Predicting The Age of Abalone Through Regression Analysis

February 19, 2008

Contents

1 Introduction

A linear regression analysis was performed on a data set of characteristics related to the number of abalone rings using the R analysis package. These characteristics were: Sex (Male, Female, or Infant), Length, Diameter, Height, Whole weight, Shucked weight, Viscera weight, and Shell weight.

The independent variables (characteristics) were first standardized before a matrix of scatter plots were generated to get a high level understanding of the linear relationships between pairs of attributes. From here, we began constructing our linear model by randomly partitioning the data in training and test sets, calculated weights for the linear model for various ridge regression parameters (λ) and training partitions, identified the λ that minimized root mean square error (RMSE) and then used the model to make some predictions.

The parameters for the optimized linear model was as follows:

```
                   weight      mean   std dev
bias             9.98485036
M                0.46746332 0.4154110 0.5316056
F                0.42869850 0.3637231 0.5140590
Length          -0.01305779 0.5739761 0.1700483
Diameter         1.09625791 0.4578596 0.1491762
Height           0.60452821 0.1895171 0.0916944
Whole weight     4.30201619 0.8784376 0.5396926
Shucked weight  -4.26114081 0.4092768 0.2716407
Viscera weight  -1.09340499 0.2305107 0.1594469
Shell weight     1.28036417 0.2887933 0.1891060
```

The optimized ridge regression parameter λ was found to be 0.8. The `mean` and the `std dev` columns above were used to in the standardization which will explained in detail later.

2 Background Information

2.1 What are Abalone?

Abalone are marine snails. Abalone shells have a low and open spiral structure, and are characterized by several respiratory holes in a row near the shell's outer edge. The innermost layer of the shell is composed of nacre or mother-of-pearl, which in many species is highly iridescent, giving rise to a range of strong and changeable colors which make them attractive to humans as a decorative object. [1]

Abalone has long been a delicacy around the world. You'll find several species of these large sea-snails living in the wild along the Pacific coast of North America, as well as Japan, China, Australia and New Zealand. [2]

Although wild abalone populations remain in a state of recovery, abalone farming sustains continuing public consumption of this ocean delicacy. Farm-raised abalone are harvested when their shells are no more than four inches long, so any abalone bigger than that was probably "poached," or sold illegally by a sport diver. [2]

2.2 Why Model Rings?

Rings in abalone correspond to their age, like rings do in trees. The age of abalone would be of interest to marine biologists as well as abalone farmers for a variety of reasons. The typical method for determining the age of abalone involves cutting the shell through the cone, staining it, and counting the number of rings through a microscope which is a destructive and time-consuming task. [3]

In order to make the task of predicting age faster and easier, other measurements, which are easier to obtain, were investigated to predict the age. [3]

3 Data Preparation

3.1 Reading the Data

Reading the data into R was accomplished by running the following line of code at the R command prompt:

```
abalone <- read.table("abalone.data", sep=',')
```

When running this command, one must make sure that the working directory is set to the same directory where the data file resided.

3.2 Converting Categorical Data

The first column of the original data [3] was labeled **Sex** and was populated with the categorical values M, F, and I. In order to include the **Sex** parameter in the model, these categorical values had to be converted to numerical values. This was done by replacing the original column with two new columns. The first new column was labeled **M** for male and the second was labeled **F** for female. Males were designated by putting a 1 in the **M** column and a 0 in the **F** column. Females were designated by putting a 0 in the **M** column and a 1 in the **F** column. Infants were designated by a 0 in both the **M** and **F** columns.

In order to accomplish this conversion, the following R code was executed after the data had been read into the **abalone** table:

```
# converts the first column of the abalone data (M,F,or I) into
# two columns: M and F where M has (M=1,F=0), F has (M=0,F=1), and
# I has (M=0,F=0)
consex <- function(orig){
   # create M and F column vectors
   Mcol <- orig[,1,drop=FALSE]
   Fcol <- Mcol
   # convert the M's to 1 & F's to 0 in the first col
   Mcol <- mtoone(Mcol)
   # convert the F's to 1 & M's to 0 in the second col
   Fcol <- ftoone(Fcol)
   # create the new nx2 vector that are proper format of the
   # converted "Sex" column
   MandF <- cbind(Mcol, Fcol)
   colnames(MandF) <- c("M","F")
   # now build the new dataframe with proper conversion
   converted <- cbind(MandF, orig[,2:9])
```

```
   converted
}
```

The function **consex** calls two helper functions: **mtoone** and **ftoone**, which are listed together below.

```
# take in a vector of M's and F's and return a vector that has
# replaced M's by 1, F's by 0, and I's by 0
mtoone <- function(mvec) {
   newvec <- c()
   for(rowi in 1:nrow(mvec)){
    if (mvec[rowi,1] == "M")
       newvec <- c(newvec,1)
    if (mvec[rowi,1] == "F")
       newvec <- c(newvec,0)
    if (mvec[rowi,1] == "I")
       newvec <- c(newvec,0)
   }
   newvec
}

# take in a vector of M's and F's and return a vector that has
# replaced M's by 0, F's by 1, and I's by 0
ftoone <- function(mvec) {
   newvec <- c()
   for(rowi in 1:nrow(mvec)){
    if (mvec[rowi,1] == "M")
       newvec <- c(newvec,0)
    if (mvec[rowi,1] == "F")
       newvec <- c(newvec,1)
    if (mvec[rowi,1] == "I")
       newvec <- c(newvec,0)
   }
   newvec
}
```

3.3 Data Partitioning

In order to simulate real world conditions, we constructed our model using randomly selected portions of the data. The portion used to create the model we call the *training set* while the remaining portion which we use to make predictions we call the *testing set*.

During the analysis, many iterations based on different randomly selected training and test sets had to be constructed. The following two R functions were frequently called to accomplish this task:

```
# Returns a randomly selected subset of rows from the original input dataframe.
# orig - original (complete) dataset (dataframe), assumed that samples
#        are stored in rows of a 2-D table
```

```
# fraction - the fraction of the original dataset used for training
#            (e.g. 0.8 is 80%)
trainSetRows <- function(orig, fraction) {
  randorder <- sample(nrow(orig))  # create set of randomly selected rows
  nTrain <- round(nrow(orig)*fraction) # calc # of rows in training set
  trainRows <- randorder[1:nTrain]  # 1st nTrain rows in rand order
  trainRows
}

# returns a vector of the row numbers of the test set
# allRows is a vector containing all the row numbers in the data table
# trainRows is the vector of training rows returned from the trainSetRows function
testSetRows <- function(allRows, trainRows) {
  testRows <- setdiff(allRows, trainRows)
  testRows
}
```

This code was usually called from within other functions that managed various iterative analysis. The `randorder` function randomizes the row numbers in a table and returns a vector of these random row numbers. This vector was then used to extract the training and testing set rows used in various analysis.

4 Regressions Analysis

4.1 The Linear Model

4.1.1 Definition of the Linear Model

The model used to describe the data is as follows:

$$R = w_0 + \sum_{i=1}^{9} w_i x_i \tag{1}$$

In equation (1), R is the number of rings the abalone has, the w's are the weights (our fitted parameters) and the x's are the independent variables. The independent variable can be described as: x_1 = M, x_2 = F, x_3 = Length, x_4 = Diameter, x_5 = Height, x_6 = Whole weight, x_7 = Shucked weight, x_8 = Viscera weight, and x_9= Shell weight.

4.1.2 λ Parameterization

The goal was to fit the w vector described in (1) in such a way as to minimize the root mean square error (RMSE) parameterized by the ridge regression factor λ. This error can be described as follows:

$$\frac{1}{2}\sum_{i=1}^{9}\left(t_n - \mathbf{w}^T\mathbf{x}_i\right)^2 + \frac{1}{2}\lambda\mathbf{w}^T\mathbf{w} \tag{2}$$

Taking the gradient of (2), setting it equal to zero, and solving for the vector $\mathbf{w}$, yields in matrix notation:

$$\mathbf{w} = \left(\lambda + \mathbf{X}_T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{T} \tag{3}$$

The matrices $\mathbf{X}$ and $\mathbf{T}$ are the independent and dependent (target) variables respectively.

4.1.3 Calculating Weights

With equation (3), we now have an expression that can be used to solve for the weight vector **w** given parameter λ. The following R function was used to accomplish this task:

```
# This function returns a vector of the linear least square regression
# weights with the Ridge Regression penalty factored in.  Arguments:
# X - the coefficient matrix wrt the independent variables
# Y - the matrix of dependent variables
# lam - square weight penalty factor
# Notes on usage of this function:
# (1) Independent var's should NOT be standardized (this is done internally).
# (2) Samples w/ missing data need to be removed before running this function.
# (3) This function can be called within a loop of incremental lambdas.
#
llsMake <- function(X,Y,lam) {
   # if X or Y come in as a list, convert it to numeric
   X <- as.matrix(apply(X,2,as.numeric))
   # account for the bias
   # standardize the independent variables
   means <- matrix(apply(X,2,mean),ncol(X),1)
   stdevs <- matrix(apply(X,2,sd),ncol(X),1)
   N <- nrow(X)  # number of samples
   D <- ncol(X)  # number of parameters of dimensions
   Xs <- (X - matrix(rep(means,N),N,D,byrow=TRUE))/
          matrix(rep(stdevs,N),N,D,byrow=TRUE)

   Y <- as.matrix(apply(Y,2,as.numeric))
   # account for the bias
   Xs1 <- cbind(1, Xs)
   colnames(Xs1)[1] <- "bias"
   # now solve for the weights, be sure not to penalize w0
   w <- solve((t(Xs1) %*% Xs1)+(lam*diag(c(0,rep(1,ncol(Xs1)-1)))),
              t(Xs1) %*% Y)
   # store the means & std devs along with the weights so we can properly
   # standardize test values when we use our weights for predictions
   meansX <- matrix(apply(cbind(1,X),2,mean),ncol(X)+1,1)
   stdevsX <- matrix(apply(cbind(1,X),2,sd),ncol(X)+1,1)
   w <- cbind(w, meansX)
   w <- cbind(w, stdevsX)
   colnames(w) <- c("weight", "mean", "std dev")
   w
}
```

The *llsMake* function returns a three column matrix. In the first column are the weights as defined by **w**. The second and third columns of this matrix contain the means and standard deviations of the independent variables that were used in training the model. By training, we mean solving for **w**.

The second and third columns of the **w** matrix are used to standardize the independent variables. Standardizing takes the following form:

$$\mathbf{X}_s = \frac{(\mathbf{X} - \mathbf{X}_{mean})}{\mathbf{X}_{stdev}} \tag{4}$$

In equation (4), $\mathbf{X}_s$ is the matrix of standardized inputs (independent variables) and the matrices $\mathbf{X}_{mean}$ and $\mathbf{X}_{stdev}$ are the means and standard deviations of the inputs respectively. Standardization provides two advantages: (i) it allows us the ability to better account for significant differences between the regions of training and testing and (ii), it allows us to more easily compare the relative magnitude of the weights.

4.2 Making Predictions

To make predictions from the weights calculated from $llsMake$, another R function was created:

```
# This function returns a matrix of the predicted values based on the weights
# determined in llsMake.
#
llsUse <- function(weights, X) {
   X <- as.matrix(apply(X,2,as.numeric))
   # standardize X
   N <- nrow(X)  # number of samples
   D <- ncol(X)  # number of parameters of dimensions
   means <- weights[2:nrow(weights),2]
   stdevs <- weights[2:nrow(weights),3]
   Xs <- (X - matrix(rep(means,N),N,D,byrow=TRUE))/
          matrix(rep(stdevs,N),N,D,byrow=TRUE)

   Xs1 <- cbind(1,Xs)  # account for the bias
   colnames(Xs1)[1] <- "bias"
   weights[1,3] <- 1  # replace bias stdev to avoid a division by 0
   model <- Xs1 %*% weights[,1]  # calc the prediction
   model
}
```

The $llsUse$ function outputs the vector *model* which contains the model predictions for the independent variables **X** that were passed in. These predictions could then be used to evaluate RMSE which in turn allowed us to explore how λ and training partition impacted the models accuracy. This will be explored in detail in section 5.3.

5 Experiments

5.1 A Quick Look at the Data

Before diving into regression analysis, it's normally a good idea to take a look at the relationships between the variables - particularly the independent variables. To accomplish this task, the following R code, based on the *pairs* function was used:

```
panel.hist <- function(x) {
   usr <- par("usr"); on.exit(par(usr))
   par(usr = c(usr[1:2], 0, 1.5))
   h <- hist(x, plot = FALSE)
   breaks <- h$breaks; nB <- length(breaks)
   y <- h$counts; y <-y/max(y)
   rect(breaks[-nB], 0, breaks[-1], y, col="cyan")
   dev.copy2eps(file="scatter2.eps")
}
panel.cor <- function(x,y,digits=4, prefix="", cex.cor)
{
   usr <- par("usr"); on.exit(par(usr))
   par(usr = c(0,1,0,1))
   r <- abs(cor(x,y))
   txt <- format(c(r, 0.123456789), digits=digits)[1]
   txt <- paste(prefix, txt, sep="")
   if(missing(cex.cor)) cex <- 0.8/strwidth(txt)
   text(0.5, 0.5, txt, cex=cex*r)
}

diag_hist <- function(y) {
   pairs(y, diag.panel=panel.hist, upper.panel=panel.cor)
}
```

This code produced the chart shown in Figure 1.

In Figure 1., the upper triangular plots were just a transposed version of the charts in the lower triangular region, so it seemed reasonable to put something more meaningful in this region. In this case, the correlation coefficient between x (horizontal axis) and y (vertical axis) variables was selected. Similarly, histograms were chosen for the diagonals.

Dependencies among independent variables is an undesirable quality when doing regression analysis and we can observe some strong dependencies especially between *Length* and *Diameter*. In section 6, we'll delve into this in more detail. There also appears to be some dependencies between the different kinds of weights (Shucked, etc...), but the thickness of the scatter is greater than the thickness between *Length* and *Diameter*.

5.2 Most (MSFs) & Least (LSFs) Significant Factors

For convenience, the table shown in section 1 is presented here again.

	weight	mean	std dev
bias	9.98485036		
M	0.46746332	0.4154110	0.5316056
F	0.42869850	0.3637231	0.5140590
Length	-0.01305779	0.5739761	0.1700483
Diameter	1.09625791	0.4578596	0.1491762
Height	0.60452821	0.1895171	0.0916944

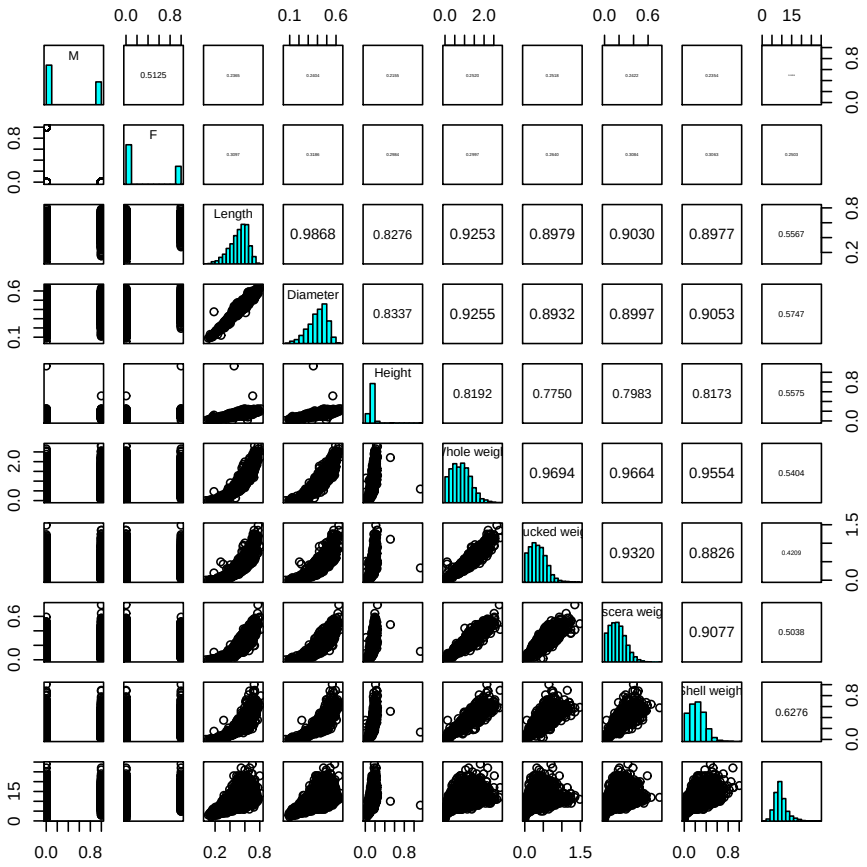

Figure 1: Scatter Plot of Abalone Data.

```
Whole weight    4.30201619 0.8784376 0.5396926
Shucked weight -4.26114081 0.4092768 0.2716407
Viscera weight -1.09340499 0.2305107 0.1594469
Shell weight    1.28036417 0.2887933 0.1891060
```

Now we can reap the benefit of standardizing the inputs. Since the inputs were standardized, the importance of the weights can now easily be seen. The three most significant factors are (in order of importance): bias, Whole weight and Shucked weight.

The three least significant factors are: Length, F, and M. In fact, if we removed these three factors from our model, it does not change noticeably. The following R code was used to remove the aforementioned factors and generate the plots shown in Figure 2.

```
par(mfrow = c(1,2))
tp=0.5
```

```
trainRows <- trainSetRows(abalone, tp)
testRows <- testSetRows(seq(1:nrow(abalone)),trainRows)
Xtrain <- abalone[trainRows, 1:9]
Xtrain <- as.matrix(apply(Xtrain,2,as.numeric))
Xtest <- abalone[testRows, 1:9]
Xtest <- as.matrix(apply(Xtest,2,as.numeric))
Ttrain <- abalone[trainRows, 10]
Ttrain <- as.matrix(Ttrain)
Ttest <- abalone[testRows, 10]
Ttest <- as.matrix(Ttest)
lambda <- 0.8

w <- llsMake(Xtrain, Ttrain, lambda)
Tpred <- llsUse(w, Xtest)
# check the dimensions
dim(Tpred)
dim(Ttest)
plot(Ttest,Tpred,main=c("All the Parameters",
      "(training partition 0.5)"), xlab="Rings (actual)",
      ylab="Rings (predicted)")
abline(coef=c(0,1), col="red")
legend("bottomright",c("Test vs. Prediction", "Test = Prediction"),
        col=c("black","red"), lty=c(-1,1), pch=c(1,-1), merge = TRUE)

Xtrain <- Xtrain[,4:9]
Xtest <- Xtest[,4:9]
w <- llsMake(Xtrain, Ttrain, lambda)
Tpred <- llsUse(w, Xtest)
# check the dimensions
dim(Tpred)
dim(Ttest)
plot(Ttest,Tpred,main=c("LSF Variables Removed",
      "(training partition 0.5)"), xlab="Rings (actual)",
      ylab="Rings (predicted)")
abline(coef=c(0,1), col="red")
legend("bottomright",c("Test vs. Prediction", "Test = Prediction"),
        col=c("black","red"), lty=c(-1,1), pch=c(1,-1), merge = TRUE)

dev.copy2eps(file="remove3lsf.eps")
```

The value of λ for the plots in Figure 2. will be explained later on in this section.

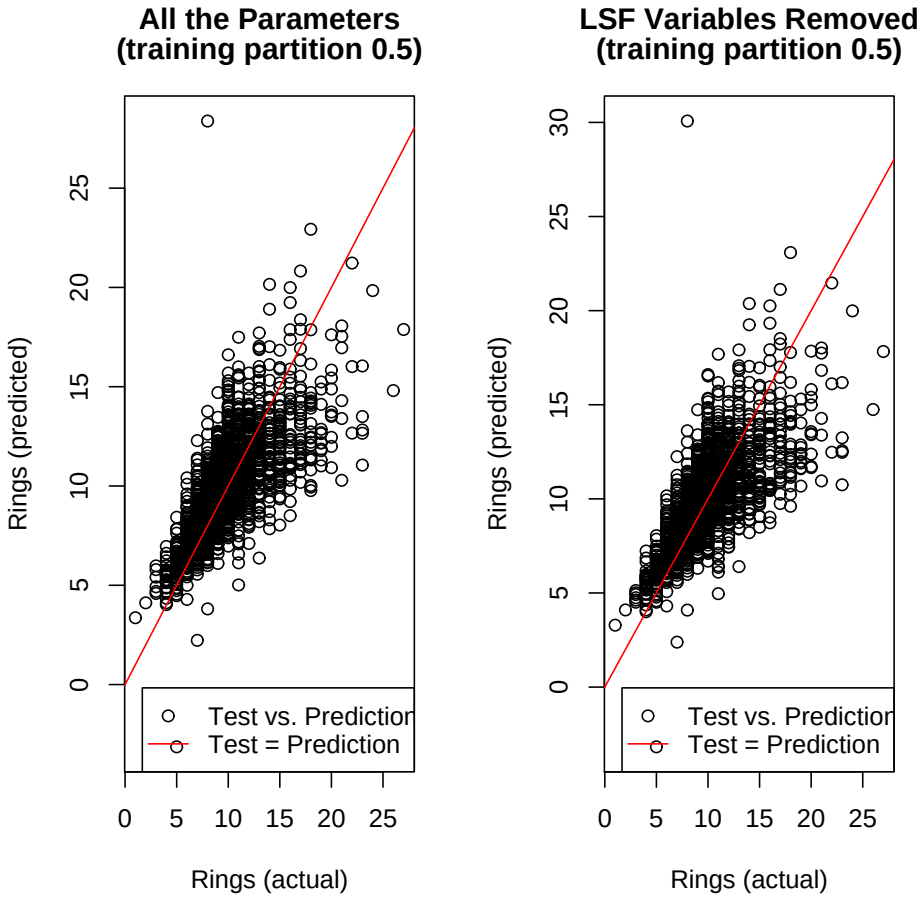

Figure 2: Model with & without the three LSFs. $\lambda = 0.8$

5.3 Effects of Training Partition and λ

To explore the effect of the training partition size, many plots were generated over the entire range of training partitions ranging from 0.005 to 0.9. From a practical standpoint, the area of most interest was when training partitions were relatively small. This is because in reality, good data to build models upon is usually scarce.

A typical looking plot in the low training partition region would look like the plot in Figure 3.

The code used to generate the plot in Figure 3. is shown here. Notice how the functions *llsMake* and *llsUse* are utilized in this code.

```
# creates a 3 column dataframe - 1st col is lambdas, 2nd col is RMSE of
# the training data, 3rd col is RMSE of testing data
# should not be standardized before passing them in because this is done
# within the llsMake and llsUse functions that are called within this
# function.
```

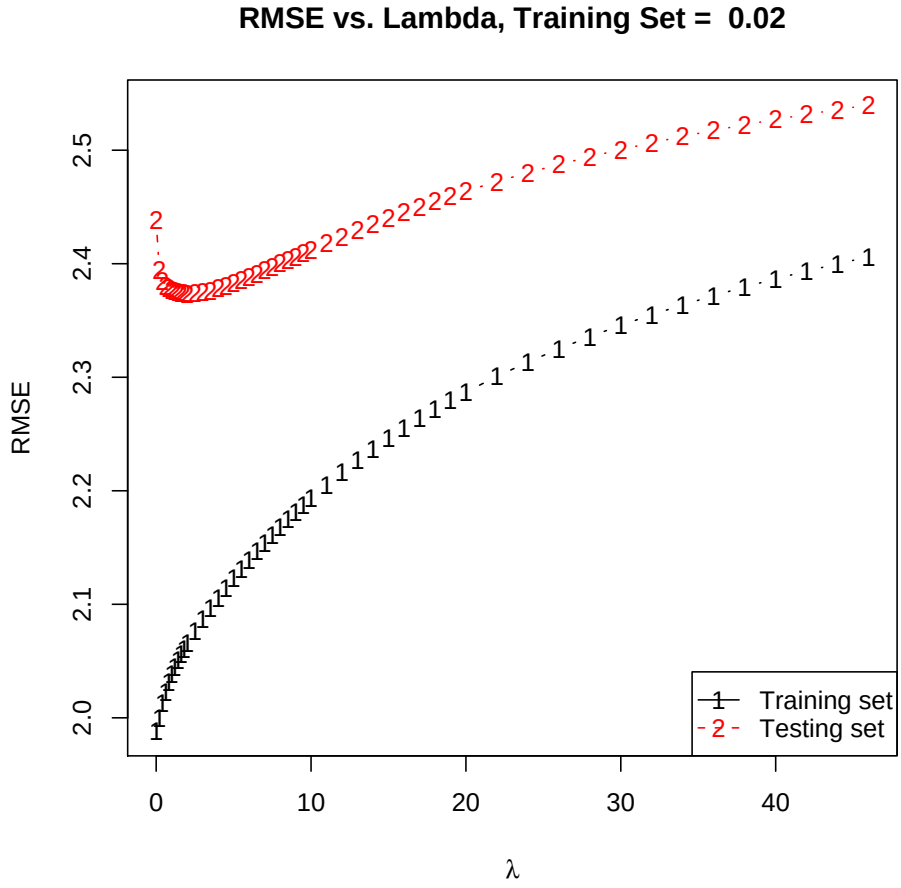

Figure 3: Model with & without the three LSFs.

```
# Xtrain & Xtest are the training and test inputs respectively
# Ytrain & Ytest are the training and test target outputs respectively
# lambdas is a vector of lambdas which we want to test
rmseVsLambda <- function(Xtrain, Xtest, Ytrain, Ytest, lambdas) {

   # initialize results vector which will be built in the for loop
   results <- c()
   # calc weights, calc model prediction, and calc rmse for each lambda
   # append the results vector after each loop pass
   for(lam in lambdas) {
      # calculate the weights for a given lambda
      w <- llsMake(Xtrain, Ytrain, lam)
      pred <- llsUse(w, Xtest)
      results <- rbind(results, c(lam, sqrt(mean((llsUse(w,Xtrain) - Ytrain)^2)),
```

```
                                        sqrt(mean((pred - Ytest)^2))))
   }
   #results <- as.matrix(results)
   results
}
```

In order to explore a range of training partitions, the following code was written to do a number of iterations on each training partition (200). This needed to be done so that we could average the results because the $RMSE$ vs. λ curve could vary significantly within the same training partition due to the random nature of the selection of members for the training and test partitions.

In order to accomplish this task, the following R function was written. This function was built from functions which have been described earlier:

```
# analyzes the effect of lambdas and training partitions on RMSE
# std_data is the matrix containing the input and output param's,
# lambdas is a vector of lambdas to test over
# outputs a matrix were the 1st col. are the lambdas that were passed in
# two col's are then added fro every training partition: one for the RMSE
# (training) and one for the RMSE (testing).
partition_exps <- function(std_data, lambdas) {
   init <- matrix(0,length(lambdas),2)  # used to initialize resultsum
   # initialize the training set fractions
   tsetfracs <- c(0.005,0.01,0.02,0.04,0.06,0.08,0.1,0.2,0.3)
   # initialize the overall results with just the lambdas
   resultall <- matrix(c(lambdas),length(lambdas),1)
   colnames(resultall)[1] = "lambda"
   # outer loop: training set partitions
   for(tsf in tsetfracs) {
      colnames(init)[1:2] <- c(paste("RMSE(trn)",tsf),
                               paste("RMSE(tst)",tsf))
      resultsum <- init
      # inner loop: 200 random iterations on each training set fraction (TSF)
      for(parts in 1:200) {
         # establish the random set of training rows based on the TSF
         trainRows <- trainSetRows(std_data, tsf)
         # the test set was everything that wasn't the training set...
         testRows <- testSetRows(seq(1:nrow(std_data)),trainRows)

         # load the independent var's for training and test sets
         Xtrain <- std_data[trainRows, 1:9]
         Xtest <- std_data[testRows, 1:9]
         # load the target values for training and test sets
         Ttrain <- std_data[trainRows, 10, drop=FALSE]
         Ttest <- std_data[testRows, 10, drop=FALSE]
         # now calc. the results...
         results <- rmseVsLambda(Xtrain, Xtest, Ttrain, Ttest, lambdas)
         # tally our results
```

```
      resultsum <- resultsum + results[,2:3]
    }  # close inner loop

    # component division to turn resultsum into averages
    resultavg <- resultsum / 200
    resultall <- cbind(resultall, resultavg)
  }  # close outer loop

  resultall
}  # close partition_exps
```

The plotting all 9 of these graphs together is shown in Figure 4. These graphs were scaled on both x and y axis so that they could be readily compared.

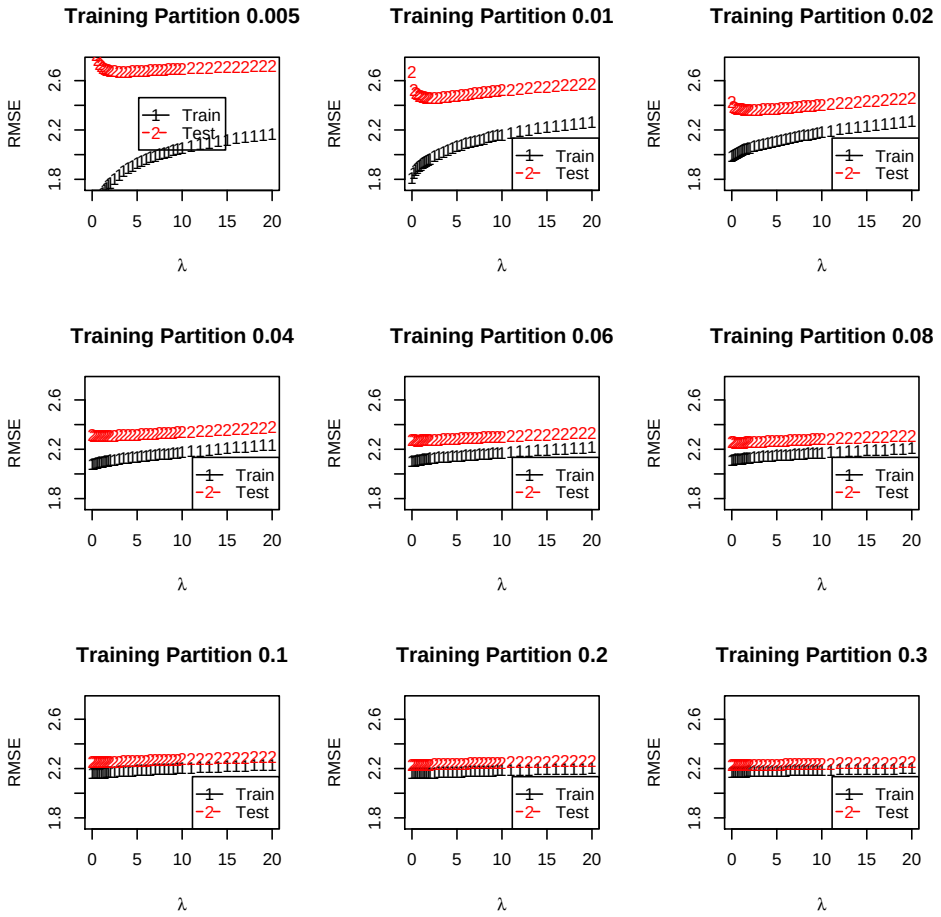

Figure 4: Effect of Training Partition and λ on RMSE

The code used to generate this figure is shown here.

```
# plots the effect of training partition on lambda
plot_lambda_study <- function(results) {
tparts <- matrix(c(0.005,0.01,0.02,0.04,0.06,0.08,0.1,0.2,0.3,seq(2,18,by=2)),
                 2,9,byrow=TRUE)
   par(mfrow = c(3,3))  # set up output of 3x3 matplots
   for(i in 1:9) {
      legpos <- "bottomright"
      if(i==1)
         legpos <- "center"
      traincol <- tparts[2,i]
      testcol <- traincol + 1
      ylim <- range(results[,2:ncol(results)])  # find the RMSE range
      # override the ylim to squeeze range a bit...
      ylim[2] <- 2.75
      ylim[1] <- 1.75
      #label <- c("RMSE vs. Lambda","training partition",tparts[1,i])
      matplot(results[,1],results[,traincol:testcol],type="b",
         xlab=expression(lambda),
         ylab="RMSE",
         ylim=ylim,
         main=paste("Training Partition", tparts[1,i]))
      legend(legpos,c("Train", "Test"),
         col=c("black","red"),
         pch=c("1","2"), lty=c(1,2))
   }
   dev.copy2eps(file="tparts_study3x3a.eps")
}
```

The plots in Figure 4. seem to suggest that the λ which minimizes RMSE for the test set appears to flatten out at about 0.06 training partition and thereafter. The minimum RMSE also appears to be falling as training fraction increases. Both of these observations can be seen more clearly in Figure 5.

The right side plot in Figure 5. suggests that a λ of 0.8 be used as this minimizes RMSE for a large majority of the existing data.

6 Results

The ultimate question to be asked is "*How good is the model?*". To move toward an answer, a 0.5 training partition was selected and the model was used to predict each partition using a λ of 0.8 which was derived from Figure 5. The actual vs. the predicted results were plotted against each other as shown in Figure 6.

The 45 degree line is used as a reference in each plot. The accuracy of the model can be qualitatively assessed by how well the points in these plots fall on this reference line. As we can see, the accuracy is marginal.

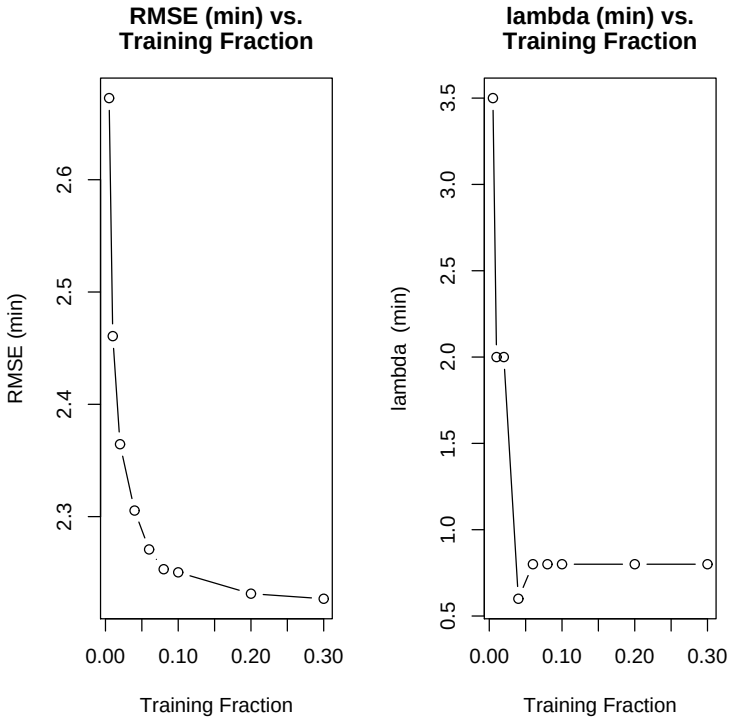

Figure 5: Effect of Training Partition and λ on RMSE. (200 iterations)

7 A Personal Curiosity: The Impact of Training Partition on Bias

Since the bias was the strongest factor in the regression model, it seemed reasonable to ask the question: "*Is there a relationship between training partition and bias?*"

Bias was the only fitted model parameter that was not penalized by λ, so it was logical to assume that bias will not be impacted by λ. This left training partition. So to get some insight into this, the following plot was created:

Figure 7. was generated from the following code. At first it looked like there this plot suggested that there might be such relationship. But upon closer inspection of the magnitude of change in the y-axis, it appears that bias is relatively unaffected by training partition.

```
# returns an Nx2 matrix, where N=training partitions
# std_data = the abalone data table
bias_exps <- function(std_data,
             tpart=c(seq(0,0.3,by=0.02),seq(0.35,0.9,by=0.05)),
             lam=0.8, itr=100) {
   init <- matrix(0,length(tpart),1)  # used to initialize resultsum
   colnames(init) <- c("bias avg")
```

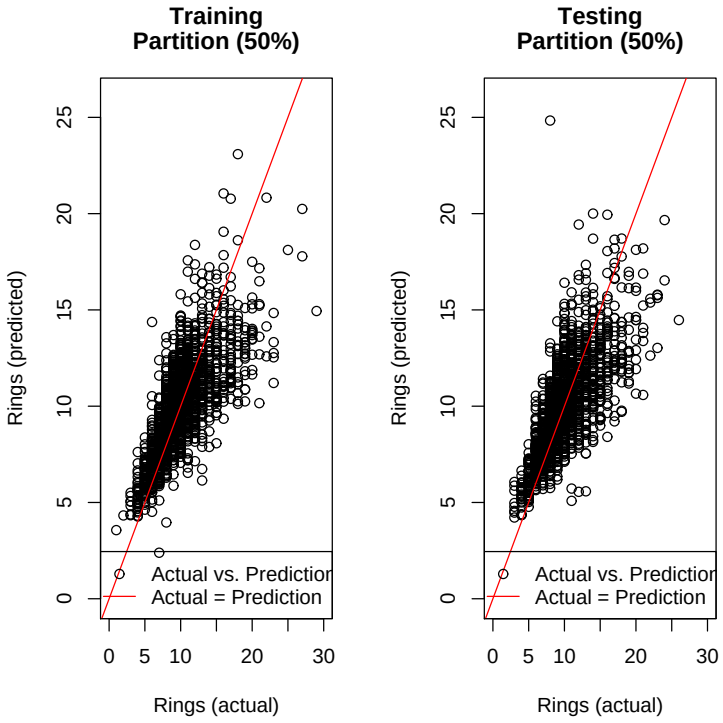

Figure 6: Effect of Training Partition and λ on RMSE. (200 iterations)

```
# initialize the overall results with just the tparts
resultall <- matrix(c(tpart,seq(1:length(tpart))),length(tpart),2)
colnames(resultall) = c("tpart","bias avg")
# outer loop: training set partitions
lrow = 0
for(tsf in tpart) {
   lrow <- lrow+1
   resultsum <- init
   wsum <- 0
   # inner loop: 200 random iterations on each training set fraction (TSF)
   for(parts in 1:itr) {
      # establish the random set of training rows based on the TSF
      trainRows <- trainSetRows(std_data, tsf)
      # the test set was everything that wasn't the training set...
      testRows <- testSetRows(seq(1:nrow(std_data)),trainRows)
      # load the independent var's for training and test sets
      Xtrain <- std_data[trainRows, 1:9]
```

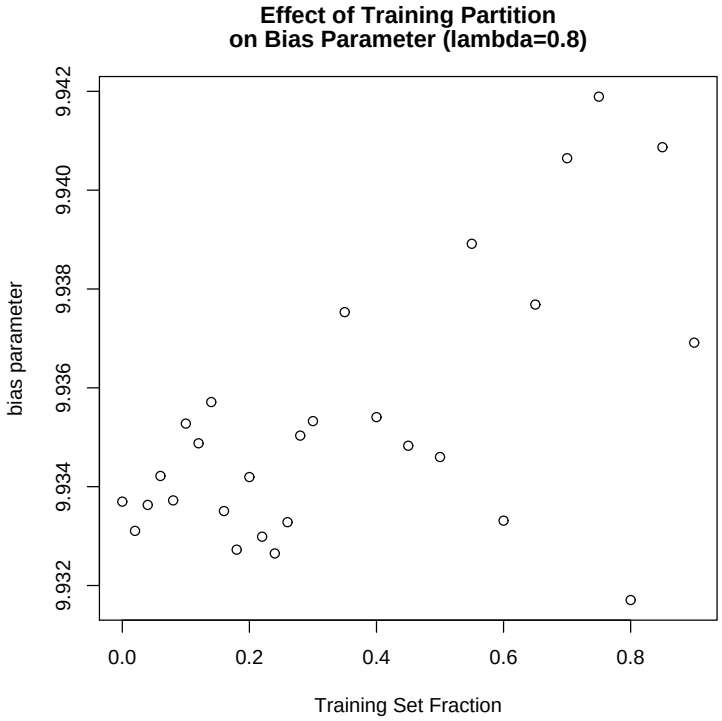

Figure 7: Effect of Training Partition and λ on RMSE. (200 iterations)

```
      Xtest <- std_data[testRows, 1:9]
      # load the target values for training and test sets
      Ttrain <- std_data[trainRows, 10, drop=FALSE]
      Ttest <- std_data[testRows, 10, drop=FALSE]
      # now calc. the weights
      w <- llsMake(Xtest, Ttest, lam)
      # tally the bias...
      wsum <- wsum + w[1,1]
   }  # close inner loop

   # component division to turn resultsum into averages
   wavg <- wsum / itr
   # overwrite the 2nd col with the bias averages
   resultall[lrow,2] <- wavg
}  # close outer loop

resultall
```

```
}  # close bias_exps
```

8 Discussion

In addition to points already made, the dependencies noted in section 5.1 actually make sense from a practical point of view. For example, given that abalone are basically ellipsoid in shape, there is a well defined relationship between *Length* and *Diameter*.

The impact of training set partition and λ can be seen clearly in Figure 5. A λ that minimized RMSE dramatically between 0 and 0.04 training fraction and levels off. This makes sense because small training fractions represent sparse data in the real world. Under such conditions, the model is being asked to make predictions based on very little training. It does appear however that the model "learns" quickly.

In our analysis, RMSE approached a value of 2.2 for training fractions $> 10\%$. Given that a majority of the ring readings are below 15 (bottom right histogram in Figure 1.), this represents an error in the range of about 15% or more which, depending on the application, seems to be marginally accurate.

References

[1] *W*ikipedia `http://en.wikipedia.org/wiki/Abalone`

[2] *M*onterey Bay Aquarium `http://www.mbayaq.org/cr/SeafoodWatch/web/sfw_factsheet.aspx`

[3] *U*CI Machine Learning Repository Website `http://archive.ics.uci.edu/ml/machine-learning-databases/abalone/`