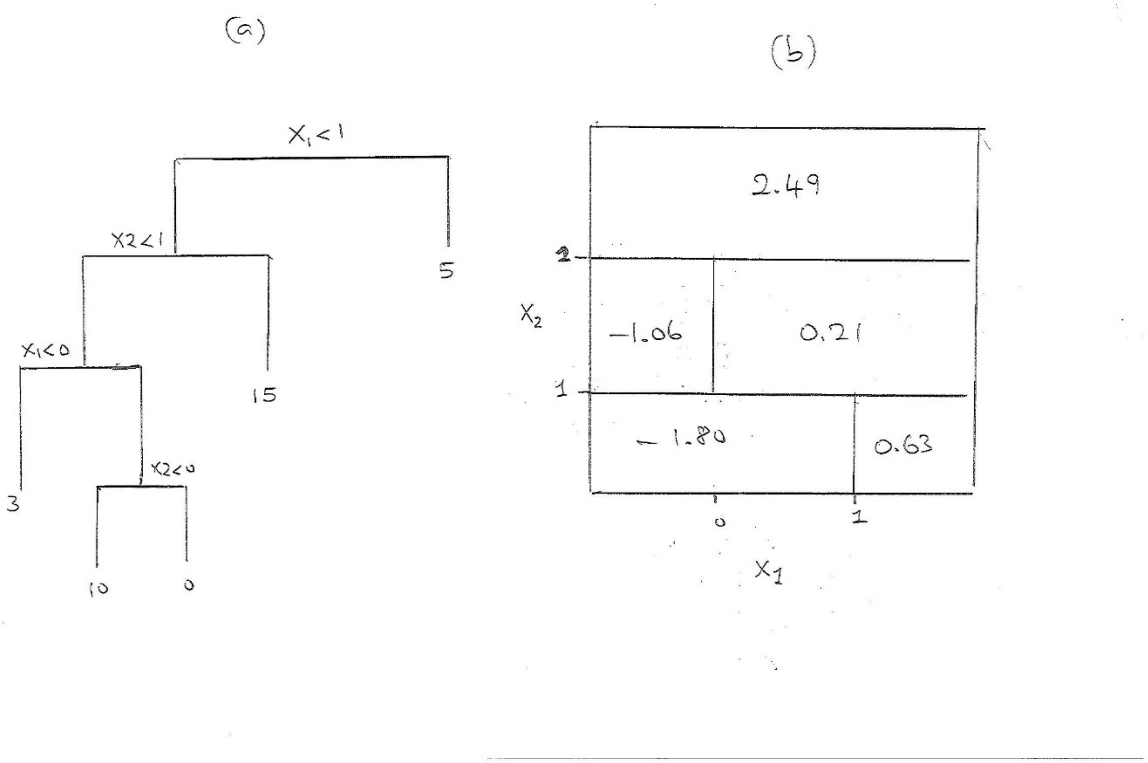
Department of Statistics

STATS 784: Data Mining

Assignment 2 2016 Model Answer

Question 1



Question 2

1. *Read the training and test data into R*

```
infile = "https://www.stat.auckland.ac.nz/~lee/784/Assignments/training.csv"
training.df = read.csv(infile, header=TRUE, stringsAsFactors = FALSE)


infile2 = "https://www.stat.auckland.ac.nz/~lee/784/Assignments/test.csv"
test.df = read.csv(infile2, header=TRUE, stringsAsFactors = FALSE)
```

2. *Fit a tree to the x and y coordinates separately and calculate the predicted values for the test set. Print out the first 10 values of each predictor.*

First the x-coordinate:

```
library(rpart)
fit.xtree = rpart(left_eye_center_x~., data=training.df[,-2], cp=0.001)
plotcp(fit.xtree)
x.cp=printcp(fit.xtree)
x.cp[order(x.cp[,4])[1],1] # extract cp for the smallest CV error
[1] 0.02106293
```

Unfortunately the minimum CV error is quite variable, but repeating this code several times indicates that cp=0.02 is not an unreasonable choice.

Let's go with this.

```
> x.tree.pruned = prune(fit.xtree, cp=0.02)

> # training error
> mean(residuals(x.tree.pruned)^2)
[1] 2.19279

> # test error
> x.predict = predict(x.tree.pruned, newdata=test.df[,-2])
> mean((x.predict-test.df[,1])^2)
[1] 3.946122
```

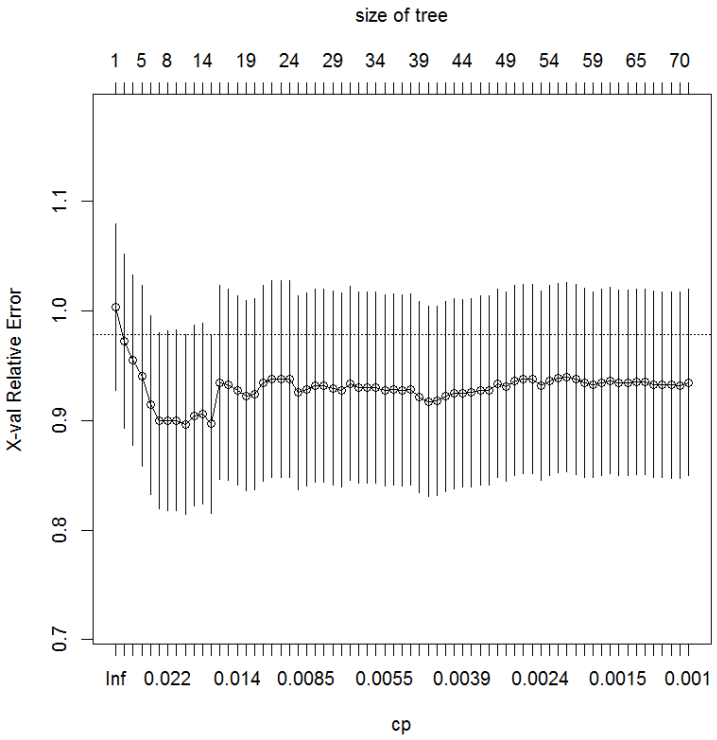


Figure 1. CP plot for the tree with cp=0.001

Printing out the first 10 values:

```
> head(predict(x.tree.pruned),10)
 [1] 65.97659 68.10075 64.63368 65.97659 65.97659 64.63368 66.42259 66.59466
 [9] 65.97659 65.97659
```

Doing the same thing for the y-cordinate, we selected cp of 0.01 as a reasonable choice.

[10 marks]

3. *Estimate the prediction error using the test set estimate, and CV and the bootstrap on the training data. Compare.*

First the test and training errors:

```
> # training error
> mean(residuals(x.tree.pruned)^2)
[1] 2.19279

> # test error
> x.predict = predict(x.tree.pruned, newdata=test.df[,-2])
> mean((x.predict-test.df[,1])^2)
[1] 3.946122

> y.predict = predict(fit.ytree, newdata=test.df[,-1])
> # training error

> mean(residuals(fit.ytree)^2)
[1] 0.887749
> # test error
> mean((y.predict-test.df[,2])^2)
[1] 3.423971
```

Now do the cross-validation and the bootstrap

```
library(bootstrap)

theta.fit <- function(x,y){
data=data.frame(y=y,x)
rpart(y~.,data=data, cp=0.02)
}
theta.predict <- function(fit,x){
  predict(fit, newdata=data.frame(x))
}
 sq.err <- function(y,yhat) { (y-yhat)^2}
```

```
# x-coordinate
pixels = training.df[,-c(1,2)]
x = training.df[,1]
> CV5 = numeric(10)
> for(i in 1:10){
+ results.cv.x = crossval(pixels,x,theta.fit,theta.predict, ngroup=5)
+ CV5[i] = mean((results.cv.x$cv.fit - x)^2)
+ }
> mean(CV5)
[1] 3.53789
# y-coordinate

> y = training.df[,2]
> CV5.y = numeric(10)
> for(i in 1:10){
+ results.cv.y = crossval(pixels,y,theta.fit,theta.predict, ngroup=5)
+ CV5.y[i] = mean((results.cv.y$cv.fit - y)^2)
+ }
> mean(CV5.y)
[1] 3.847702

# bootstrap results
results.boot.x = bootpred(pixels, x,  theta.fit, theta.predict,
err.meas=sq.err, nboot=50)
> results.boot.x
[[1]]
[1] 2.19279

[[2]]
[1] 1.042388

[[3]]
[1] 3.304834

$call
bootpred(x = pixels, y = x, nboot = 50, theta.fit = theta.fit,
    theta.predict = theta.predict, err.meas = sq.err)

>  2.19279+ 1.042388
[1] 3.235178
> 3.304834
[1] 3.304834

results.boot.y = bootpred(pixels, y,  theta.fit, theta.predict,
err.meas=sq.err, nboot=50)
> results.boot.y
[[1]]
[1] 1.96758

[[2]]
[1] 1.033014

[[3]]
[1] 3.323163

$call
bootpred(x = pixels, y = y, nboot = 50, theta.fit = theta.fit,
```

```
    theta.predict = theta.predict, err.meas = sq.err)

> 1.96758+1.033014
[1] 3.000594
> 3.323163
[1] 3.323163
```

Summarizing these in a table, we get

	x-cordinate	y-coordinate
Training	2.19	0.89
Test	3.95	3.42
CV5	3.54	3.85
Err+opt	3.30	3.00
632	3.23	3.32

Points to note:

- The model for the y-coordinate is overfitting, the test error is much more than the training error.
- The CV estimates are quite variable (running the code again produces a different result, even averaging 10 times.)
- Bootstrap generally less than CV.

[10 marks]

4. Print out the first 25 images in the test data (on one page) and mark the position of the predicted eye centre.

```
# define a function to drew a single face

plotImage = function(imagevec){
imagemat = matrix(imagevec,96,96)
for(i in 1:96)imagemat[i,] = rev(imagemat[i,])
image(1:96, 1:96, imagemat, col = gray((0:255)/255), axes=FALSE)
}

# draw 25 images from the test data

par(mfrow=c(5,5), mar = c(0,0,0,0))
for(i in 1:25){
 plotImage(unlist(test.df[i,-c(1,2)]))
 points(x.predict[i],96-y.predict[i], pch=19, cex=1.5, col="red")
}
par(mfrow = c(1,1), mar = c(5, 4, 4, 2) + 0.1)
```

[10 marks]

5. *Fit random forests and see if there is any improvement in the PE over fitting a single tree. You may want to use* ***caret*** *to tune the random forests.*

For this part we will work with a reduced data set for the two models, as caret and random Forest run too slowly using the full set of 9216 features. The tree we fitted to the x-coordinate with cp=0.001 used only a small fraction of the available variables. This is not surprising, as pixels in the lower part of the image are unlikely to contribute much to predicting the eye position.

We can extract the variables used in the model, and create a formula including just those variables using the following code:

```
> head(fit.xtree$variable.importance)
   V3621    V3717    V3622    V3423    V3520    V3519
461.2292 437.0245 306.8044 305.0914 251.6629 233.4213
> length(fit.xtree$variable.importance)
[1] 361
```

Thus, there are 361 variables used out of the 9216 available. To create the formula:

```
formula.x = as.formula(paste("left_eye_center_x~",
    paste(names(fit.xtree$variable.importance), collapse="+"), sep=""))
```

Then we can explore different cp values using caret

```
> tree.CV.x <- train(formula.x, data=training.df[,-2],
+     method = "rpart",
+     tuneGrid = my.grid,
+     trControl = trainControl(method="cv", number=5, repeats=10))
> tree.CV.x
CART

 953 samples
9216 predictors

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 761, 763, 762, 762, 764
Resampling results across tuning parameters:

  cp     RMSE      Rsquared   RMSE SD     Rsquared SD
  0.010  1.810037  0.2295849  0.11344762  0.05599240
  0.015  1.837394  0.1995643  0.09130515  0.04161033
  0.020  1.844103  0.1870761  0.10467826  0.05067258
  0.025  1.883172  0.1545290  0.10968738  0.05208872
  0.030  1.907965  0.1183185  0.11047588  0.06664810
```

In the light of this, our choice of cp=0.02 doesn't seem too bad, given the variability in the CV estimates. Compare 1.844103 ^2= 3.40 for the reduced model to 3.88 for the full model, so the reduced model is doing a bit better.

Using caret in this way, we can see the results for the reduced models, using cp=0.02 and 0.01 for x and y.

	x-cordinate	y-coordinate
Training	2.19	1.62
Test	3.91	3.35
CV5	3.21	3.25
Err+opt	3.47	3.80
632	3.12	3.14

The results are broadly similar, suggesting that not much is lost by using the reduced variables.

Now we fit a random forest. The software default is 1/3 of the number of features, in this case 361/3 about 120, so we used values of mtry as 100,110, 120,130,140.

```
> my.grid <- expand.grid(.mtry=c(100, 110,  120, 130, 140))
> rf.CV <- train(formula.x, data=training.df,
+     method = "rf",
+     ntree=200,
+     tuneGrid = my.grid,
+     trControl = trainControl(method="cv", number=5, repeats=10))
> rf.CV
Random Forest

 953 samples
9217 predictors

No pre-processing
Resampling: Cross-Validated (5 fold)

Summary of sample sizes: 763, 763, 763, 762, 761

Resampling results across tuning parameters:

  mtry  RMSE      Rsquared   RMSE SD    Rsquared SD
  100   1.440508  0.5035774  0.1724213  0.1142423
  110   1.446948  0.4940041  0.1782463  0.1196643
  120   1.453582  0.4853469  0.1657095  0.1091293
  130   1.449397  0.4871280  0.1620692  0.1052647
  140   1.452793  0.4824016  0.1763763  0.1161318

RMSE was used to select the optimal model using  the smallest value.
The final value used for the model was mtry = 100.
```

Seems like the default setting is OK.

We could use caret fit the bootstrap as well, but an alternative is to fit the model directly and use the fact that the model fit also calculates an OOB estimate of PE:

```
fit.rf120 = randomForest(formula.x, data = training.df, ntree=200,mtry=120)
fit.rf120$mse[200]
[1] 2.027798
```

Compare this with $1.453582^2 = 2.11$. If we use caret for the bootstrap, we get

```
> rf.boot <- train(formula.x, data=training.df,
+     method = "rf",
+     ntree=200,
+     tuneGrid = my.grid,
+     trControl = trainControl(method="boot", number=50))
```

```
> rf.boot
Random Forest

 953 samples
9217 predictors

No pre-processing
Resampling: Bootstrapped (50 reps)
Summary of sample sizes: 953, 953, 953, 953, 953, 953, ...
Resampling results across tuning parameters:

  mtry  RMSE      Rsquared
  100   1.474113  0.4591610
  110   1.470690  0.4606809
  120   1.474715  0.4565168
  130   1.474185  0.4558633
  140   1.471538  0.4580095

RMSE was used to select the optimal model using  the smallest value.
The final value used for the model was mtry = 110.
```

This value of 1.474715^2 = 2.174784 comperes well with the value of OOB estimate (2.02).

The RF errors are in the table below.

	x-cordinate	y-coordinate
Training	2.06	1.78
Test	2.34	1.47
CV5	2.10	1.79
Err+opt	2.17	1.93
OOB	2.03	1.80

These errors are consistently lower than those in the tree table above, so the RF is doing a better job at predicting.