CS545: Linear Modeling

Chuck Anderson

Department of Computer Science
Colorado State University

Fall, 2009
Outline

R Tips for Linear Modeling

Backing up Files in Unix

Collinearity
Standardizing Inputs

- Standardize attribute values (each has mean zero, unit variance):

  - Calculate mean of each attribute for training data.
  - Calculate standard deviation of each attribute for training data.
  - Subtract means and divide by stdevs, column by column:
    \[
    X_{train} = (X_{train} - \text{matrix}(\text{means}, \text{nrow}(X_{train}), \text{ncol}(X_{train}), \text{byrow}=\text{TRUE})) / \text{matrix}(\text{stdevs}, \text{nrow}(X_{train}), \text{ncol}(X_{train}), \text{byrow}=\text{TRUE})
    \]
  - To standardize testing data, use means and stdevs calculated from training data.
    \[
    X_{test} = (X_{test} - \text{matrix}(\text{means}, \text{nrow}(X_{test}), \text{ncol}(X_{test}), \text{byrow}=\text{TRUE})) / \text{matrix}(\text{stdevs}, \text{nrow}(X_{test}), \text{ncol}(X_{test}), \text{byrow}=\text{TRUE})
    \]
Standardizing Inputs

- Standardize attribute values (each has mean zero, unit variance):
  - Calculate mean of each attribute for **training data**.

```
means <- colMeans(Xtrain)
```

- Calculate standard deviation of each attribute for training data.

```
stdevs <- sd(Xtrain)
```

- Subtract means and divide by stdevs, column by column.

```
Xtrain <- (Xtrain - matrix(means, nrow(Xtrain), ncol(Xtrain), byrow=TRUE)) / matrix(stdevs, nrow(Xtrain), ncol(Xtrain), byrow=TRUE)
```

- To standardize testing data, use means and stdevs calculated from training data.

```
Xtest <- (Xtest - matrix(means, nrow(Xtest), ncol(Xtest), byrow=TRUE)) / matrix(stdevs, nrow(Xtest), ncol(Xtest), byrow=TRUE)
```
Standardizing Inputs

- Standardize attribute values (each has mean zero, unit variance):
  - Calculate mean of each attribute for training data.
    ```r
    means <- colMeans(Xtrain)
    ```
  - Calculate standard deviation of each attribute for training data.
    ```r
    stdevs <- sd(Xtrain)
    ```

- To standardize testing data, use means and stdevs calculated from training data.
  ```r
  Xstest <- (Xtest - matrix(means, nrow(Xtest), ncol(Xtest), byrow=TRUE)) / matrix(stdevs, nrow(Xtest), ncol(Xtest), byrow=TRUE)
  ```
Standardizing Inputs

- Standardize attribute values (each has mean zero, unit variance):
  - Calculate mean of each attribute for training data.
    
    ```r
    means <- colMeans(Xtrain)
    ```
  - Calculate standard deviation of each attribute for training data.
    
    ```r
    stdevs <- sd(Xtrain)
    ```
  - Subtract means and divide by stdevs, column by column
    
    ```r
    Xstrain <- (Xtrain - matrix(means, nrow(Xtrain), ncol(Xtrain), byrow=TRUE)) / matrix(stdevs, nrow(Xtrain), ncol(Xtrain), byrow=TRUE)
    ```
  - To standardize testing data, use means and stdevs calculated from training data.
    
    ```r
    Xtest <- (Xtest - matrix(means, nrow(Xtest), ncol(Xtest), byrow=TRUE)) / matrix(stdevs, nrow(Xtest), ncol(Xtest), byrow=TRUE)
    ```
Standardizing Inputs

- Standardize attribute values (each has mean zero, unit variance):
  - Calculate mean of each attribute for training data.
    \[
    \text{means} <- \text{colMeans}(X\text{train})
    \]
  - Calculate standard deviation of each attribute for training data.
    \[
    \text{stdevs} <- \text{sd}(X\text{train})
    \]
  - Subtract means and divide by stdevs, column by column
    \[
    X\text{train} <- (X\text{train} - \text{matrix(}\text{means, nrow(X\text{train}), ncol(X\text{train}), byrow=TRUE}))) \div \text{matrix(}\text{stdevs, nrow(X\text{train}), ncol(X\text{train}), byrow=TRUE}))
    \]
  - To standardize testing data, use means and stdevs calculated from training data.
    \[
    X\text{test} <- (X\text{test} - \text{matrix(}\text{means, nrow(X\text{test}), ncol(X\text{test}), byrow=TRUE}))) \div \text{matrix(}\text{stdevs, nrow(X\text{test}), ncol(X\text{test}), byrow=TRUE}))
    \]
Must keep track of means and stdevs from training data. Can do as variables returned from standardize function:

```r
standardize <- function(X, means=apply(X,2,mean), stdevs=apply(X,2,sd), returnParms=FALSE) {
  ## X is nSamples by nInputComponents
  stdevs[stdevs==0] <- 1
  N <- nrow(X)
  p <- ncol(X)
  X <- (X - matrix(rep(means,N),N,p,byrow=TRUE))/
      matrix(rep(stdevs,N),N,p,byrow=TRUE)
  if (returnParms)
    list(data=X, means=means, stdevs=stdevs)
  else
    X
}
```

used like

```r
tp <- standardize(Xtrain, returnParms=TRUE)
Xstrain <- tp$1data
Xstest <- standardize(Xtest, tp$1means, tp$1stdevs)
```
Or can "store" means and stddevs as local variables bound to values inside a newly created function

```r
makeStandardizeF <- function(X) {
  if (missing(X)) {
    cat("Usage: 
    standardize <- makeStandardizeF(X) ## X is nSamples x nDimensions
    Xs <- standardize(X) 
    X2s <- standardize(X2)
"
    return( invisible() )
  }
  ## X is nSamples x nDimensions
  mu <- colMeans(X)
  sigma <- sd(X) ##sd should be named colSds

  function(newX) {
    nr <- nrow(newX)
    nc <- ncol(newX)
    (newX - matrix(mu,nr,nc,byrow=TRUE)) / matrix(sigma,nr,nc,byrow=TRUE)
  }
}

used like

standardize <- makeStandardizeF(Xtrain)
Xstrain <- standardize(Xtrain)
Xstest <- standardize(Xtest)
```
What should `makeLLS` return?

- Certainly want the weights returned. After all, that is the model. What else?

```r
return(list(weights = w, standardize = standardize))
```

Use like

```r
model <− makeLLS(Xtrain, Ttrain, lambda)
predictions <− useLLS(model, Xtest)
```
What should \texttt{makeLLS} return?

- Certainly want the weights returned. After all, that is the model. What else?
- The means and stddevs should also be associated with this model. Different models will have different weights and different standardization parameters.
What should `makeLLS` return?

- Certainly want the weights returned. After all, that is the model. What else?
- The means and stdevs should also be associated with this model. Different models will have different weights and different standardization parameters.
- How would `makeLLS` return all of this?

```r
return(list(weights=w, standardize=standardize))
```

Use like

```r
model <- makeLLS(Xtrain,Ttrain,lambda)
predictions <- useLLS(model,Xtest)
```
What should `makeLLS` return?

- Certainly want the weights returned. After all, that is the model. What else?
- The means and stdevs should also be associated with this model. Different models will have different weights and different standardization parameters.
- How would `makeLLS` return all of this?

```
return(list(weights=w, standardize=standardize))
```
What should \texttt{makeLLS} return?

- Certainly want the weights returned. After all, that is the model. What else?
- The means and stdevs should also be associated with this model. Different models will have different weights and different standardization parameters.
- How would \texttt{makeLLS} return all of this?

\begin{verbatim}
return( list( weights=w, standardize=standardize ) )
\end{verbatim}

- Use like

\begin{verbatim}
model <- makeLLS(Xtrain, Ttrain, lambda)
predictions <- useLLS(model, Xtest)
\end{verbatim}
What should `makeLLS` return?

- Certainly want the weights returned. After all, that is the model. What else?

- The means and stdevs should also be associated with this model. Different models will have different weights and different standardization parameters.

- How would `makeLLS` return all of this?

  ```r
  return( list( weights=w, standardize=standardize ) )
  ```

- Use like

  ```r
  model <- makeLLS(Xtrain, Ttrain, lambda)
predictions <- useLLS(model, Xtest)
  ```

- Inside `useLLS` how would you use `model`?
What should `makeLLS` return?

- Certainly want the weights returned. After all, that is the model. What else?
- The means and stdevs should also be associated with this model. Different models will have different weights and different standardization parameters.
- How would `makeLLS` return all of this?

```r
return( list( weights=w, standardize=standardize ) )
```

Use like

```r
model <- makeLLS(Xtrain,Ttrain,lambda)predictions <- useLLS(model,Xtest)
```

Inside `useLLS` how would you use `model`?

Say `useLLS` has arguments named `model` and `X`:

```r
Xs <- model$standardize(X)predictions <- Xs %*% model$weights
```
Collecting and Combining Multiple Results

- We often want to repeat a calculation a number of times using different parameter values, like values of $\lambda$ and of training set fraction. So, you might use a for loop like

```r
for (trainf in c(0.2, 0.4, 0.6, 0.8, 0.9)) {
  for (repi in 1:200) {
    for (lambda in seq(0,10,by=0.5)) {
      ## do calculation here using trainf and lambda to obtain
      ## trainRMSE and testRMSE
    }
  }
}
```

Can try to do sums of RMSE’s so you can calculate average later. But, let’s use that cheap memory, and just collect each result in a new row in a matrix.

```r
results <- rbind(results, c(trainf,lambda, trainRMSE, testRMSE))
```

Don’t forget to initialize

```r
results <- c()
```
Collecting and Combining Multiple Results

- We often want to repeat a calculation a number of times using different parameter values, like values of $\lambda$ and of training set fraction. So, you might use a for loop like

```r
for (trainf in c(0.2, 0.4, 0.6, 0.8, 0.9)) {
  for (repi in 1:200) {
    for (lambda in seq(0,10,by=0.5)) {
      ## do calculation here using trainf and lambda to obtain
      ## trainRMSE and testRMSE
    }
  }
}
```

- Can try to do sums of RMSE’s so you can calculate average later. But, let’s use that cheap memory, and just collect each result in a new row in a matrix.

```r
## do calculation here using trainf and lambda to obtain
## trainRMSE and testRMSE
results <- rbind(results,
                 c(trainf,lambda, trainRMSE, testRMSE))
```
Collecting and Combining Multiple Results

- We often want to repeat a calculation a number of times using different parameter values, like values of $\lambda$ and of training set fraction. So, you might use a for loop like

```r
for (trainf in c(0.2, 0.4, 0.6, 0.8, 0.9)) {
    for (repi in 1:200) {
        for (lambda in seq(0, 10, by=0.5)) {
            ## do calculation here using trainf and lambda to obtain
            ## trainRMSE and testRMSE
        }
    }
}
```

- Can try to do sums of RMSE’s so you can calculate average later. But, let’s use that cheap memory, and just collect each result in a new row in a matrix.

```r
## do calculation here using trainf and lambda to obtain
## trainRMSE and testRMSE
results <- rbind(results,
                 c(trainf, lambda, trainRMSE, testRMSE))
```

- Don’t forget to initialize `results` before you start the for loops.
Now, the matrix has many rows (200) for each pair of (trainf, lambda) values. How can we calculate the means of those 200 values? Check out `?unique`.

```r
> results
[1,]  0.2  0.1  3.2  3.6
[2,]  0.2  0.5  5.3  3.2
[3,]  0.2  0.1  5.5  3.3
> unique(results[,1:2])
   [,1] [,2]
[1,]  0.2  0.1
[2,]  0.2  0.5
```

So, we can use `unique` to identify unique combinations of parameter values in our results matrix. Generate a boolean mask to select rows for one unique combination.

```r
uniqueCombos <- unique(results[,1:2])
oneCombo <- uniqueCombos[1,]
mask <- apply(results[,1:2], 1, function(ps) all(ps==oneCombo))
> results[mask,]
[1,]  0.2  0.1  3.2  3.6
[2,]  0.2  0.1  5.5  3.3
> colMeans(results[mask,3:4])
[1] 4.35 3.45
```
Now, the matrix has many rows (200) for each pair of (trainf, lambda) values. How can we calculate the means of those 200 values? Check out `unique`.

```r
> results
[1,]  0.2  0.1  3.2  3.6
[2,]  0.2  0.5  5.3  3.2
[3,]  0.2  0.1  5.5  3.3
> unique(results[,1:2])
  [,1] [,2]
[1,]  0.2  0.1
[2,]  0.2  0.5
```

So, we can use `unique` to identify unique combinations of parameter values in our `results` matrix.
Now, the matrix has many rows (200) for each pair of (trainf, lambda) values. How can we calculate the means of those 200 values? Check out `unique`.

```r
> results
[1,] 0.2 0.1 3.2 3.6
[2,] 0.2 0.5 5.3 3.2
[3,] 0.2 0.1 5.5 3.3
> unique(results[,1:2])
[,1] [,2]
[1,] 0.2 0.1
[2,] 0.2 0.5
```

So, we can use `unique` to identify unique combinations of parameter values in our `results` matrix.

Generate a boolean mask to select rows for one unique combination.

```r
> uniqueCombos <- unique(results[,1:2])
> oneCombo <- uniqueCombos[1,]
> mask <- apply(results[,1:2], 1, function(ps) all(ps == oneCombo))
> results[mask,]
[1,] 0.2 0.1 3.2 3.6
[2,] 0.2 0.1 5.5 3.3
> colMeans(results[mask,3:4])
[1] 4.35 3.45
```
Outline

R Tips for Linear Modeling

Backing up Files in Unix

Collinearity
Making Backups in Unix

- Simple shell script linked to in CS545 Schedule web page.

```bash
Simple shell script linked to in CS545 Schedule web page.
```
Making Backups in Unix

- Simple shell script linked to in CS545 Schedule web page.
- Copy into your ~/bin directory and make it executable by chmod a+x backup.
Making Backups in Unix

- Simple shell script linked to in CS545 Schedule web page.
- Copy into your ~/bin directory and make it executable by chmod a+x backup.
- Add your ~/bin directory to your PATH shell variable
  ```bash
  export PATH=$PATH:/s/parsons/e/fac/anderson/bin:
  ```
Making Backups in Unix

- Simple shell script linked to in CS545 Schedule web page.
- Copy into your ~/bin directory and make it executable by `chmod a+x backup`.
- Add your ~/bin directory to your PATH shell variable.
  ```bash
  export PATH=$PATH:/s/parsons/e/fac/anderson/bin:
  ```
- Then, to backup files just do
  ```bash
  > backup *.R
  Creating BACKUP directory.
  Created the following BACKUP files:
  BACKUP/highd.R.2009-09-10-11-48-46
  BACKUP/mpgCollinearity.R.2009-09-10-11-48-46
  BACKUP now contains 2 files
Outline

R Tips for Linear Modeling

Backing up Files in Unix

Collinearity
Collinearity of Attributes

What if one attribute is a linear function of another attribute, say pressure $= -2$ temperature? How might this affect their weight values?
Collinearity of Attributes

- What if one attribute is a linear function of another attribute, say pressure = \(-2\) temperature? How might this affect their weight values?
- Given any linear model (weights), the model will make exactly the same predictions if the weight value for temperature is multiplied by \(a\) and the weight value for pressure is multiplied by \(-2a\).
Collinearity of Attributes

- What if one attribute is a linear function of another attribute, say pressure = −2 temperature? How might this affect their weight values?
- Given any linear model (weights), the model will make exactly the same predictions if the weight value for temperature is multiplied by $a$ and the weight value for pressure is multiplied by $-2a$.
- How can we create a new attribute that is a linear function of another, and vary how close it is to linearly dependent?
Collinearity of Attributes

- What if one attribute is a linear function of another attribute, say pressure = $-2$ temperature? How might this affect their weight values?

- Given any linear model (weights), the model will make exactly the same predictions if the weight value for temperature is multiplied by $a$ and the weight value for pressure is multiplied by $-2a$.

- How can we create a new attribute that is a linear function of another, and vary how close it is to linearly dependent?

- Add noise to a linear function. Say $X$ has 7 columns.

```r
X <- cbind(X, -2 * X[,7] + rnorm(nrow(X), 0, 0.1))
```
Doing this for the mpg data, and varying the standard deviation $\sigma$ of the noise, we get these weight values.

\[
\begin{array}{cccccccc}
-5.0 & -4.5 & -4.0 & -3.5 & -3.0 & -2.5 & -2.0 \\
-6000 & -2000 & 0 & 2000 & 4000 & 6000 \\
\end{array}
\]

$\log \sigma = \text{degree of independence between 8 and 9}$