Preprocessing Data

Notes for CS 6232: Data Analysis and Visualization
Georgia Tech (Dr. Guy Lebanon), Fall 2016
as recorded by Brent Wagenseller

Loading Datasets Needed for this Lesson

To install the player dataset:
install.packages("plyr")
To install the movies dataset:
install.packages("ggplot2movies")
To use the diamonds dataset:
library("ggplot2")
To use the MASS library:
library(MASS)
Installing robustHD

- The RobustHD instructions can also be found at: RobustHD install instructions
- To use the robustHD library:
  - first: install dependent packages

install.packages(c("MASS", "akima", "robustbase"))

  - second: install suggested packages

install.packages(c("cobs", "robust", "mgcv", "scatterplot3d", "quantreg", "rrcov", "lars", "pwr", "trimcluster", "parallel", "mc2d", "psych", "Rfit"))

  - third: install an additional package which provides some C functions

install.packages("devtools")
library("devtools")

NOTE: The following "mrxiaohe/WRScpp" ONLY works on MAC – see below for other options

install_github("mrxiaohe/WRScpp")

This may need an additional parameter in the command: force=TRUE
The above may not work on windows – if not try this:

- Download the zipped package at https://github.com/mrxiaohe/WRScppWIN (click the 'clone or download' button)
- Save it to your Downloads folder (or working directory, but use ~/ below instead of ~/Downloads/)

install.packages("~/Downloads/WRScppWin-master.zip", repos = NULL, type = "win.binary")

The above may not work on Linux – it not try this (see more info at https://github.com/JoeJohnston/WRScppLin)

install.packages("WRS", repos="http://R-Forge.R-project.org", type="source")
install.packages(c("RcppArmadillo", "devtools"))fourth: install WRS
library("devtools")
install_github(repo="JoeJohnston/WRScppLin")

  - Fourth, install WRS

install_github("nicebread/WRS", subdir="pkg")
install.packages("robustHD")
library("robustHD")
To use the melt command:
library(reshape2)
To use the tips dataset:
library(reshape2)
To use the Ozone dataset

library(plyr)

Lesson Preview

We will be learning about data preprocessing

Data pre-processing takes the longest amount of time in data analytics and is the most crucial

Poorly prepared data can lead to wrong conclusions

Goals

- Learn how to handle missing data
- Learn how to handle outliers
- Learn standard data manipulation techniques

Missing Data

The presence of no data or missing data can sometimes convey some information about the data in itself which may be useful in the data analysis dask

Data can be missing for many reasons

- Corruption during its transfer or storage
  - Examples
    - An old record has years of abuse
    - Data spread over multiple files and some files are erased / corrupt
    - Movie recommendation systems may have movies with no reviews / users don't review all movies they have seen
    - Longitudinal studies – subjects may drop out of the study
    - Sensors may fail in sensor data
    - Users taking user surveys may not answer all questions due to privacy concerns
- Some instances in the data collection process were skipped due to difficulty or price associated with obtaining the data

We identify samples as rows and variables / features as columns

- Samples may occasionally be missing for certain features

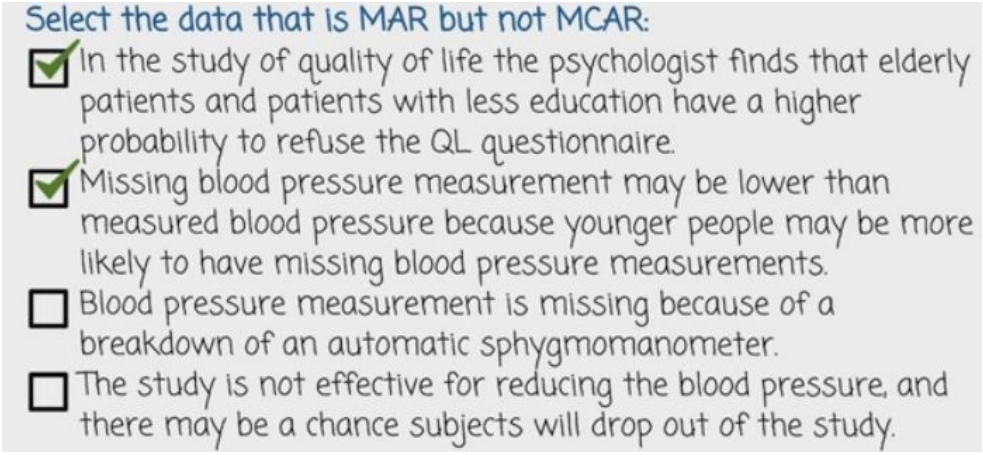
MCAR (**Missing Completely at Random**) is a concept that states the probability of an observation being missing does not depend on observed or unobserved measurements

- This can be violated; for example if a specific movie is not popular, it won't get the volume of ratings that a more popular movie would see (Think 'The Godfather' vs 'Garbage Pail Kids: The Movie')

MAR (**Missing at Random**) states given the observed data, the probability that data is missing does not depend on the unobserved data

- Example
  - A user survey may survey gender, race, and income. Gender and race are not as objectionable as income, so the first two features may be fully filled out while income may only have a few entries
    - IF the missing data depends on gender and race (two features included) this is considered to be MAR
    - IF the missing data depends on features not listed in the data – or the feature itself (aka income) – this is not MAR
      - This is likely the case for this survey

See examples below to spot different scenarios that are MAR but not MCAR (selected are MAR, not selected are MCAR)



- The first statement is MAR assuming the specified dependencies are recorded in the survey (as stated) are recorded in the survey and there are no additional dependencies
- The second is pretty much the same as the first
- The third statement is missing completely at random since the device breaking does not depend on observed or unobserved data (randomness that has no seemingly rhyme or reason)
- The fourth statement is MCAR if the subject drops out with fixed probabilities OR its not MAR if the likelihood of dropping out depends on patients unobserved variables

Handling Missing Data

- Most methods are designed to work with fully observed data
- There are some general ways to convert missing data to non-missing data
  - The simplest – but not the best – is to simply remove the rows with missing data
  - Another way is to replace the values with the median or mean (or, if it's a category, the most used category)
  - The third method would be to build a probability model for the variable that is missing, sample for that model's values, and insert the randomly sampled values in place of the missing entries
    Basically, get the weight distribution of each answer across all actual given answers and – using that distribution – randomly distribute the values to the missing segments
    This is called **imputation**
- In the case of MCAR, all three techniques are reasonable in that they will not introduce systematic errors
  - some are better, though
    imputation is considered better than substitution with mean or median
    substitution is better than removing instances altogether
- In the case of MAR of Non-MAR, the techniques may introduce systematic bias into the data analysis process
  - The first and second method introduce systematic bias
  - Imputation may or may not, depending on how carefully the probability model is constructed

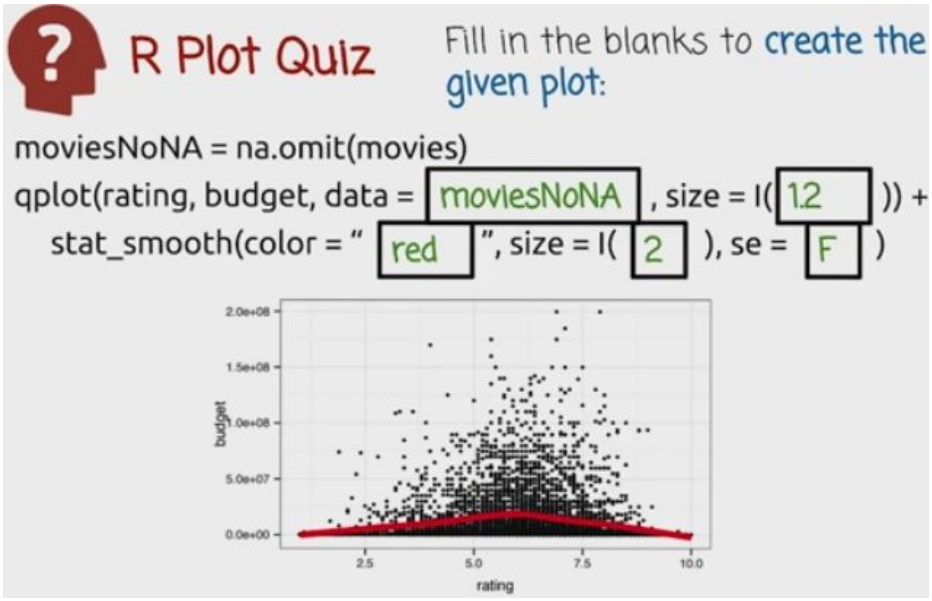
Missing Data and R

- R usually stores data in dataframes where rows are samples and columns are features
- R uses the 'NA' keyword to represent missing data
- Functions to find missing data
  - is.na
    Returns TRUE for missing data, FALSE otherwise
    For example, this computers the frequency of missing budget entries

mean(is.na(movies$budget))

- complete.cases()
  Returns a vector whose components are FALSE for all samples
  TRUE otherwise
- na.omit()
  Returns a new dataframe omitting all samples containing missing values
  Example
  moviesNoNA = na.omit(movies)
- na.rm
  Some functions have this argument, which if set to TRUE, changes the function behavior so that it proceeds to operate on the supplied data after removing all dataframe rows with missing values

R Plot Quiz



- This eliminates all rows with missing data and then plots it
- ‘se=F’ means the standard error bars will not be produced

Outliers

Outliers are extreme values that are bigger or smaller than the rest of the data
Two types of outliers

- Corrupted values
  - Human mistakes
  - Other corruption
- Unlikely values given our modelling assumption

In both cases, data analysis based on outliers may result in drastically wrong conclusions

- Corrupted values are the worst, but unlikely values can do this as well

Robustness describes a lack of sensitivity of data analysis procedures to outliers

- The mean is typically non-robust
  - It is effected by outliers
  - As outliers move to infinity, so does the mean
- The median is typically a robust procedure
  - Medians remain in place, regardless of outliers

Two types of uncertainty / bias

- **Bias**
  - This seems to be a problem with the mean and outliers

    - Could be system error if the data is corrupted
- **Precision** (uncertainty)
    - Also referred to as random error
    - This is the deviation of the data from the mean

Dealing with Outliers

- **Truncating**
    - Removing all values deemed as outliers
- **Winsorization**
    - Shrink outliers to border of main parts of data
    - Removes the outliers and replaces them with the most extreme data points that remain after removing the outliers
- Robustness
    - Analyze the data using a robust procedure
    - Keeps the outliers as they are but we choose robust procedures to analyze the data
        An example is using the median instead of the mean

Detecting Outliers

- Few ways
    - Values below the alpha percentile or above the 100-alpha percentile
        For a small alpha, such as 5 or 2 or even 1
        Alpha is just a variable here, its not an actual concept
        This method is very robust
    - Values more than c times standard deviation away from the mean
        C is a small integer, such as 1, 2, 3, or 5
- Note that in order for this to work well – or maybe even at all – the data needs to be Gaussian
- Another problem is: we need to use the mean or median for these numbers, but both the mean and median may be affected by the outliers
    - One way around this is to compute the mean and standard deviation after removing the most extreme values
    - Another way around this is to use the percentiles (alpha percentile as mentioned above)

Example R code that Winsorizes data

```
library(robustHD)
#Create the data to be Winsorized:
originalData <- c(1000.0000000  ,  1.8410249 ,  -0.7923505 ,   0.1776514  ,  0.4002135)
#Print the data to be Winsorized:
print(originalData[1:5])
winsorize(originalData[1:5])
```

- This has 1 outlier value of 1000 and the rest are 10 random floats with mean 0 and standard deviation 1
- The winsorize function replaces the outlier

Example: Write R code to remove data that is 5 std less than the mean and 5 std greater than the mean, where the std and mean are computed without extreme measurements

```
original_data <- rnorm(20)
#replace the 1st position with 1000 (so we have an outlier)
original_data[1] <- 1000
sorted_data <- sort(original_data)
#filter the top two and bottom two numbers out (assume they are outliers)
filtered_data <- original_data[3:18]
```

```
#come up with limits on the filtered data
lower_limit<-mean(filtered_data) - 5*sd(filtered_data)
upper_limit<-mean(filtered_data) + 5*sd(filtered_data)
#the following actually stores a list of booleans for each value in original_data; this comes up with TRUE or FALSE and tells if the data point belongs in the data
not_outlier_ind <- (lower_limit <original_data) & (original_data < upper_limit)
print not_outlier_ind
#save the list with no outliers
data_w_no_outliers = original_data[not_outlier_ind]
```

Skewness and Power Transformation

In many cases, data is drawn from a highly skewed distribution that is not well described by one of the common statistical distributions

- A **simple transformation** may map the data to a form that IS well described by common distributions (Gaussian, Gamma distribution, etc)
- A suitable model can then be fitted to the transformed data
- If necessary, the model can be inverted so predictions can be made on the original scale
  - Basically, we can go back to what we had

The Power Transformation family is frequently used to combat data that has a skewed distribution

$$f_\lambda(x) = \begin{cases} (x^\lambda - 1)/\lambda & \lambda > 0 \\ \log x & \lambda = 0 \\ -(x^\lambda - 1)/\lambda & \lambda < 0 \end{cases} \qquad x > 0, \quad \lambda \in \mathbb{R}.$$

- They replace a nonnegative data value x by f(x) parameterized by λ
- The definition of f(x) differs based on λ
  - See above what happens for different values of λ
- Its not a single transformation but a family of transformations
  - For every selection of λ, we define a different transformation
- The transformation maps x to x^ λ
  - The operation of subtracting 1 and dividing by λ makes $f_\lambda(x)$ continuous in both λ and in x
  - The transformation amounts to raising x to the power of λ with some small modifications of multiplication by a constant and addition of a constant
  - For λ greater than 1, the curve (transformation) is convex
    This is used to remove left skewness from the data
  - For λ less than 1, the curve (transformation) is concave
    This is used to remove right skewness from the data
- How do we actually select the value of λ?
  - Two methods
    Try different values
    - Graph the histograms and select one of them
    - Use the value that shows the most insight

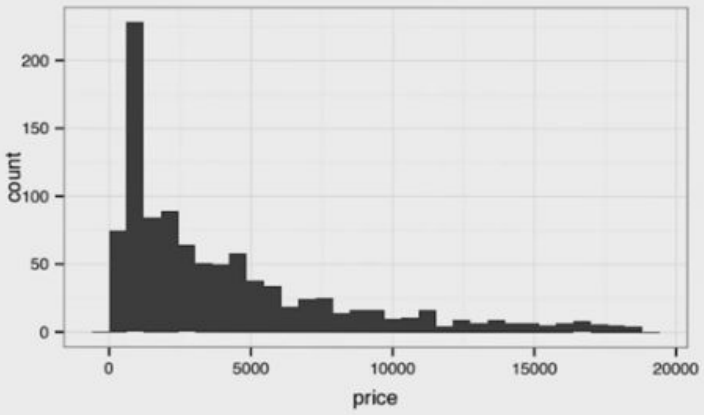
    Use a method based on a maximum likelihood estimator
    - This is more principled method that uses statistics
    - The maximum likelihood estimator finds / fits the best λ to the data
  - In practice….most people just try different values, graph the histograms, and use the

best value they find

Example: Power Transformation on the price of Diamonds

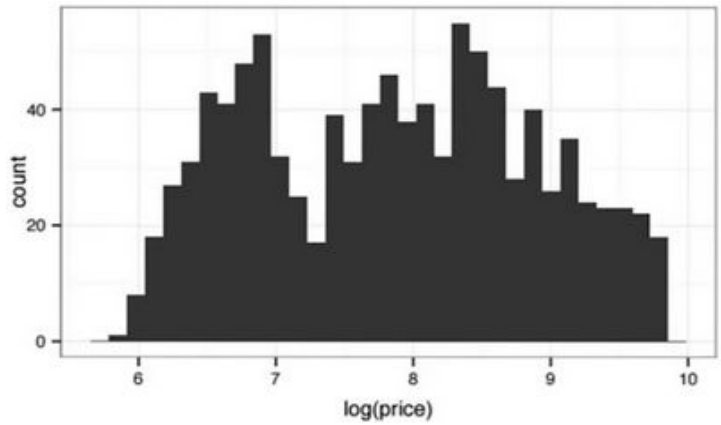
- The price of diamonds in the diamonds dataset is not Gaussian
  - It peaks early and then has a long right tail

```
diamondsSubset = diamonds[sample(dim(diamonds)[1], 1000),]
qplot(price, data = diamondsSubset)
```



- A power transformation – in the example, it's the log(price) – can be used to get a better visualization
  - NOTE THIS IS A SUBSET – ALL EXAMPLES THAT USE DIAMONDS USE A SUBSET

```
qplot(log(price), size = I(1), data = diamondsSubset)
```



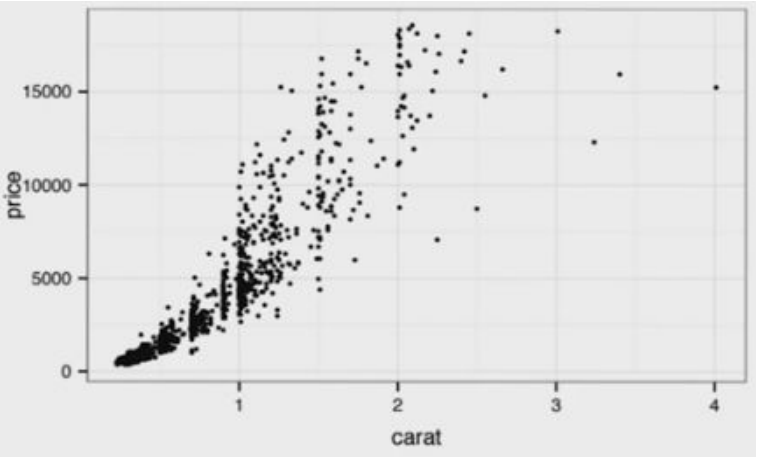
  - We see a bi-modal relationship that was not visible on the original scale
  - As the log of the price increases we see an increase in the count of diamonds, a decrease, and then another increase

    This was not visible on the original scale

Power Transformation

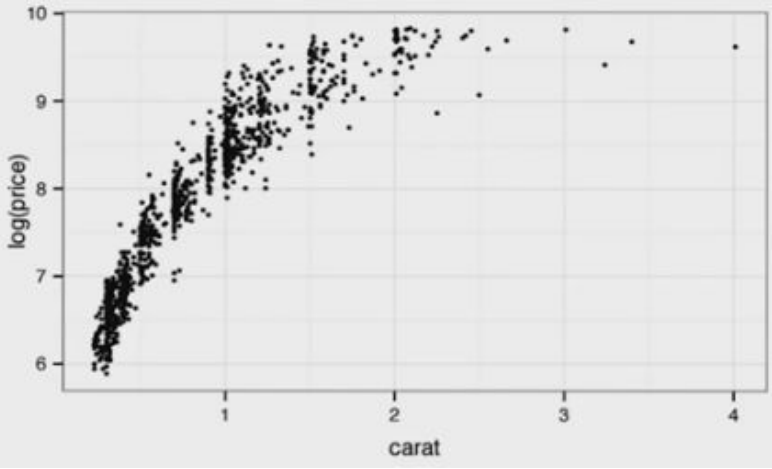
- Not only for histograms, power transformations can be used to examine the relationship between two or more data types
- Scatterplot example

```
qplot(carat,
      price,
      size = I(1),
      data = diamondsSubset)
```



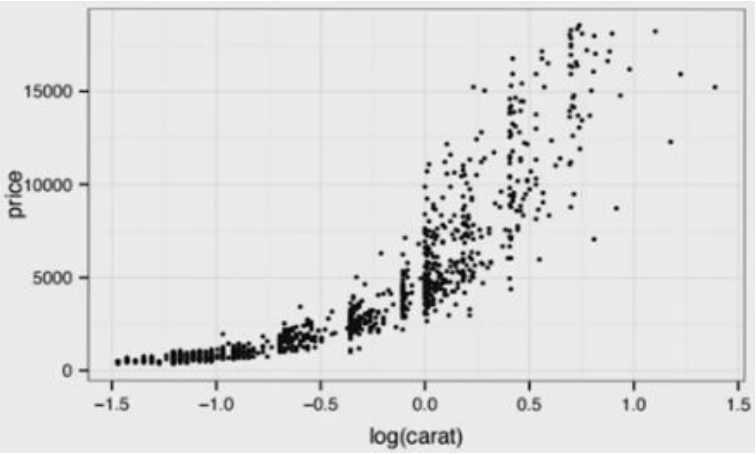
- The above is a sample from the diamonds dataset that shows a clear correlation between price and carat
- We can transform this by using a power transformation on price, carat, or both!
- This is a power transformation on price (using log):

```
qplot(carat,
      log(price),
      size = I(1),
      data = diamondsSubset)
```



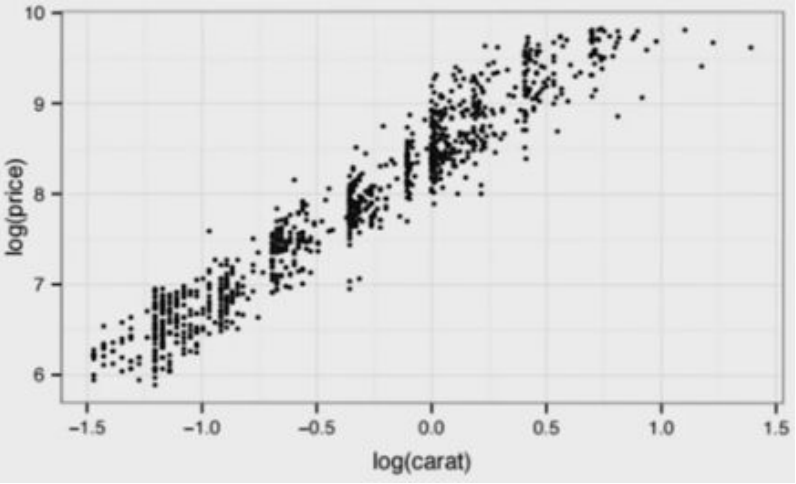
- This is a power transformation on carat (log):

```
qplot(log(carat),
      price,
      size = I(1),
      data = diamondsSubset)
```



- This is a power transformation on both price and carat:

```
qplot(log(carat),
      log(price),
      size = I(1),
      data = diamondsSubset)
```



This is also known as a **log log plot**

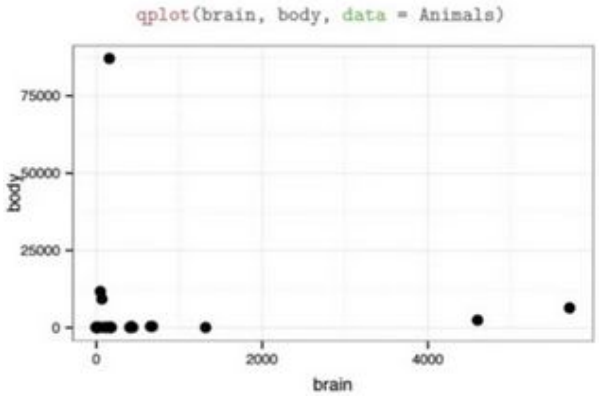
This shows (in the log-log scale) a linear relationship between carat and price that we could not see before in the data

Why is this useful?

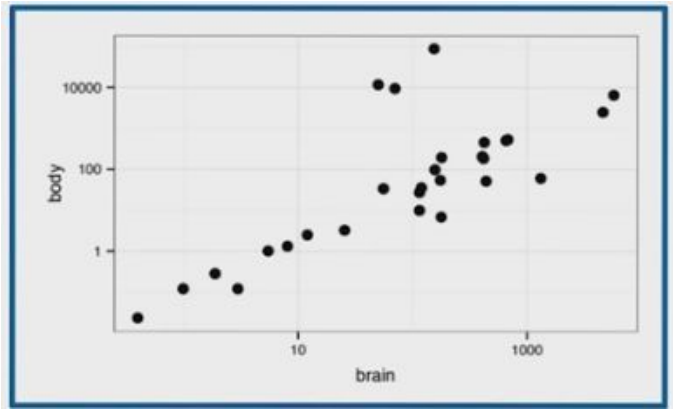
- Its insightful to know there is a simple relationship between these two variables
- If we want to build a statistical model to predict the price on the carat (a regression model), it makes more sense to use linear regression to do the regression model on the log of the carat (rather than the carat) and interpret the result as the log of the price (instead of the price)
- If we want to predict price, we can use the exponential transformation to invert the log of the price transformation

Power Quiz

- Problem: Using the MASS library / Animals dataset, change the qplot command to show the relationship between the body and brain mass on a log log scale
- Given: the body vs brain scatterplot below (note it only uses the first 12 rows)



- Answer
  - One way to do this is via
    qplot(brain,body,log="xy",data=Animals)
    This uses the 'log' argument instead of how it was done in the example above this one
  - Actual log log graph



Again, this is the log log scale
Using the power transformation, we see a nice and clear linear relationship between the body weight and the brain weight
We can also see 3 animals that deviate significantly

- These actually correlate to dinosaurs – why are they different
  - It could be that in prehistoric times, the body/brain ratio was different
  - It could also be that these data points are simply incorrect (no one has ever directly measured a dinosaur brain, let alone their weight)

Data Transformations: Binning

Numeric variables represent real valued measurements whose values are ordered in a manner consistent with the natural ordering of the real line

- The similarity between two measurements (A and B) can be described by the Euclidean distance: the absolute value of B – A
- Examples: height, weight

Ordinal variables represent measurements in a certain range R (R=range) for which we have a well defined order relation

- Numeric variables are a special case of ordinal values
  - All numeric variables are ordinal values, but not vice versa
- Seasons of the year are ordinal measurements
  - There is a clear order
  - They are not numeric – its absurd to subtract winter from spring

Categorical variables represent measurements that do not satisfy the ordinal or numeric assumption

- Example: food items

Binning (also known as discretization) is a data transformation that takes a numeric variable (typically a real value, though it may be an integer), dividing its range into several bins, and replacing it with a number representing the corresponding bin

- **Binarization** is a special case of binning; it replaces a variable with either 0 or 1 depending on whether the variable is greater or smaller than a certain threshold.
- Discretization can be done in the R language by using the function ‘cut’

Why would we want to bin values

- Data reduction
  - Storing bin values and processing them may be much faster than storing the original values while taking less space
  - It can improve scalability of data analysis procedures that handle big data
- It can help capture nonlinear effects when using linear models
  - A common technique is to bin a variable and then different nonlinear transformations are operated on the bin values with all values being concatenated into a vector of

measurements that are then inserted into a linear model
This can capture nonlinear relationships between the data using linear modelling tools

Binning Example

- Suppose x represents the tenure of an employee (in years) and ranges from 0 to 50
  - Binning would divide the range into (0,10], (11,20]…(41,50] and pick a representative number for each bin (for example middle point)
    Recall that, for example, (0,10] means “greater than 0 and less than or equal to 10”

Data Transformations: Indicator Variables

This is a transformation related to binning

Indicator variables replace a variable x (which may be numeric, ordinal, or categorical) taking k values with a binary k-dimensional vector v, such that v[i] (or v_i in mathematical notation) is 1 if and only if x takes on the i-value in its range

- In other words, replace variable (that may take multiple values) by a vector that is all zeros, except for one component that equals 1 (corresponding to the index representing the original value of the variable).

Often, indicator variables are used in conjunction with binning

- We first bin the variable into k bins and then create a k dimensional indicator variable

Indicator variables may be high dimensional

- **High dimensional indicator variables** may be easily handled in computations by taking advantage of its extreme sparsity
  - High dimensional means the original variable has many possible values
  - High dimensionality of indicator vectors is not a problem
    It can be easily handled on storage and computation by taking advantage of its extreme sparsity
    - After all, the vectors are almost all 0s
    - Storing can be done efficiently and processing done quickly, even if there are tens of thousands of dimensions

Models for numeric or binary data cannot directly model ordinal or categorical data

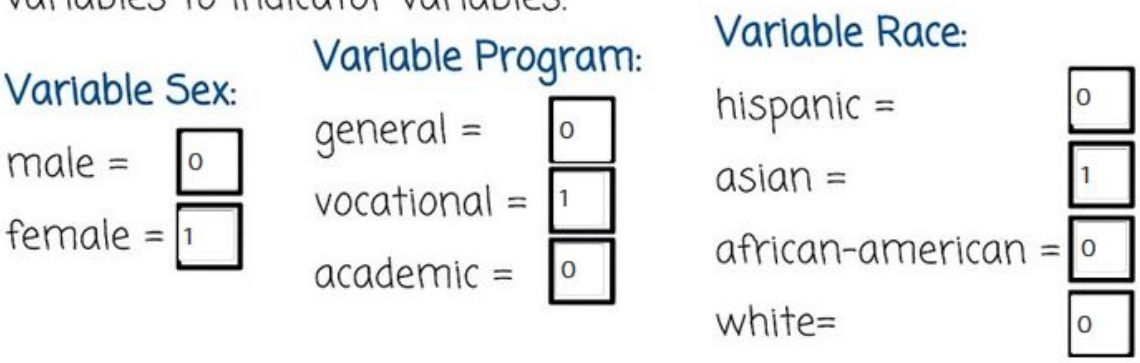
- An important use case for indicator variables is when we want to sue a standard statistical model (linear regression or logistic regression) but our data is ordinal or categorical
- We can convert the variables into an indicator variable; we then concatenate all indicator vectors into a single vector, feeding this binary vector into the standard statistical model (aka linear regression)
- Computation would be fast if we take advantage of the data being comprised of mostly 0s
- The model will be able to find interesting relationships in the data
- <u>This is mandatory for converting ordinal or categorical data into standard statistical models</u>

Summary

- We can transform the data using several non-linear transformations (aka power transformations), then bin the transformed data and create indicator vectors
- It is often easier to compute with indicator functions since they are binary, and thus replacing numeric variables with indicator vectors may improve scalability
  - We can also transform existing ordinal/categorical variables into indicator vectors and concatenate them to one big vector
    This vector can then be used in a standard statistical model
- These methods are popular as its scalable, is effective, and is an easy way to handle both ordinal and categorical variables that are usually difficult to handle using standard models
- It can help capture nonlinear effects using linear models because we used nonlinear transformations on the numeric variables

Indicator Variables Quiz

A study on students and standardized test scores collected the following information: female, vocational, asian. Translate the variables to indicator variables.



Data Manipulations: Shuffling

A common operation in data analysis is to select a random subset of the rows of a dataframe, with or without replacement

The R function sample() accepts a vector of values from which to sample (typically a vector of row indices), the number of samples, whether the sampling is done with or without replacement, and the probability of sampling different values

- Example
  - Sample(k,k) generates a random permutation of order k
  - Example in R

```
#create an array of 4 rows and 5 columns and insert the numbers 1-20
D <- array(data=seq(1,20,length.out = 20),dim=c(4,5))
#use sample() to re-order the dataframe
#the first 4 represents the numbers to use, and the second 4 indicates the number of times to draw a sample
#since we need without replacement, both numbers must be the same, which is the row count!
#note that it defaults to without replacement; to use with replacement use 'replace=TRUE'
D_Shuffled = D[sample(4,4),]
```

  - Since we are using without replacement, the code simply shuffles the rows
  - This is a common operation as we frequently want to take the data, randomly shuffle, then create a training and testing set
    - In other words, this is helpful for cross validation
    - This is important because earlier measurements in the data may be different than later measurements

Data Manipulation: Partitioning

In some cases we need to partition the dataset's rows into two or more collection of rows

- For example, a training set and a testing set

Generate a random permutation of k objects (using sample(k,k)), where k is the number of rows in the data, and then divide the permutation vector into two or more parts based on the prescribed sizes, and new dataframes whose rows correspond to the divided permutation vector

Splitting data into a training and testing set in R

- Uses the above example code to shuffle
- Code

```
#create an array of 4 rows and 5 columns and insert the numbers 1-20
DataSet <- array(data=seq(1,20,length.out = 20),dim=c(4,5))
randomPermutation <- sample(4,4)
trainingIndicies <- randomPermutation[1:floor(4*0.75)]
testingIndicies <- randomPermutation[(floor(4*0.75)+1):4]
trainingData <- DataSet[trainingIndicies,]
testingData <- DataSet[testingIndicies,]
```

Tall Data

Data in **tall format** (**tall data**) is an array or data frame containing multiple columns where one or more columns act as a unique identifier and an additional column (or several columns) represents value

Example

Date	Item	Quantity
2015/01/01	apples	200
2015/01/01	oranges	150
2015/01/02	apples	220
2015/01/02	oranges	130

- The unique identifier is a combination of date AND item (as both are needed to identify a unique row)
- The quantity is the value
- This is exactly like a SQL table

Tall data is convenient for adding new records incrementally and for removing old records

- Its simply a matter of adding more rows or removing specific rows

The tall data format makes it more difficult to conduct analysis or summarizing

- You would need to cycle through ALL rows

Tall data is useful if you need to quickly add / remove entries

Wide Data

Wide data represents in multiple columns the information that tall data holds in multiple rows

Advantage

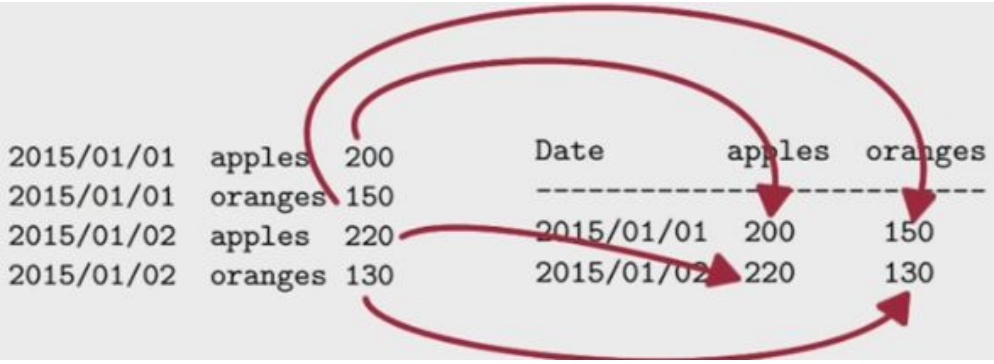
- Wide data is simpler to analyze
  - You don't need to do a full pass over the data / cycle through all rows

Disadvantage

- Harder to add/remove entries

Wide data is best when all data is collected and we want to analyze it

Example



- Same data as in the tall data example, but much easier to analyze
- Each row corresponds to one day (easier to graph!)
- Apples / Oranges split making it easier

When converting tall data to wide data, we need to specify ID variables that define the row and column structure (date and item in the example above)

Reshaping Data

The R package 'reshape2' converts data between tall and wide formats

- The '**melt**' function accepts a frame in the wide format, and the indices of the columns that act as unique identifiers with the remaining columns acting as measurement or values
  - A tall version of the data is returned
- Example: smiths dataset in R

```
##      subject time age weight height
## 1 John Smith    1  33     90   1.87
## 2 Mary Smith    1  NA     NA   1.54

  smiths_tall = melt(smiths, id = 1)
  print(smiths_tall[1:4,])

  ##      subject variable value
  ## 1 John Smith     time     1
  ## 2 Mary Smith     time     1
  ## 3 John Smith      age    33
  ## 4 Mary Smith      age    NA
```

  - Note the code
    smiths_tail <- melt(smiths,id=1)
  - This simply tells the melt() function to use the smiths dataset and use the first column as the unique ID
  - Note it uses the generic variable and value for all other columns (but subject remains the same)

acast/decast is the inverse of melt

- That is to say, it converts tall data to wide
  - acast returns an array
  - dcast returns a dataframe
- The arguments are a dataframe in wide form, a formula a~b~c… where each of a, b, …. Represents a sum of variables whose values will be displayed along the dimensions of the returned array or data frame (a for rows, b for columns, etc), and a function fun.aggregate that aggregates multiple values into a single value
  - This is needed in case multiple values that are mapped to a specific value or cell in the new array/dataframe
- Example

- This uses the ‘tips’ dataset
  The tips dataset has columns: total_bill, tip, sex, smoker, day(of week), time (lunch/dinner), size (of party)
- In previous lessons we used facets to break this out, but we can also do this in code
- We want to transform the data into a series of small tables that will show different insights
- Code

```
library(reshape2)
data(tips)
#this is only needed to get the data in tall format
tipsm<- melt(tips, id = c("sex","smoker","day","time","size"))
#get the mean of measurement variables broken down by sex
dcast(tipsm, sex~variable, fun.aggregate = mean)
```

- This is the result (note the heads are used to show you what the data looks like and is not in the above code)

```
> head(tips)
  total_bill  tip    sex smoker day   time size
1      16.99 1.01 Female     No Sun Dinner    2
2      10.34 1.66   Male     No Sun Dinner    3
3      21.01 3.50   Male     No Sun Dinner    3
4      23.68 3.31   Male     No Sun Dinner    2
5      24.59 3.61 Female     No Sun Dinner    4
6      25.29 4.71   Male     No Sun Dinner    4
> head(tipsm)
     sex smoker day   time size   variable value
1 Female     No Sun Dinner    2 total_bill 16.99
2   Male     No Sun Dinner    3 total_bill 10.34
3   Male     No Sun Dinner    3 total_bill 21.01
4   Male     No Sun Dinner    2 total_bill 23.68
5 Female     No Sun Dinner    4 total_bill 24.59
6   Male     No Sun Dinner    4 total_bill 25.29
> dcast(tipsm, sex~variable, fun.aggregate = mean)
     sex total_bill      tip
1 Female   18.05690 2.833448
2   Male   20.74408 3.089618
```

  Whats cool is that we didn’t have to go over the data in a FOR loop to generate this
- Now get the length (that is to say, the number of measurements)

```
> dcast(tipsm, sex~variable, fun.aggregate = length)
     sex total_bill tip
1 Female         87  87
2   Male        157 157
>
```

  There were 87 instances of females paying and 157 of males paying
- Now we would like to get a combination of sex AND time – here is how that is done (in this case just show length or number of measurements)

```
> dcast(tipsm, sex+time~variable, fun.aggregate = length)
     sex   time total_bill tip
1 Female Dinner         52  52
2 Female  Lunch         35  35
3   Male Dinner        124 124
4   Male  Lunch         33  33
```

- Now we want to get the mean for sex+time, but with a twist: include the aggregate for each bucket (so female/all, male/all, and all/all. To do this we use the ‘margins=TRUE’ parameter:

```
> dcast(tipsm, sex+time~variable, fun.aggregate = mean, margins=TRUE)
     sex   time total_bill      tip    (all)
1 Female Dinner   19.21308 3.002115 11.10760
2 Female  Lunch   16.33914 2.582857  9.46100
3 Female  (all)   18.05690 2.833448 10.44517
4   Male Dinner   21.46145 3.144839 12.30315
5   Male  Lunch   18.04848 2.882121 10.46530
6   Male  (all)   20.74408 3.089618 11.91685
7  (all)  (all)   19.78594 2.998279 11.39211
```

Sometimes its best to look at the straight number over the graph as the number itself offers a clearer picture

Split-Apply-Combine

Data analysis on data frames can be decomposed into three stages

- **Split** - Splitting the dataframe along some dimensions to form smaller arrays or dataframes
- **Apply** - Applying some operation to each of the smaller arrays or dataframes
- **Combine** - Combining the results of the application stage (previous step) into a single meaningful array or dataframe

Performing these three stages again and again on the same dataset (or even on different data sets) can cause problems

Its useful to automate these stages

- This way the programmer will not introduce error
- Will shorten this process to leave more time for the analysis

The R package ‘plyr’ automates these stages

Basic functions of plyr for various input and output types (see below)

input \ output	array	dataframe	list	discarded
array	aaply	adply	alply	a_ply
dataframe	daply	ddply	dlply	d_ply
list	laply	ldply	llply	l_ply

- Note that the functions typically end in ‘ply’ and the first two letters correspond to the input / output data
- Arguments: data, dimensions/columns used to split the data, function to execute in the apply stage

Example of ‘plyr’ application

```
library(plyr)
names(baseball)
##  [1] "id"    "year"  "stint" "team" "lg"  "g"    "ab"
##  [9] "h"     "X2b"   "X3b"   "hr"   "rbi" "sb"   "cs"
## [17] "so"    "ibb"   "hbp"   "sh"   "sf"  "gidp"
# count number of players recorded for each year
bbPerYear = ddply(baseball, "year", "nrow")
head(bbPerYear)
##   year nrow
## 1 1871    7
## 2 1872   13
## 3 1873   13
## 4 1874   15
## 5 1875   17
## 6 1876   15
```

- Note the placement of the variables
  - specifically the function ‘nrow’ (in quotes)
- The input/output are both dataframes (hence the ddply)
- We split across the year

Another example: further automation

```
library(plyr)
data(baseball)
myYear<-names(baseball)[2]#this represents the position that holds the word 'year'
myFunc<-"nrow"
head(ddply(baseball,myYear,myFunc))
```

- Notice that you can pull out the column names for further automation!!!!!!

More examples

```
bbMod <- ddply(baseball, "year", summarize, mean.rbi = mean(rbi, na.rm=TRUE),
length=length(!is.na(rbi)), sd = sd(rbi, na.rm=TRUE))
```

- the function we use for this one is 'summarize, which takes as an argument the function that computes the RBI mean
- it seems summarize is for specific columns, wheras the last example (nrow) was for the entire dataframe
- this seems to be the difference between using summarize and the last usage (nrows)
- this gets some basic things like length, mean, and sd

```
bbmod2 <-ddply(baseball, "year", function(x) c(mean.rbi = mean(x$rbi, na.rm=TRUE)))
```

- it can also be done without summarize, if you make an in-line function

```
myRBI_MeanFunction <- function(x) {
  return(mean(x$rbi, na.rm=TRUE))
}
bbmod3 <-ddply(baseball, "year", myRBI_MeanFunction)
```

- finally, you can make your OWN function as well! note that its assumed a dataframe will be sent as x

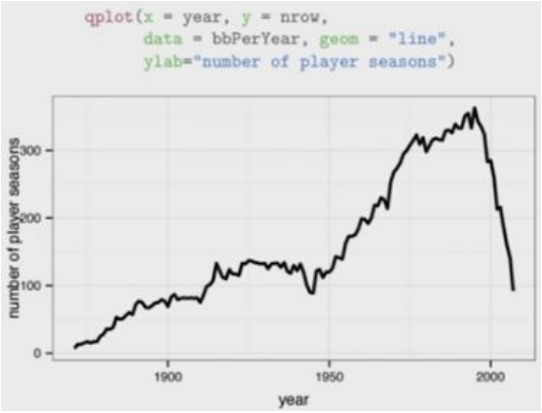
```
baseballTransformed <- ddply(baseball, "id", transform,career.year = year - min(year) + 1)
```

- Transform simply appends a column to the end of the dataframe - in this case, career.year
- you can also do inner calculations (much like GROUP BY in SQL) with transform
- this figures out each players years playing baseball each year (so if a player played from 1912-1917, 1912=1, 1913=2...1917=5)
- (recall that a player is listed each year)

<u>Plyr is one way to get around using FOR loops and can act as a GROUP BY a la SQL</u>

Baseball example

- Graph



- Looking at this is important because we see that the years after 2000 may be incomplete

Ozone Example

Ozone is a 3 dimensional dataset in plyr

- Specifically, it's a 3 dimensional array of ozone measurements varying by latitude, longitude, and time

This trended ozone across longitude, latitude, and time.

It showed that ozone has a periodicity trend (that is to say, it has cycle periods, which in this case relate to the seasons of the year)

Lesson Summary

We learned what to do when there is missing data and how to handle outliers

Transforming the data can reveal insights (that were previously hard to recognize) and can improve modeling

Standard data manipulation by tools in the R language like reshape and plyr