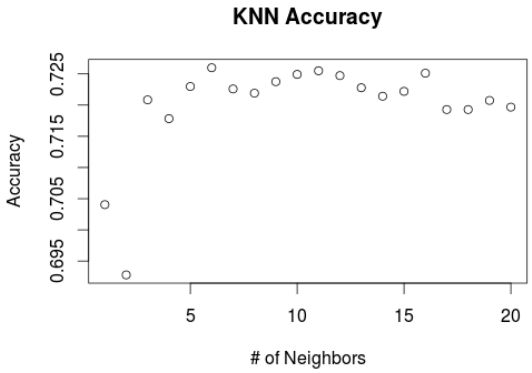


Table 5: KNN confusion matrix. Accuracy: 89.8%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	481	3	12	0	0	0
	2	42	427	2	0	0	0
	3	40	49	331	0	0	0
	4	0	1	0	393	94	3
	5	0	0	0	25	507	0
	6	0	0	0	12	11	514

3.1.5 Weighted k-Nearest Neighbors

For the weighted k-nearest neighbors algorithm, a `for` loop returned an optimal k of 10 and an optimal kernel of *triweight*. Then, using `y_train` as the formula object, the standardized training set as the training data, and the

standardized testing set as the testing data, the confusion matrix returned an accuracy of 90%, as shown in table 6.

Table 6: Weighted KNN confusion matrix; $k = 65$. Accuracy: 90.0%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	476	2	18	0	0	0
	2	31	438	2	0	0	0
	3	40	45	335	0	0	0
	4	0	1	0	388	99	3
	5	0	0	0	30	502	0
	6	0	0	0	10	11	516

3.1.6 Neural Networks

We made our first neural network model using one node, as a baseline. Unsurprisingly, this yielded an accuracy of only 17.8%, which is roughly equivalent to randomly guessing for a 6-class problem. Next, we employed a `for` loop to determine the optimal number of nodes to use for this data set. As shown in figure 5, the loop returned an optimal number of 18 nodes; however, based on the plot generated from the outputs, we decided to use 9 nodes since there is little improvement in accuracy. As few as six nodes may have been sufficient, but 9 proved to be a good compromise. The accuracy from using 9 nodes for the neural network is 93.1%, as shown in table 7, and the area under the ROC curve is 0.9042.

Figure 5: Accuracy of neural network plotted against number of nodes

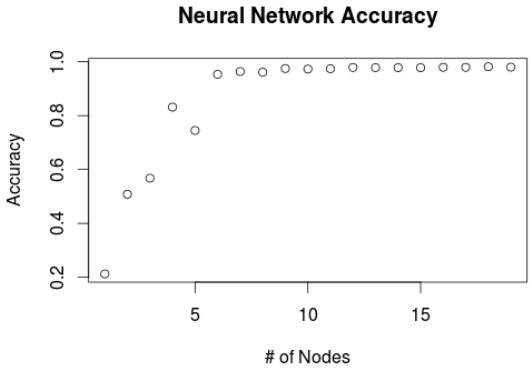


Table 7: Neural network confusion matrix; $k = 9$. Accuracy: 93.6%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	463	4	29	0	0	0
	2	34	425	12	0	0	0
	3	17	14	389	0	0	0
	4	0	0	0	433	58	0
	5	0	0	0	18	513	1
	6	0	0	0	0	0	537

3.1.7 Random Forests

For our random forests, we made our `forestmodel` using the `randomForest` function using `y_train` as my matrix of predictors and `X_train` as the data frame containing the variables in the model. Next, we utilized the `predict` function to determine the predictions for the testing set. Finally, we applied the `confmatrix` to find the accuracy of the predictions, which is 92.4%, as shown in table 8. Moreover, the area under the ROC curve is 0.9372.

Table 8: Random forest confusion matrix; $k = 9$. Accuracy: 92.4%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	481	8	7	0	0	0
	2	40	425	6	0	0	0
	3	22	41	357	0	0	0
	4	0	0	0	437	54	0
	5	0	0	0	45	487	0
	6	0	0	0	0	0	537

3.2 Summary

A summary of the accuracies of the models discussed in this section is provided in table 9. Overall the tuned SVM model performed best, followed closely by neural networks.

Table 9: Summary of model accuracies for 6-case activity prediction

Model	**Accuracy**
Tree	0.840
Bayes	0.770
SVM	0.951
Tuned SVM	**0.963**
KNN	0.900
Weighted KNN	0.901
Neural Network	0.937

4 Predicting Walkers

In addition to a vector containing the activity performed at each moment, the data set includes a second target vector indicating which of the 30 walkers performed each activity. In this section, we look at the problem of identifying participants based on their walking data.

4.1 Method

To begin, we combined all data into one data frame and removed all inactive data from the set. The resulting data includes an average of over 300 observations for each of the 30 participants while they are walking. Next, to test a model's accuracy in distinguishing among $n \leq 30$ walkers, we repeated the following i times: select n walkers at random, split the roughly $300n$ observations from all of the walkers at random into 70% training data, and calculate the model's test accuracy. The model's effectiveness was taken to be the average accuracy of the i iterations. Ideally, we would like i to be large to avoid variability, but in practice here we settled for $i = 8$. A summary of the procedure outlined here is shown in figure 6. Figure 7 shows an example confusion matrix with $n = 10$.

Figure 6: The algorithm used to determine a model's accuracy for classifying $n \leq 29$ walkers.

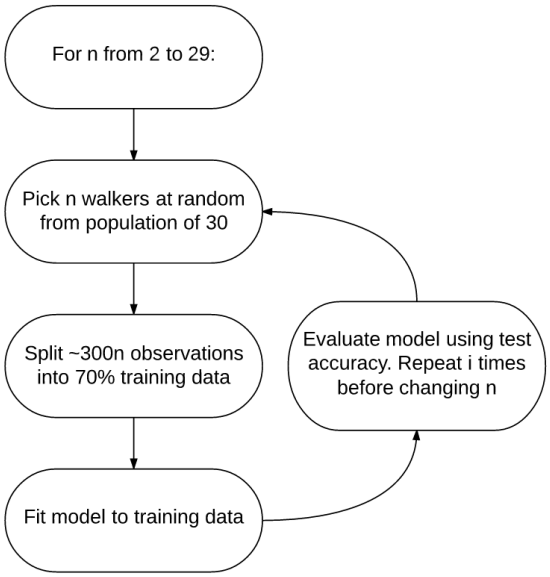


Figure 7: An example confusion matrix distinguishing among 10 random walkers, based on an `rpart` tree model

Confusion Matrix: 10 Walkers

Actual Walker \ Predicted Walker	2	6	8	11	14	16	17	23	25	27
2	44	0	0	2	0	0	0	3	2	0
6	0	38	5	0	0	0	1	0	0	2
8	0	4	30	0	1	0	0	4	0	0
11	2	0	0	47	0	0	0	0	0	0
14	0	0	6	0	37	0	0	0	0	0
16	1	0	0	0	0	32	11	0	0	1
17	0	0	1	0	1	6	37	3	0	1
23	0	0	3	0	1	0	0	41	0	2
25	1	1	0	0	0	1	1	0	54	0
27	0	0	0	0	0	2	3	0	0	39

Accuracy: 84.7%

4.2 Results

The above algorithm was performed for n from 2 to 29 using `rpart` trees and support vector machines. As figure 8 shows, both models decrease in accuracy as the number of walkers increases. The accuracy of `rpart` trees decreases to just under 50%, whereas the accuracy of the SVM models remain above 94% even with all 30 walkers included. Both graphs show some variablity due to the fact that the values were generated from the average of only $i = 8$ iterations. Key values from figure 8 are summarized in table 10.

Figure 8: The accuracy of `rpart` and SVM models, applied to n walkers from 1 to 30.

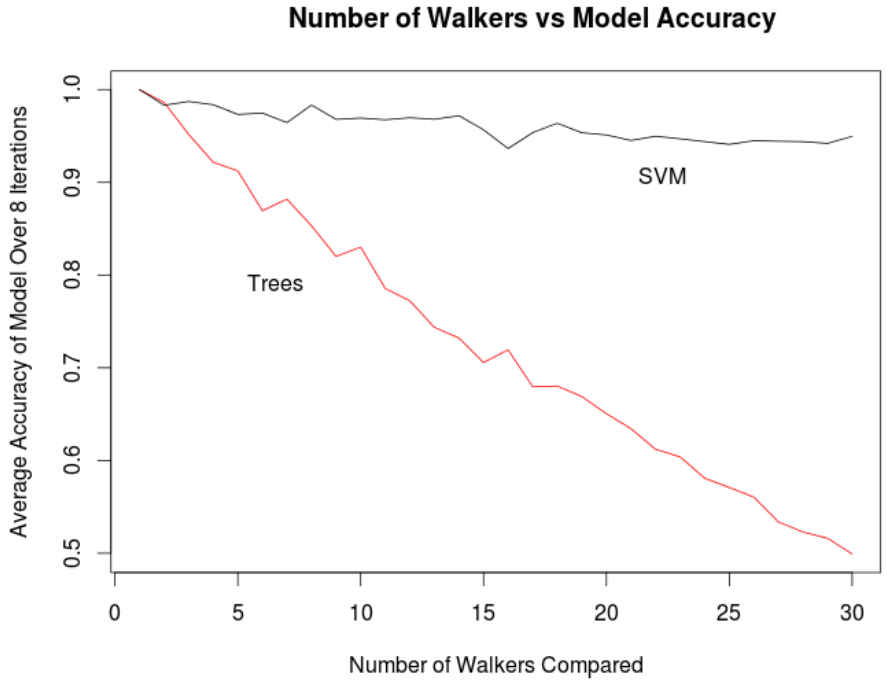


Table 10: Average model accuracy of n walkers over 8 iterations

n	Tree	SVM
5	0.912	0.973
10	0.830	0.969
15	0.705	0.956
20	0.650	0.951
25	0.570	0.940
30	**0.499**	**0.949**

5 Dimensionality Reduction

Though the data set has 561 features, only 6 attributes are necessary to produce the tree in table 1 with an accuracy of 84%. Moreover, as we saw in

figure 3, a tree with just *one* node is sufficient to classify nearly all walking and inactive states. Clearly we should investigate methods for simplifying the data, thereby speeding up computations while maintaining the data's structure and integrity.

5.1 Method

From the `caret` package, the `preProcess` function was used to apply centering, scaling, and principal component analysis to the data. The data was thereby simplified to 102 principal components maintaining 95% of the variability of the original data. A scree plot is given in figure 9.

The first two components (accounting for some 67% of the variability of the data) are plotted in figure 10. The plot shows separation between the walking (lower left) and inactive (upper right) states. Moreover, the 6 activities form identifiable clusters.

Figure 9: Scree plot of principal component analysis of training data

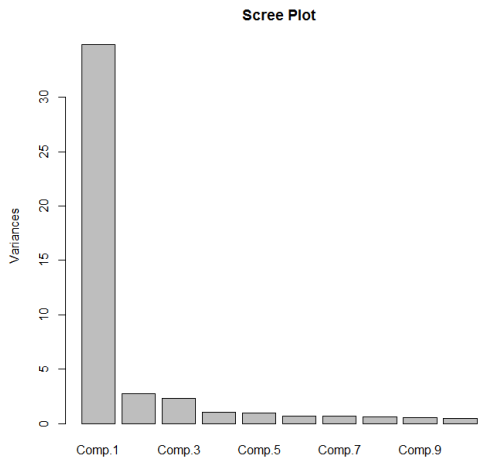
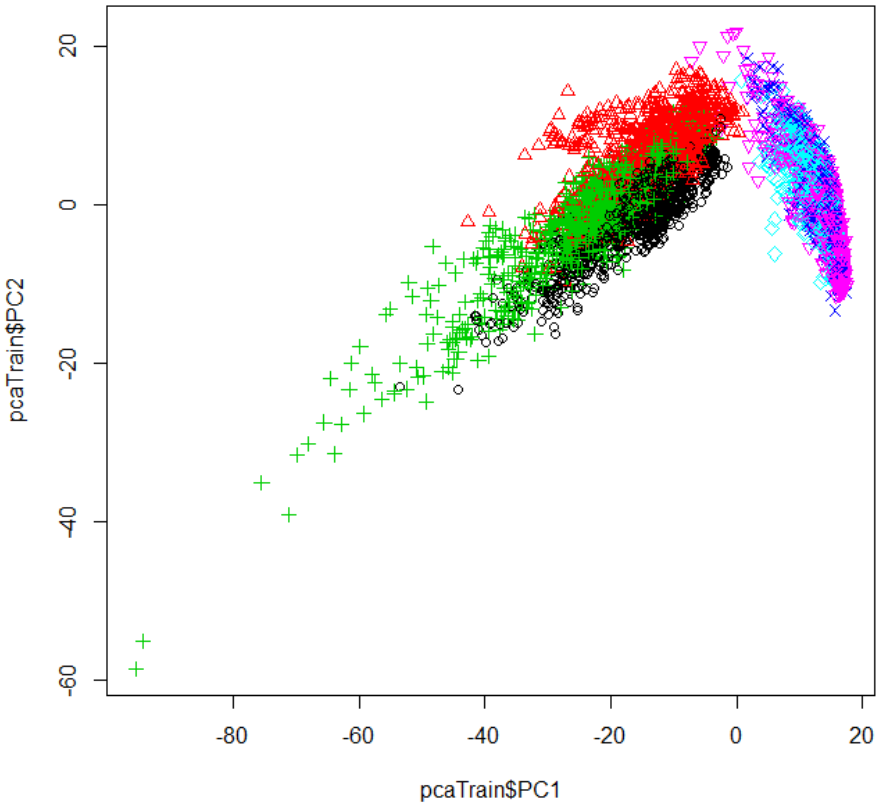


Figure 10: Two principle components of training data; points colored according to class



5.2 Results

Tree and SVM models were used to find training and test accuracy using the preprocessed data. The tree model yielded a diminished accuracy of 73.9%, while the SVM model yielded a new accuracy of 93.7%. The confusion matrices for both reveal that most of the loss of accuracy resulted from misclassification of inactive states, suggesting that the variability inherent in the walking cases was captured more readily by principle component analysis that of the inactive states.

Table 11: Tree confusion matrix. Accuracy: 73.9%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	453	8	35	0	0	0
	2	115	327	29	0	0	0
	3	135	49	236	0	0	0
	4	0	2	0	234	254	1
	5	0	0	0	71	461	0
	6	0	1	0	91	2	443

Table 12: SVM confusion matrix. Accuracy: 93.7%

		Predicted Walker					
		1	**2**	**3**	**4**	**5**	**6**
Actual Walker	**1**	475	0	21	0	0	0
	2	31	419	21	0	0	0
	3	2	10	408	0	0	0
	4	0	1	4	431	51	4
	5	1	0	1	51	496	0
	6	0	0	3	4	0	534

6 Conclusions

So far our work supports the following conclusions:

- Classifying walking versus inactive states is trivial.
- Among the models for predicting activity, the tuned SVM approach yielded the best results, followed closely by neural networks.
- Additionally, SVM's appear to be effective in predicting walkers' identity. The high accuracy suggests that people have an almost uniquely identifiable walking "fingerprint."
- Many (if not all) models do not require anywhere close to 561 attributes for classification. Early results with principal component analysis indicate the data can be reduced to 100 or fewer attributes without sacrificing much accuracy.

7 Further Research

With more work this research may be extended to the following areas:

- Identify the most predictive accelerometer and gyroscope attributes in order to tune models and increase efficiency. Since many phones do not have a gyroscope, consider using only the accelerometer data.
- Develop models which focus only on classifying the 3 walking (or 3 inactive) states to achieve higher accuracy.
- Apply neural networks to the problem of identifying walkers to see if they are as successful as SVM's
- Expand the research to more activities and participants.
- Develop an implementation for finding a person's walking "fingerprint." Use it to track down bad guys, commit the perfect crime, or write the next great American novel.
- Cross validate the models by splitting the data differently. Find out how much the classification accuracy improves when their data is included in the training data.

Appendix: R Source Code

```
# HARmodels.R
# Updated 5 December 2015
# David Ebert

####################
### Import data ###
####################
X_train <- read.table("~/Desktop/HAR/Data/X_train.txt", quote="\"",
                      comment.char="")
X_test <- read.table("~/Desktop/HAR/Data/X_test.txt", quote="\"",
                     comment.char="")

y_train <- read.table("~/Desktop/HAR/Data/y_train.txt", quote="\"",
                      comment.char="")
y_test <- read.table("~/Desktop/HAR/Data/y_test.txt", quote="\"",
                     comment.char="")


##################
####  TREES #####
##################
library(rpart)
library(rattle)

# Begin by importing X_test, X_train, y_test, and y_train
# from the UCI repository.
# http://archive.ics.uci.edu/ml/machine-learning-databases/00240/
# UCI HAR Dataset.zips


##########################################
# Case 1: Active and passive states Tree #
##########################################
#set new y vectors to be TRUE/FALSE levels instead of 6 levels
bin.y_train = as.factor(as.numeric(y_train$V1)>3)
```

```
bin.y_test = as.factor(as.numeric(y_test$V1)>3)

#fit tre
bin.HARtree = rpart(bin.y_train~., data = X_train)
print(bin.HARtree)
fancyRpartPlot(bin.HARtree)

#training data
bin.train.tree.HAR.pred = predict(bin.HARtree, newdata = X_train)
bin.train.tree.HAR.pred = apply(bin.train.tree.HAR.pred, 1, which.ma
confmatrix(bin.y_train, bin.train.tree.HAR.pred)
#Training accuracy: 99.9%

#test data
bin.test.tree.HAR.pred = predict(bin.HARtree, newdata = X_test)
bin.test.tree.HAR.pred = apply(bin.test.tree.HAR.pred, 1, which.max)
confmatrix(bin.y_test, bin.test.tree.HAR.pred)
#Test accuracy: 100%


# This turns out to be unnecessary, due to the results in case 2
# Nevertheless, the tree in case 1 is simpler than in case 2
# Note that the tree in case one is splitting on only ONE category,
#     namely V390, which is fBodyAccJerk-bandsEnergy()-1,16,
#     which I don't completely understand, but it has something
#     to do with the 1) the accelerometer, 2) the jerk, and
#     3) range of energy band???


##############################
# Case 2: All 6 States: Tree #
##############################
HARtree = rpart(as.factor(y_train$V1)~., data = X_train)
print(HARtree)
fancyRpartPlot(HARtree)


#Training accuracy
```

```
train.tree.HAR.pred = predict(HARtree, newdata = X_train, type = "cl
train.tree.HAR.pred
confmatrix(y_train$V1, train.tree.HAR.pred)
#Training accuracy: 89.5%


#Test
test.tree.HAR.pred = predict(HARtree, newdata = X_test)
test.tree.HAR.pred = apply(test.tree.HAR.pred, 1, which.max)
confmatrix(y_test$V1, test.tree.HAR.pred)
# Test Accuracy: 82.9%

# Details of HARtree:
# V53 -> 6
# V266 & V560 -> 4 & 5
# V509 -> 3
# V75 & V441 -> 1 & 2

# Double check to make sure HARtree categorizes active
# and passive states with 100% accuracy
bin.train.tree.HAR.pred = predict(HARtree, newdata = X_train)
bin.train.tree.HAR.pred = bin.train.tree.HAR.pred>3
confmatrix(bin.y_train, bin.train.tree.HAR.pred)
#Training accuracy: 100%


########################
####  naiveBayes  #####
########################
library(e1071)

#Build model
HARbayes = naiveBayes(y_train$V1~., data = X_train)

#Training data
train.bayes.HAR.pred = predict(HARbayes, newdata = X_train, type = "
train.bayes.HAR.pred = apply(train.bayes.HAR.pred, 1, rand.which.max
confmatrix(y_train$V1, train.bayes.HAR.pred)
```

```
#Training accuracy: 74.5%

OAR.pred=HAR.levels[apply(vote.matrix,
                          1,
                          rand.which.max)]

#Test data
test.bayes.HAR.pred = predict(HARbayes, newdata = X_test, type = "cl
test.bayes.HAR.pred = apply(test.bayes.HAR.pred, 1, rand.which.max)
confmatrix(y_test$V1, test.bayes.HAR.pred)
#Test accuracy: 77.0%


####################
#### HAR SVM  #####
####################
library(e1071)

# Begin by importing X_test, X_train, y_test, and y_train
# from the UCI repository.
# http://archive.ics.uci.edu/ml/machine-learning-databases/00240/
# UCI HAR Dataset.zips

################################
# Case 1: basic SVM in 2-case #
################################
#Fit SVM model to training data
bin.HARsvm = svm(bin.y_train~., data = X_train, kernel = "linear", c

bin.train.SVM.HAR.pred = predict(bin.HARsvm, newdata = X_train)
confmatrix(bin.y_train, train.SVM.HAR.pred)

bin.test.SVM.HAR.pred = predict(bin.HARsvm, newdata = X_test)
confmatrix(bin.y_test, test.SVM.HAR.pred)


# 100% Accuracy. Linearly Separable. Yay!
plot(X_train$V390, col = bin.y_train)
```

```
plot(X_train$V390, col = y_train$V1)
# Because the data are in so many dimensions, it's difficult to see
# but column V390 gives a pretty good idea that the two classes are


#################################################################
# Case 2: basic SVM in 6-case using one against one approach #
#################################################################

plot(X_train$V390, col = y_train$V1)
# A quick look at the same plot from above indicates taht there is m
# though, again, we must remember that this is just one attribute ou

# Note also that the plot indicates that the "silly walks" concept i
# different from one another.


#################################################################
# Case 3: Tune SVM in 6-case using one against one approach #
#################################################################
SVMmodel = svm(as.factor(y_train$V1)~.,
               data = X_train,
               gamma = 1e-04,
               cost = 10000)
SVMpred = predict(SVMmodel, newdata = X_test)

confmatrix(y_test$V1, SVMpred)


#Try it by hand first. Autotune
SVMparams = expand.grid(gamma = 10^(-4:1), cost = 10^(3:8))
SVMmodels = vector(length = nrow(SVMparams), mode = "list")
SVMconfmatrices = vector(length = nrow(SVMparams), mode = "list")
SVMparams

for(i in 1:nrow(SVMparams)){
  #make models
  SVM.model = svm(y_train$V1~.,
```

```
                    data = X_train,
                    gamma = SVMparams$gamma[i],
                    cost = SVMparams$cost[i])
  SVMmodels[[i]]=SVM.model

  #store confusion matrices
  temp.pred = predict(SVMmodels[[i]], newdata = X_test)
  SVMconfmatrices[[i]] = confmatrix(y_test$V1, temp.pred)

  #fraction completed
  print(i/nrow(SVMparams))
}

SVMmodels
SVMconfmatrices

# Round 1 params: gamma = 10^(-5:-1), cost = 10^(0:3)
# Best results from round 1: gamma = 1e-04; cost = 1000; accuracy =

# Round 2 params: gamma = 10^(-6:0), cost = 10^(1:4)
# Best results from round 2: gamma = 1e-04; cost = 10000
# accuracy: 96.3%

# Round 3 params: gamma = 10^(-4,1), cost = 10^(3,8)
```

```
# HARoversc.R
# Updated 6 December 2015
# Parker Rider

#################### K-Nearest Neighbors ####################

#Standardize X_train

xtrnbar<-apply(X_train,2,mean)
xtrnbarmat=cbind(rep(1,nrow(X_train)))%*%xtrnbar
sxtrn=apply(X_train,2,sd)
sxtrnmat=cbind(rep(1,nrow(X_train)))%*%sxtrn
stdxtrn=(X_train-xtrnbarmat)/sxtrnmat
#apply(stdxtrn,2,mean)
#apply(stdxtrn,2,sd)

#Standardize X_test

xtstbar<-apply(X_test,2,mean)
xtstbarmat=cbind(rep(1,nrow(X_test)))%*%xtstbar
sxtst=apply(X_test,2,sd)
sxtstmat=cbind(rep(1,nrow(X_test)))%*%sxtst
stdxtst=(X_test-xtstbarmat)/sxtstmat
#apply(stdxtst,2,mean)
#apply(stdxtst,2,sd)

###### Misc Assignments ######

names(y_train)<-c("class")
names(y_test)<-c("class")
y_train$class<-as.factor(y_train$class)
y_test$class<-as.factor(y_test$class)

####### Predictions #######

library(class)
```

```
knnmodel1<-knn(train=stdxtrn,
                                  test=stdxtst,
                cl=y_train$class,
                k=1)
confmat1<-confmatrix(y_test$class,knnmodel1)
confmat1

accvect = rep(0,20)
for(k in 1:20){
  predcvk = knn.cv(train = rbind(stdxtrn,stdxtst),
                                        cl=rbind(y_train,y_test)$cla
  accvect[k] = confmatrix(
                                  rbind(y_train,y_test)$class, predcvk
}
k_0 = which.max(accvect)

knnmodel16<-knn(train=stdxtrn,
                                  test=stdxtst,
                cl=y_train$class,k=k_0)
confmat16<-confmatrix(y_test$class,knnmodel16)
confmat16

################ Weighted KNN ##################

fithar100<-train.kknn(ytot~.,data=xtot, kmax=100, kernel=
                        c('rectangular','triangular',
                          'epanechnikov','biweight',
                          'triweight','cos','inv',
                          'gaussian','optimal'),distance=2)
ktype100<-(fithar100$best.parameters)$kernel
k100<-(fithar100$best.parameters)$k

kknnmodel10<-(kknn(y_train$class~.,
                                  train=stdxtrn,
                test=stdxtst,
                                  k=k100,
                kernel=ktype100,
```

```
                distance=2))$fitted.values
confmkknn10<-confmatrix(y_test$class,predkknn10)
confmkknn10

############### Neural Networks ##################

library(pROC)
library(nnet)
library(reshape)
library(scales)

model1<-nnet(y_train$class~.,data=X_train,size=1)
prednn1<-predict(model1,newdata=X_test,type="class")
confmnn1<-confmatrix(y_test$class,prednn1)
confmnn1
plot(model1)

modelnn9<-nnet(y_train$class~.,
                                data=X_train,
                size=9,
                MaxNWts=6000)
prednn9<-predict(modelnn9,
                                newdata=X_test,
                type="class")
confmnn9<-confmatrix(y_test$class,prednn9)
confmnn9
plot(modelnn9)
nnhat9<-predict(modelnn9,newdata=X_test,type='class')
plot(roc(response=y_test$class,predictor=as.numeric(nnhat9)))

############### Random Forests ##########################

forestmodel<-randomForest(as.factor(y_train$class)~.,
                                                    data=X_train
forestpred<-predict(forestmodel,newdata=X_test)
forestacc<-confmatrix(as.factor(y_test$class),
                                                 forestpred)$accuracy
```

```
plot(forestmodel)
foresthat<-predict(forestmodel,newdata=X_test,type='class')
plot(roc(response=as.factor(y_test$class),
                predictor=as.numeric(foresthat)))
```

```
# sillywalks.R
# Updated 5 December 2015
# David Ebert

# Given walking data for 2 people, can we determine who the person i

########################
#### Set up data ######
########################

# Import subject factor data
subject_train <- read.table("~/Desktop/HAR/Data/subject_train.txt",
                            quote="\"", comment.char="")
subject_test <- read.table("~/Desktop/HAR/Data/subject_test.txt",
                           quote="\"", comment.char="")

# Combine data into one large data frame
walkerY = rbind(subject_train, subject_test)
walkerX = rbind(X_train, X_test)

# Reduce training data so it is only 1, 2, or 3: Walking, walking up
# or walking down.
walkerX = walkerX[(as.numeric(y_train$V1) < 4),]
walkerY = walkerY[(as.numeric(y_train$V1) <4),]


#Select walker records for walkers 1,2, 3, and 7
walkerX.example = walkerX[walkerY %in% c(1:3, 7),]
walkerY.example = walkerY[walkerY %in% c(1:3, 7)]


#########################################################################
### Use a tree to find training and test accuracy for n<= 30 walkers
#########################################################################

library(rpart)
```

```
library(rattle)

n = 10
set.seed(1)
walkerID= sample(30,n)

walkerX.sample = walkerX[walkerY %in% walkerID,]
walkerY.sample = as.factor(walkerY[walkerY %in% walkerID])

# 70% training and 30% test data
train = sample(length(walkerY.sample), length(walkerY.sample)*.7)

# rpart tree
library(rpart)
tree = rpart(walkerY.sample[train] ~., data = walkerX.sample[train,]
train.treepred = predict(tree, newdata = walkerX.sample[train,],
                         type = "class")
train.treeresult = confmatrix(walkerY.sample[train], train.treepred)

test.treepred = predict(tree, newdata = walkerX.sample[-train,],
                        type = "class")
test.treeresult = confmatrix(walkerY.sample[-train], test.treepred)


# Accuracy with 3 walkers: ~97%
# Accuracy with 10 walkers: ~84%
# Accuracy with 15 walkers: ~70%
# Accuracy with 20 walkers: ~65%
# Accuracy with 25 walkers: ~63%
# Accuracy with all 30 walkers: 49.29%    <- THAT IS AWESOME!

#Pretty print the above table:
a = matrix(1:20, nrow = 5, ncol = 4)
library(gridExtra)
grid.table(a)
grid.table(test.treeresult$matrix)

write.table(test.treeresult$matrix, file = "tenWalkers.csv", sep = "
```

```
#####################################################################
# Find how tree accuracy changes with increased number of walkers #
#####################################################################
maxWalkers = 29
nIterations = 8

bigAccVector = 1:maxWalkers

set.seed(1)
for(i in 2:maxWalkers){

  smallAccVector = 1:nIterations

  for(j in 1:nIterations){

    walkerID= sample(30,i)

    walkerX.sample = walkerX[walkerY %in% walkerID,]
    walkerY.sample = as.factor(walkerY[walkerY %in% walkerID])

    # 70% training and 30% test data
    train = sample(length(walkerY.sample), length(walkerY.sample)*.7

    # rpart tree
    library(rpart)
    tree = rpart(walkerY.sample[train] ~., data = walkerX.sample[tra
    test.treepred = predict(tree, newdata = walkerX.sample[-train,],
                            type = "class")
    test.treeresult = confmatrix(walkerY.sample[-train], test.treepr

    smallAccVector[j] = test.treeresult$accuracy
  }
  print(i)
  bigAccVector[i] = mean(smallAccVector)
}

bigAccVector
```

```
finalAccVectorTREE = c(bigAccVector, 0.4992)
# Note that the case of 30 walkers is the same no matter which
# 30 are chosen, though there is still randomness in the training/te
# data split

plot(finalAccVectorTREE,
     xlab = "Number of Walkers Compared",
     ylab = "Average Accuracy of Model Over 8 Iterations",
     main = "SVM: Number of Walkers vs Model Accuracy")


#################################################################
# Use SVM to find training and test accuracy for n<= 30 walkers #
#################################################################
library(e1071)

n = 30
set.seed(1)
walkerID= sample(30,n)

walkerX.sample = walkerX[walkerY %in% walkerID,]
walkerY.sample = as.factor(walkerY[walkerY %in% walkerID])

# 70% training and 30% test data
train = sample(length(walkerY.sample), length(walkerY.sample)*.7)

# svm model
svmModel = svm(walkerY.sample[train] ~., data = walkerX.sample[train
train.svmpred = predict(svmModel, newdata = walkerX.sample[train,],
                        type = "class")
train.svmresult = confmatrix(walkerY.sample[train], train.svmpred)
train.svmresult

test.svmpred = predict(svmModel, newdata = walkerX.sample[-train,],
                       type = "class")
test.svmresult = confmatrix(walkerY.sample[-train], test.svmpred)
test.svmresult
```

```
#Pretty print the above table:
a = matrix(1:20, nrow = 5, ncol = 4)
library(gridExtra)
grid.table(a)
grid.table(test.treeresult$matrix)

write.table(test.treeresult$matrix, file = "tenWalkers.csv", sep = "

#####################################################################
# Find how tree accuracy changes with increased number of walkers #
#####################################################################
maxWalkers = 29
nIterations = 8

bigAccVector = 1:maxWalkers

set.seed(1)
for(i in 2:maxWalkers){

  smallAccVector = 1:nIterations

  for(j in 1:nIterations){

    walkerID= sample(30,i)

    walkerX.sample = walkerX[walkerY %in% walkerID,]
    walkerY.sample = as.factor(walkerY[walkerY %in% walkerID])

    # 70% training and 30% test data
    train = sample(length(walkerY.sample), length(walkerY.sample)*.7

    # rpart tree
    svmModel = svm(walkerY.sample[train] ~., data = walkerX.sample[t
    test.svmpred = predict(svmModel, newdata = walkerX.sample[-train
    test.svmresult = confmatrix(walkerY.sample[-train], test.svmpred

    smallAccVector[j] = test.svmresult$accuracy
```

```
  }
  print(i)
  bigAccVector[i] = mean(smallAccVector)
}

bigAccVector

finalAccVectorSVM = c(bigAccVector, 0.9496)
# SVM yields an incredible accuracy of 94% with all 30 walkers!

plot(finalAccVectorTREE,
     xlab = "Number of Walkers Compared",
     ylab = "Average Accuracy of Model Over 8 Iterations",
     main = "Number of Walkers vs Model Accuracy",
     type = "l",
     col = "red")
lines(finalAccVectorSVM,
      type = "l",
      col = "black")
text(locator(), labels = c("Trees", "SVM"))
```

```
# reduction.R
# Updated 5 December 2015
# David Ebert

################################################
# Dimensionality reduction applied to HAR data #
################################################

###############
# Import data #
###############
X_test <- read.table("C:/Users/000678922/Desktop/HAR/X_test.txt",
                     quote="\"")
X_train <- read.table("C:/Users/000678922/Desktop/HAR/X_train.txt",
                      quote="\"")
y_test <- read.table("C:/Users/000678922/Desktop/HAR/y_test.txt",
                     quote="\"")
y_train <- read.table("C:/Users/000678922/Desktop/HAR/y_train.txt",
                      quote="\"")

y_test = as.factor(y_test$V1)
y_train = as.factor(y_train$V1)


################################################
# Reduce dimensions using PCA using preProcess #
# function from caret package                  #
################################################
library(caret)
trans = preProcess(X_train, method = c("BoxCox",
                                       "center",
                                       "scale",
                                       "pca"))
head(trans)
pcaTrain = predict(trans, X_train)
pcaTest = predict(trans, X_test)
```

```
# Data reduced to "only" 102 columns!
# Preserves 95% of variance

plot(pcaTrain$PC1,
     pcaTrain$PC2,
     col = y_train,
     pch = c(1:6)[as.numeric(y_train)])
# Plot of first 2 components of training data
# shows some clustring possibilities!

################################
# Tree applied to reduced data #
################################
library(rpart)
pcaTree = rpart(as.factor(y_train$V1)~., data = pcaTrain)
treepred = predict(pcaTree, newdata = pcaTest, type = "class")
confusionMatrix(data = treepred, reference = y_test)

# New accuracy is a fairly bad  73.1%


##############################
#SVM applied to reduced data #
##############################
library(e1071)
pcaSVM = svm(y_train~., data = pcaTrain)
svmpred = predict(pcaSVM,
                  newdata = pcaTest,
                  type = "class")
confusionMatrix(data = svmpred, reference = y_test)
# New accuracy is a still-good 93.7%

pcaSVMtuned = svm(y_train~.,
                  data = pcaTrain,
                  cost = 10000,
                  gamma = 1e-04)
svmtunedpred = predict(pcaSVMtuned,
                       newdata = pcaTest,
```

```
                        type = "class")
confusionMatrix(data = svmtunedpred, reference = y_test)
# Tuned SVM accuracy is a now-WORSE 91.7%


##############################
#SVM applied to reduced data #
##############################

traindata = cbind(y_train, pcaTrain)

tunespecs <- tune(svm, y_train~., data = traindata,
            ranges = list(gamma = 2^(-1:3), cost = 2^(1:6)))

pcaSVMtuned = svm(y_train~.,
                  data = pcaTrain,
                  cost = 4,
                  gamma = 0.5)
svmtunedpred = predict(pcaSVMtuned,
                       newdata = pcaTest,
                       type = "class")
confusionMatrix(data = svmtunedpred, reference = y_test)
# Tuned SVM accuracy is a now-WORSE 91.7%
```