Practical Machine Learning Notes

Coursera Course by John Hopkins University

INSTRUCTORS: Dr. Jeff Leek, Dr. Brian Caffo, Dr. Roger D. Peng

Contents

Intro

- This course covers the basic ideas behind machine learning/prediction
  - Study Design - training vs. test sets
  - Conceptual issues - out of sample error, overfitting, ROC curves
  - Practical Implementation - the caret package
- What this course depends on:
  - The Data Scientist's Toolbox
  - R Programming
- What would be useful
  - Exploratory Analysis
  - Reproducible Research
  - Regression Models
  - (Notes on these 5 courses are all in my GitHub repoes)

GitHub Link for Lectures

Practical Machine Learning lectures on GitHub

Course Book

The book for this course is available on this site

Instructor's Note

"*Welcome to Practical Machine Learning! This course will focus on developing the tools and techniques for understanding, building, and testing prediction functions.*

These tools are at the center of the Data Science revolution. Many researchers, companies, and governmental organizations would like to use the cheap and abundant data they are collecting to predict what customers will like, what services to offer, or how to improve people's lives.

Jeff Leek and the Data Science Track Team"

Prediction, Errors, and Cross Validation

Prediction

Prediction Motivation

- Who predicts things?
  - Local governments -> pension payments
  - Google -> whether you will click on an ad
  - Amazon -> what movies you will watch
  - Insurance companies -> what your risk of death is
  - Johns Hopkins -> who will succeed in their programs
- Why predict things
  - Glory (Nerd cred for accomplishing certain feats)
    - A lot of competitions are hosted on **Kaggle**
  - Riches (Completing some competition that offers a reward)
  - Save lives
    - **On cotype DX** reveals the underlying biology that changes treatment decisions 37% of the time.

More Resources

- **A course on more advanced material about ML**
- **List of machine learning resources on Quora**
- **List of machine learning resources from Science**

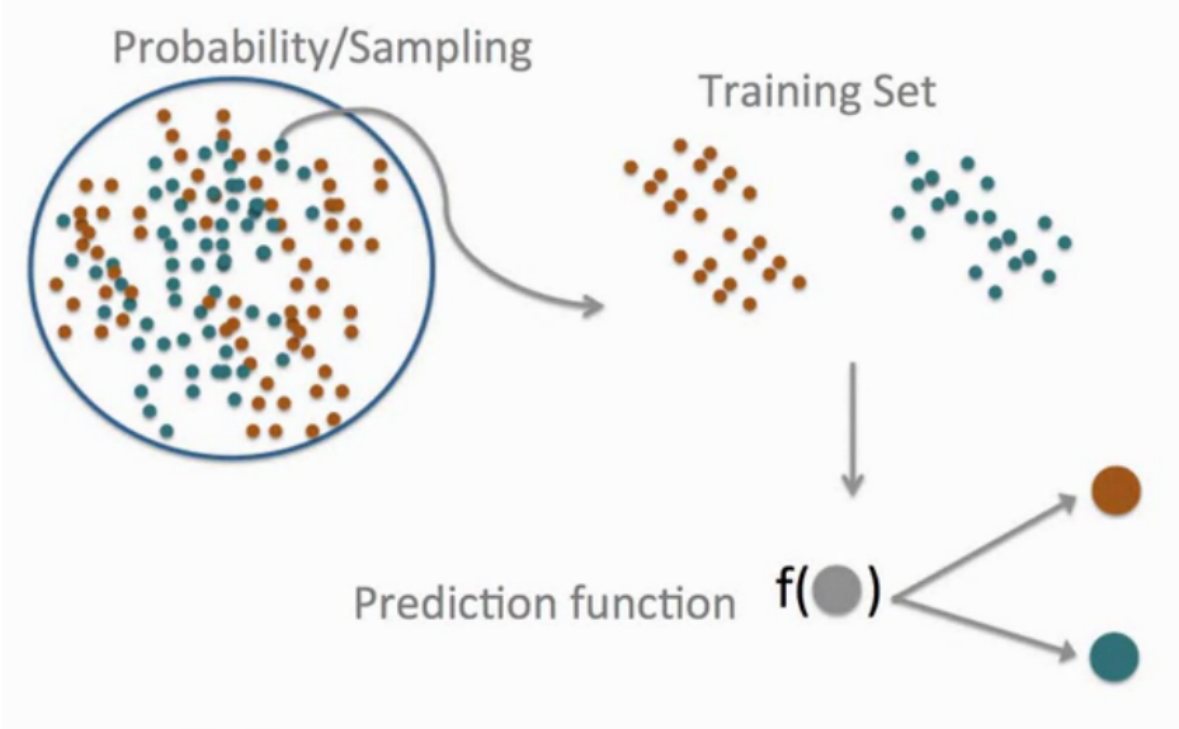
- **Advanced notes from MIT open courseware**
- **Advanced notes from CMU**
- **Kaggle - machine learning competitions**

What is Prediction?

Main Idea

- One focus of ML is on what algorithms are the best for extracting information and using it to predict.

- Although the method used for producing a training set is also quite important



- One starts off with a dataset

1) One uses Probability/Sampling to select a Training Set
2) One measures characteristics of this training set to create a Predicition Function
3) One then uses the Prediction Function to take an uncolored dot and predict if it's red or blue
4) One would then go on to test how well their Prediction Function works

What Can Go Wrong

- An example is **Google Flu trends** (**A free overview of the issue witht he accuracy**)
  - Google tried to predict rate of flu using what people would search
  - Originally the algorithm was able to represent how many cases would appear in a region within a certain time
  - Although they didn't account for the fact that the terms would change over time
  - The way the terms were being used wasn't well understood so when terms changed they weren't able to accurately account for the change.
  - It also overestimated as it the search terms it looked at were often cofactors with other illnesses

Componets of a Predictor

1) Question

- Any problem in data science starts with a question, "What are you trying to predict and what are you trying to predict it with?"

2) Input Data

- Collect best input data you can to use to predict

3) Features

- From that data one builds features that they will use to predict

4) Algorithm

- One uses ML algorithms to develop a function

5) Parameters

- Estimate parameters of the algorithm

6) Evaluation

- Apply algorithm to a data set to evaluate how well the algorithm works

Example

- Start with a general question, "Can I automatically detect emails that are SPAM and those that are not?"

- Make the question more concrete, "Can I use quantitative characteristics of the emails to classify them as SPAM/HAM?"

- Find input data
  - In this instance there is data avaliable in R via the `kernlab` package

  - Note that this data set won't necessarily be the perfect data as it doesn't contain all the emails ever sent, or the emails sent to you personally

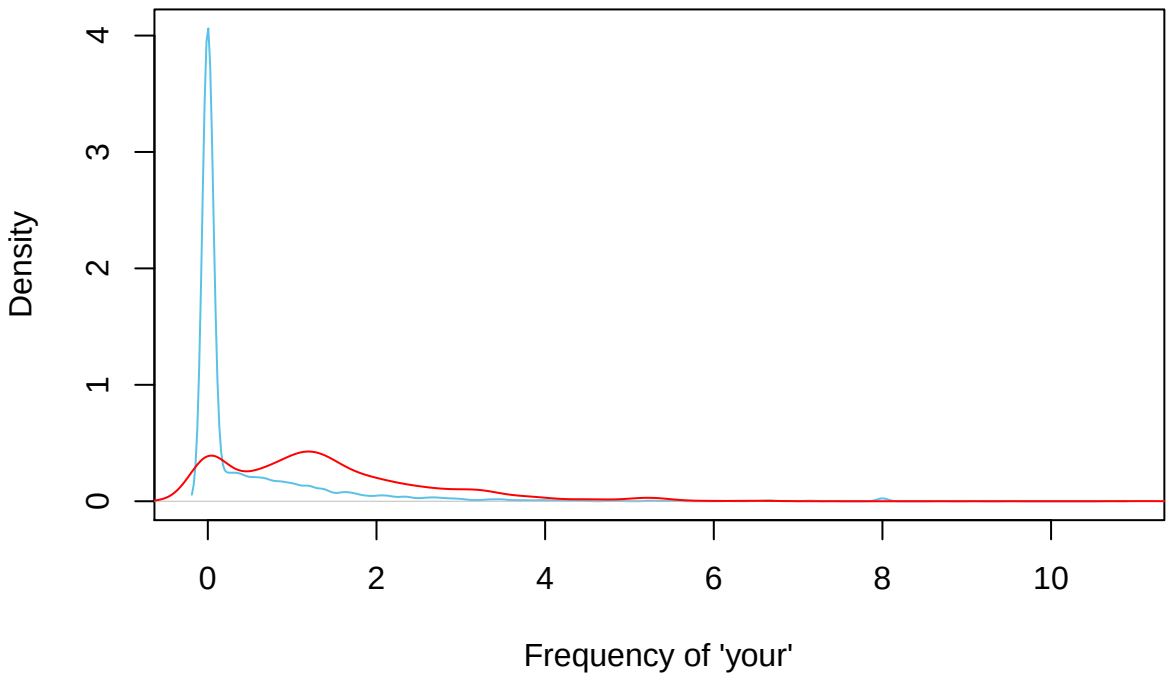
- Quantify features, such as the frequency of certain words or typeface. The `spam` dataset from `kernlab` contains these types of frequency.

```
library(kernlab)
data(spam)
str(spam)
```

```
## 'data.frame':    4601 obs. of  58 variables:
##  $ make             : num  0 0.21 0.06 0 0 0 0 0 0.15 0.06 ...
##  $ address          : num  0.64 0.28 0 0 0 0 0 0 0 0.12 ...
##  $ all              : num  0.64 0.5 0.71 0 0 0 0 0 0.46 0.77 ...
##  $ num3d            : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ our              : num  0.32 0.14 1.23 0.63 0.63 1.85 1.92 1.88 0.61 0.19 ...
##  $ over             : num  0 0.28 0.19 0 0 0 0 0 0 0.32 ...
##  $ remove           : num  0 0.21 0.19 0.31 0.31 0 0 0 0.3 0.38 ...
##  $ internet         : num  0 0.07 0.12 0.63 0.63 1.85 0 1.88 0 0 ...
##  $ order            : num  0 0 0.64 0.31 0.31 0 0 0 0.92 0.06 ...
##  $ mail             : num  0 0.94 0.25 0.63 0.63 0 0.64 0 0.76 0 ...
##  $ receive          : num  0 0.21 0.38 0.31 0.31 0 0.96 0 0.76 0 ...
##  $ will             : num  0.64 0.79 0.45 0.31 0.31 0 1.28 0 0.92 0.64 ...
##  $ people           : num  0 0.65 0.12 0.31 0.31 0 0 0 0 0.25 ...
##  $ report           : num  0 0.21 0 0 0 0 0 0 0 0 ...
##  $ addresses        : num  0 0.14 1.75 0 0 0 0 0 0 0.12 ...
##  $ free             : num  0.32 0.14 0.06 0.31 0.31 0 0.96 0 0 0 ...
##  $ business         : num  0 0.07 0.06 0 0 0 0 0 0 0 ...
##  $ email            : num  1.29 0.28 1.03 0 0 0 0.32 0 0.15 0.12 ...
##  $ you              : num  1.93 3.47 1.36 3.18 3.18 0 3.85 0 1.23 1.67 ...
##  $ credit           : num  0 0 0.32 0 0 0 0 0 3.53 0.06 ...
##  $ your             : num  0.96 1.59 0.51 0.31 0.31 0 0.64 0 2 0.71 ...
##  $ font             : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ num000           : num  0 0.43 1.16 0 0 0 0 0 0 0.19 ...
##  $ money            : num  0 0.43 0.06 0 0 0 0 0 0.15 0 ...
##  $ hp               : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hpl              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ george           : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ num650           : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ lab              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ labs             : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ telnet           : num  0 0 0 0 0 0 0 0 0 0 ...
```

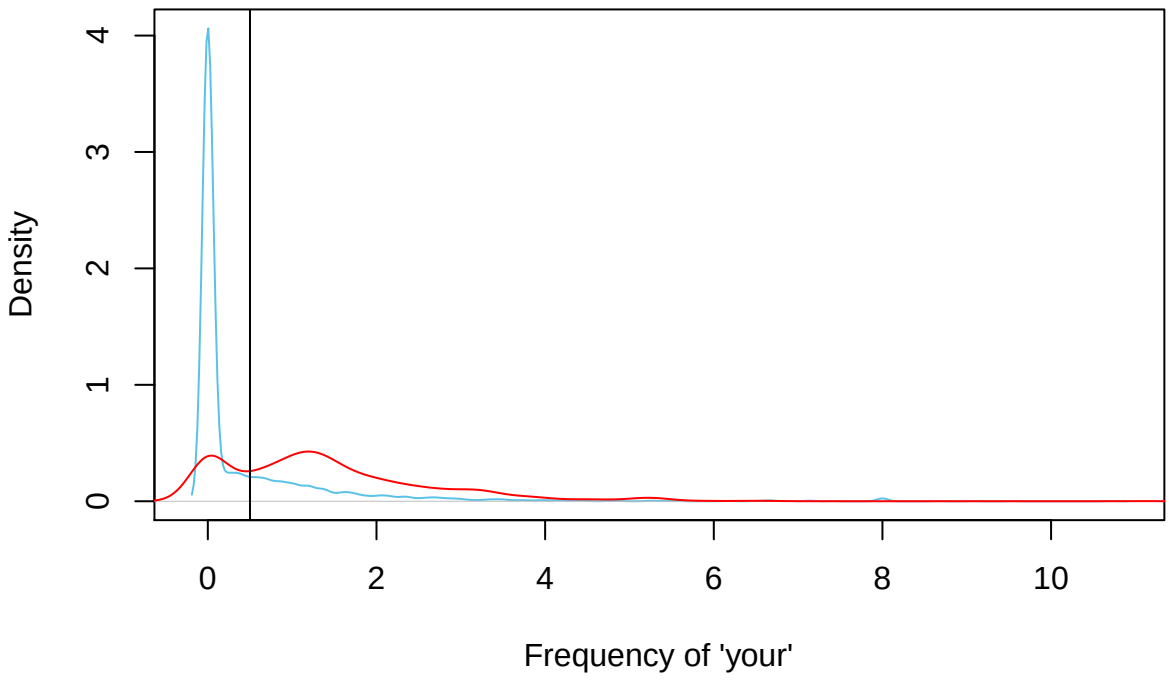
```
##  $ num857           : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ data             : num  0 0 0 0 0 0 0 0 0.15 0 ...
##  $ num415           : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ num85            : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ technology       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ num1999          : num  0 0.07 0 0 0 0 0 0 0 0 ...
##  $ parts            : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ pm               : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ direct           : num  0 0 0.06 0 0 0 0 0 0 0 ...
##  $ cs               : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ meeting          : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ original         : num  0 0 0.12 0 0 0 0 0 0.3 0 ...
##  $ project          : num  0 0 0 0 0 0 0 0 0 0.06 ...
##  $ re               : num  0 0 0.06 0 0 0 0 0 0 0 ...
##  $ edu              : num  0 0 0.06 0 0 0 0 0 0 0 ...
##  $ table            : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ conference       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ charSemicolon    : num  0 0 0.01 0 0 0 0 0 0 0.04 ...
##  $ charRoundbracket : num  0 0.132 0.143 0.137 0.135 0.223 0.054 0.206 0.271 0.03 ...
##  $ charSquarebracket: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ charExclamation  : num  0.778 0.372 0.276 0.137 0.135 0 0.164 0 0.181 0.244 ...
##  $ charDollar       : num  0 0.18 0.184 0 0 0 0.054 0 0.203 0.081 ...
##  $ charHash         : num  0 0.048 0.01 0 0 0 0 0 0.022 0 ...
##  $ capitalAve       : num  3.76 5.11 9.82 3.54 3.54 ...
##  $ capitalLong      : num  61 101 485 40 40 15 4 11 445 43 ...
##  $ capitalTotal     : num  278 1028 2259 191 191 ...
##  $ type             : Factor w/ 2 levels "nonspam","spam": 2 2 2 2 2 2 2 2 2 2 ...
```

```
plot(density(spam$your[spam$type=="nonspam"]),
     col = "#5BC2E7", main = "", xlab = "Frequency of 'your'")
lines(density(spam$your[spam$type=="spam"]), col = "#FF0000")
```



- It can be seen here "your" appears more often in SPAM emails than it does in HAM
- One could use this idea to create a cut-off point for predicting if a message is SPAM
- The proposed algorithm
  - Find a value of C
  - If the frequency of $'your' > C$ predict the message is SPAM

```
plot(density(spam$your[spam$type=="nonspam"]),
     col = "#5BC2E7", main = "", xlab = "Frequency of 'your'")
lines(density(spam$your[spam$type=="spam"]), col = "#FF0000")
abline(v = 0.5, col = "#000000")
```



- Choosing 0.5 would contain most spam messages and avoid the second spike of HAM emails

- We then evaluate this predictor

```
prediction <- ifelse(spam$your > 0.5, "spam", "nonspam")
res <- table(prediction, spam$type)/length(spam$type)
res
```

```
##
## prediction   nonspam      spam
##    nonspam 0.4590306 0.1017170
##    spam    0.1469246 0.2923278
```

- In this case our accuracy is $0.459 + 0.2923 = 0.7514$, or an accuracy of approximately 75.14%, although this is an opptamistic measure of the overall error, which will be discussed further later.

Relative Importance of Steps

question > data > features/variables > algorithms
"*The commbinaiton of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.*" -John Tukey

- In other words, an important component of prediction is knowing when to give up, that is

that the data is not sufficient

Input Data: Garbage in = Garbage out

1. May be easy (movie ratings -> new movie ratings)
2. May be harder (gene expression data -> disease)
3. Depends on what is a "good prediction".
4. Often more **data > better models**
5. The most important step is collecting the right data

Features: They matter!

- Properties of good features
  - Lead to data compression
  - Retain relevant information
  - Are created based on expert application knowledge
- Common mistakes
  - Trying to automate feature selection (Although they may be automated with care)
  - Not paying attention to data-specific quirks
  - Throwing away information unnecessarily

Algorithm: They Matter Less Than You'd Think

- The above table shows that the Linear Discrimenate Analysis (Lindisc) error often was not that far off from the best method
- Using the best approach doesn't always largely improve the error

Issues to Consider

- The "Best" machine learning method would be:
  - Interpretable
    - If predictor is to be presented to an uninformed audience you'd want to to be understandable by them
  - Simple
    - Helps with interpretablity

TABLE 1
Performance of linear discriminant analysis and the best result we found on ten randomly selected data sets

Data set	Best method e.r.	Lindisc e.r.	Default rule	Prop linear
Segmentation	0.0140	0.083	0.760	0.907
Pima	0.1979	0.221	0.350	0.848
House-votes16	0.0270	0.046	0.386	0.948
Vehicle	0.1450	0.216	0.750	0.883
Satimage	0.0850	0.160	0.758	0.889
Heart Cleveland	0.1410	0.141	0.560	1.000
Splice	0.0330	0.057	0.475	0.945
Waveform21	0.0035	0.004	0.667	0.999
Led7	0.2650	0.265	0.900	1.000
Breast Wisconsin	0.0260	0.038	0.345	0.963

Figure 1: Linear vs Model

  - Accurate
    - Getting a model to be interpretable can sometimes hurt the accuracy
  - Fast
    - Quick build the model, train, and test
  - Scalable
    - Easy to apply to a large dataset (either fast or parallelizable)

Prediction is About Accuracy Tradeoffs

- Tradeoffs are made for interpretability, speed, simplicity, or scalability.
- Interpretability matters, decision tree-like results are more interpretable
  - "**if** total cholesterol ≥ 160 **and** they smoke **then** *10 year CHD risk* $\geq$ *5%* **else if** they smoke **and** systolic blood pressure ≥ 140 **then** *10 year CHD risk* $\geq 5\%$ **else** *10 year CHD risk* $< 5\%$"
- Scalability matter
  - in "The Netflix $1 Million Challenge" Netflix never implemented the solution itself because the algorithm wasn't scalable and took way too long on the big data sets that Neflix was working with, so they went with something that was less accurate but more scalable.

Errors

In and Out of Sample Errors

- **In Sample Error** - Sometimes called *resubstitution error*. The error rate you get on the same data set you used to build your predictor.
- **Out of Sample Error** - Sometimes called *generalization error*. The error rate you get on a new data set.
- Key Ideas

1) Out of sample error is what you care about
2) In sample error $<$ out of sample error

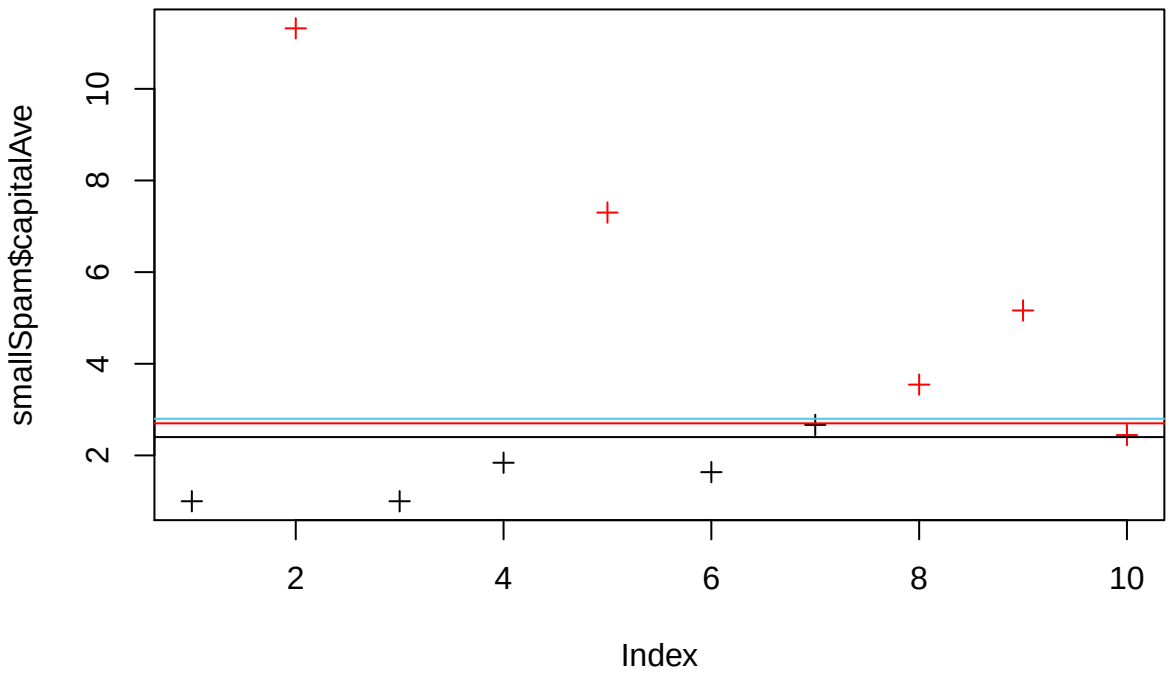
- Sometimes you want to give up some accuracy on the data you have to have greater accuracy on unkown data.

3) The reason is overfitting

- Matching your algorithm to the noise of the data you have

Spam Example

```
library(kernlab)
data(spam)
RNGkind(sample.kind = "Rounding")
set.seed(333)
smallSpam <- spam[sample(dim(spam)[1], size = 10),]
spamLabel <- (smallSpam$type == "spam")*1 + 1
plot(smallSpam$capitalAve, col = spamLabel, pch = 3)
abline(h = 2.7, col = "#FF0000")
abline(h = 2.40, col = "#000000")
abline(h = 2.80, col = "#5BC2E7")
```



Prediction 1

- capitalAve > 2.7 = "spam"

- capitalAve < 2.40 = "nonspam"
- We can add 2 params to make the prediction perfect for the training set
  - capitalAve between 2.40 and 2.45 = "spam"

  - capitalAve between 2.45 and 2.7 = "nonspam"

```
rule1 <- function(x){
  prediction <- rep(NA, length(x))
  prediction[x > 2.7] <- "spam"
  prediction[x < 2.40] <- "nonspam"
  prediction[(x >= 2.40 & x <= 2.45)] <- "spam"
  prediction[(x > 2.45 & x <= 2.70)] <- "nonspam"
  return(prediction)
}
table(rule1(smallSpam$capitalAve),smallSpam$type)
```

```
##
##             nonspam spam
##   nonspam       5    0
```

```
##    spam          0    5
```

Prediction 2

- (Note: The blue line in the plot is for 2.8)
- capitalAve > 2.80 = "spam"
- capitalAve ≤ 2.80 = "nonspam"
- This algorithm won't be perfect on the training data

```
rule2 <- function(x){
  prediction <- rep(NA, length(x))
  prediction[x > 2.8] <- "spam"
  prediction[x <= 2.8] <- "nonspam"
  return(prediction)
}
table(rule2(smallSpam$capitalAve), smallSpam$type)
```

```
##
##           nonspam spam
##   nonspam       5    1
##   spam          0    4
```

Apply 2 rulesets to all spam data

```
table(rule1(spam$capitalAve),spam$type)
```

```
##
##           nonspam spam
##   nonspam    2141  588
##   spam        647 1225
```

```
table(rule2(spam$capitalAve),spam$type)
```

```
##
##           nonspam spam
##   nonspam    2224  642
##   spam        564 1171
```

```
paste0("Rule 1 accuracy: ",
       mean(rule1(spam$capitalAve) == spam$type))
```

```
## [1] "Rule 1 accuracy: 0.731580091284503"
```

```
paste0("Rule 2 accuracy: ",
       mean(rule2(spam$capitalAve) == spam$type))
```

```
## [1] "Rule 2 accuracy: 0.737883068898066"
```

```
paste0("Rule 1 total correct: ",
       sum(rule1(spam$capitalAve) == spam$type))
```

```
## [1] "Rule 1 total correct: 3366"
```

```
paste0("Rule 2 total correct: ",
       sum(rule2(spam$capitalAve) == spam$type))
```

```
## [1] "Rule 2 total correct: 3395"
```

Overfitting

- Why is the ruleset with a *better* out of sample error (`rule2`) the one with a *worse* in sample error?
  - It's because we overfitted `rule1`
  - **Wikipedia on Overfitting**
- All data have two parts
  - Signal - Part we are trying to use to predict
  - Noise - Random variation in data set
- The goal of a predictor is to find the signal and ignore the noise
- You can always design a perfect in-sample predictor
  - Doing this will capture botht he signal and the noise
  - As such a predictor won't perform as well on new samples

Prediction Study Design

1) Define your error rate
2) Split data into:

- Training set to build model
- Testing set to validate model
- Validation set (optional) to also validate the model

3) On the training set pick features (using cross-validation to pick which features are most important in your model)
4) On the training set pick prediction function (also using cross-validation)
5a) If no validation set:

- Apply the best model to the test set exactly 1 time

  - If we apply multiple models to the test set and pick the best one we're kind of using the test set to train the model, giving an optimistic error rate
    5b) If there is a validation set:
- Apply the model the the test set and refine the model

- Then apply best model to validation set once

Benchmarks

- One should know what the prediction benchmarks are for their algorithm to help troubleshoot when something is going wrong. Often a benchmark is something like "all zeros" which tells the error rate if all values were set to 0, pretty much just ignore all the features of the dataset and taking a general average.

Study Design of Netflix Contest

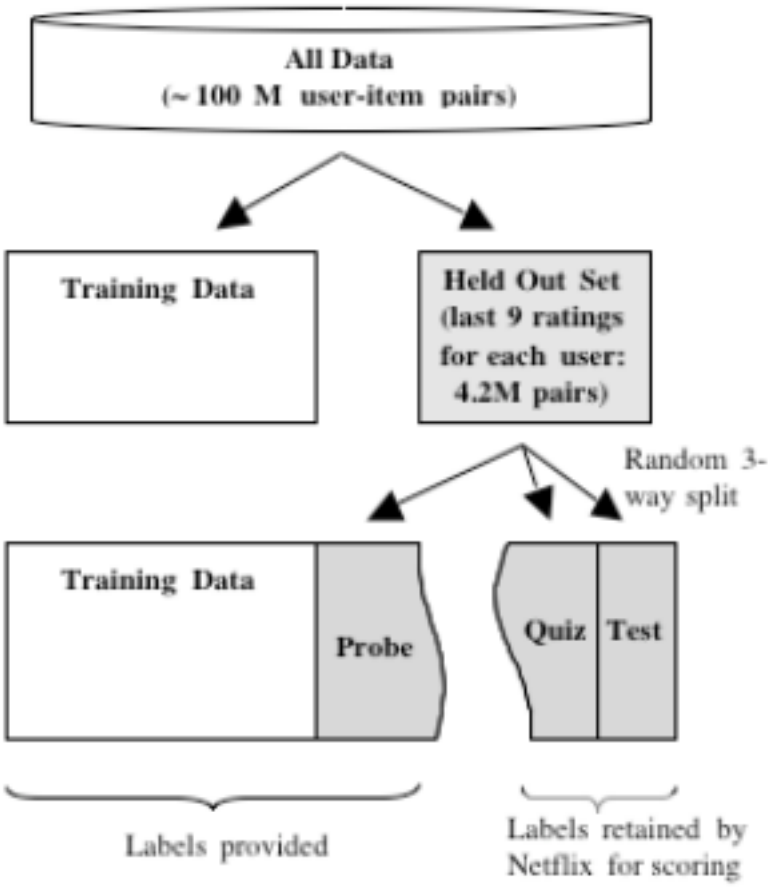


Figure 2: Netflix Design

- Of all the data they split it into a training set (*Training Data*) that they shared with competators and a test & validation set (*Held Out Set*)that they did not share.

- They shared a test set (*Probe*) so competators could test their out of sample error.

- They would then take your model and test it on the *Quiz* set, although one could submit multiple models and get a scoring from the *Quiz* set. So they held out the last bit of data as a validation set (*Test*) that would be used at the end of the competition on each model only once.

Avoid Small Sample Sizes

- Suppose you are predicting a binary outcome
  - Diseased/healthy
  - (Not) Clicking on an ad
- One classifier is flipping a coin
- Probability of perfect classification is approximately $(\frac{1}{2})^{samplesize}$
  - $n = 1$ flipping coin 50% change of 100% accuracy
  - $n = 2$ flipping coin 25% change of 100% accuracy
  - $n = 10$ flipping coin 0.10% change of 100% accuracy
- So lower sample sizes make it harder to know if your high accuracy is from chance or true.

Rules of Thumb for Prediction Study Design

- If you have a large sample size
  - 60% training
  - 20% test
  - 20% validation
- If you have a medium smaple size
  - 60% training
  - 40% testing
- If you have a small sample size
  - Do cross validation
  - Reprot caveat of small sample size

Some Principles to Remember

- Set the test/validation set aside and *don't look at it*
- In general *randomly* sample training and test sets
- Your data sets must reflect structure of the problem
  - If predictions evolve with time split train/test in time chunks (called backtesting in finance)
- All subsets should reflect as much diversity as possible
  - Random assignment does this
  - You can also try to balance by features - but this is tricky

Types of Errors

Basic Terms

In general, **Positive** = identified and **Negative** = rejected. Whereas **True** and **False** indicate correctness. Therefore:
* **True positive** = correctly identified
* **False positive** = incorrectly identified
* **True negative** = correctly rejected
* **False negative** = incorrectly rejected

Medical testing example:
* **True positive** = Sick people correctly diagnosed as sick
* **False positive** = Healthy people incorrectly identified as sick
* **True negative** = Healthy people correctly identified as healthy
* **False negative** = Sick people incorrectly identified as healthy

- Sensitivity and Specificity
  - **Sensitivity** - **True Positve Rate**, *Pr(positive test/disease)*, proportion of actual positives that are correctly identified
  - **Specificity** - **True Negative Rate**, *Pr(negative test/no disease)* proportion of actual negatives that are correctly identified
  - High amount of either ussually entails a high amount of the False ... Rate of the respective type, as ensuring you get all the positive/negative usually means you have to include some of the respective false samples.
  - Sensitivity is likely to diagnois a positive (**Sen**tence the innocent)
  - Specificity is going to be sure the positives are positive, even if they miss some (**Sp**are the innocent)
- Other key quantities
  - **Positive Predictive Value** - *Pr(disease/positive test)*

- **Negative Predictive Value** - *Pr(no disease | negative test)*
- **Accuracy** - *Pr(correct outcome) = Pr(positive test/disease) + Pr(negative test/no disease)*

Key Quantities as Fractions

	Actually_Positive	Actually_Negative
Tested_Postive	TP	FP
Tested_Negative	FN	TN

- Sensitivity = $\frac{TP}{(TP+FN)}$
- Specificity = $\frac{TN}{(FP+TN)}$
- Positive Predictive Value = $\frac{TP}{(TP+FP)}$
- Negative Predictive Value = $\frac{TN}{(FN+TN)}$
- Accuracy = $\frac{(TP+TN)}{(TP+FP+FN+TN)}$

Screening Tests Example

Assume that some disease has a 0.1% prevalence in the population. Assume we have a test kit for that disease that works with 99% sensitivity and 99% specificity. What is the probability of a person having the disease given the test result is positive, if we randomly select a subject from: (We'll look at the expected values if we sampled 100000 people) * The general population?

	Actually_Positive	Actually_Negative
Tested_Postive	99	999
Tested_Negative	1	98901

- Sensitivity = $\frac{99}{(99+1)} = 99\%$
- Specificity = $\frac{98901}{(999+98901)} = 99\%$
- Positive Predictive Value = $\frac{99}{(99+999)} = 9.016\%$
- Negative Predictive Value = $\frac{98901}{(1+98901)} = 99.999\%$
- Accuracy = $\frac{(99+98901)}{100000} = 99\%$
- A high risk sub-population with 10% disease prevalence

	Actually_Positive	Actually_Negative
Tested_Postive	9900	900
Tested_Negative	100	89100

- Sensitivity = $\frac{9900}{(9900+100)} = 99\%$
- Specificity = $\frac{89100}{(900+89100)} = 99\%$
- Positive Predictive Value = $\frac{9900}{(9900+900)} = 91.667\%$
- Negative Predictive Value = $\frac{89100}{(100+89100)} = 99.888\%$
- Accuracy = $\frac{(9900+89100)}{100000} = 99\%$
- This low Postitive Predictive Value shows the issues with predicting a very rare event from a population versus something that's more prevalent.

For Continous Data

We evaluate error by *mean squared error* and it's root
* **Mean squared error (MSE)** - $\frac{1}{n}\sum_{i=1}^{n}(Prediction_i - Truth_i)^2$
* **Root mean squared error (RMSE)** - $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(Prediction_i - Truth_i)^2}$

Common Error Measures

1. Mean squared error (or root mean squared error)
   - Continous data, sensitive to outliers
2. Median absolute deviation
   - Median of distance between predicted and observed and take absolute value, rather than squared distance (this requries all values to be positive).
   - Continous data, often more robust
3. Sensitivity (recall)
   - If you want few missed positives
4. Specificity
   - If you want few negatives called positives
5. Accuracy

- Weights false positives/negatives equally

6. Concordance

- An example is **kappa**

7. Predictive value of a positive (precision)

- When you are screening and prevalence is low

Receiver Operating Characteristics (ROC Curves)

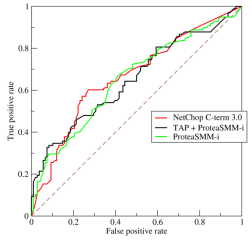
- Used to meaure the quality of a prediction algorithm

- **Wikipedia**
- Predictions are often quantitative
  - Proabbility fo being alive
  - Prediction on a scale from 1 to 10

- The *cutoff* you choose gives different results

- The curve informs you of the tradeoff of giving up some specificity for sensitivity (or vice versa)

- The curves plot the $P(FP)$ (x-axis) versus $P(TP)$ (y-axis)



Area Under the Curve

- The area under the curve (AKA the integral) describes the effectiveness of a given algorithm

- AUC = 0.5: random guessing

- AUC = 1: perfect classifier (given a certain value of the perdiction algorithm)
  - As such the closer to the top left of the plot a curve is the better it is.

- In general (depending on field & probability) an AUC above 0.8 is considered "good"

Cross Validation

Cross Validation

- A widely used tool for detecting relevant features and building models.

Key Ideas

1. Accuracy on the training set (resubstitution accuracy) is optimistic
2. A better estimate comes from an independent set (test set accuracy)
3. But we can't use the test set when building the model or it becomems part of the training set
4. So we estimate the test set accuracy with the training set

Cross-Validation Approach:
1. Use the training set
2. Split it into training/test sets (seperate from actual test set)
3. Build a model on the training set
4. Evaluate on the test set
5. Repeat with new training/test sets and average the estimated errors

What Cross-Validation is used for:
1. Picking variables to include in a model
2. Picking th etype of prediction function to use
3. Picking the parameters in the prediction function
4. Comparing different predictors

Ways to Pick Subsets

- Breaks data into K equal sized data sets.
- Leave out 1 sample then train on all the others, repeat for all samples

Considerations

- For time series data, data must be used in "chunks"
- For k-fold cross validation
  - Larger k = less bias, more variance
  - Smaller k = more bias, less variance
- Random sampling must be done *without replacement*
- Random sampling with replacement is the *bootstrap*
  - Underestimates of the error

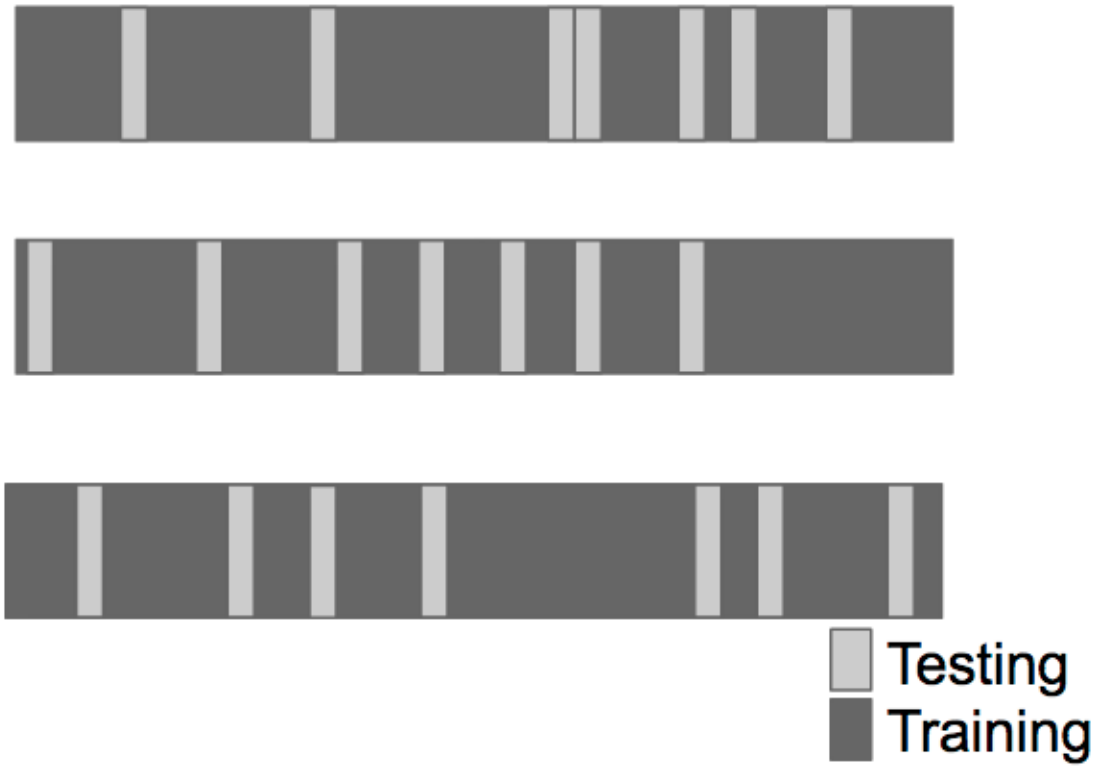


Figure 3: Random Subsampling

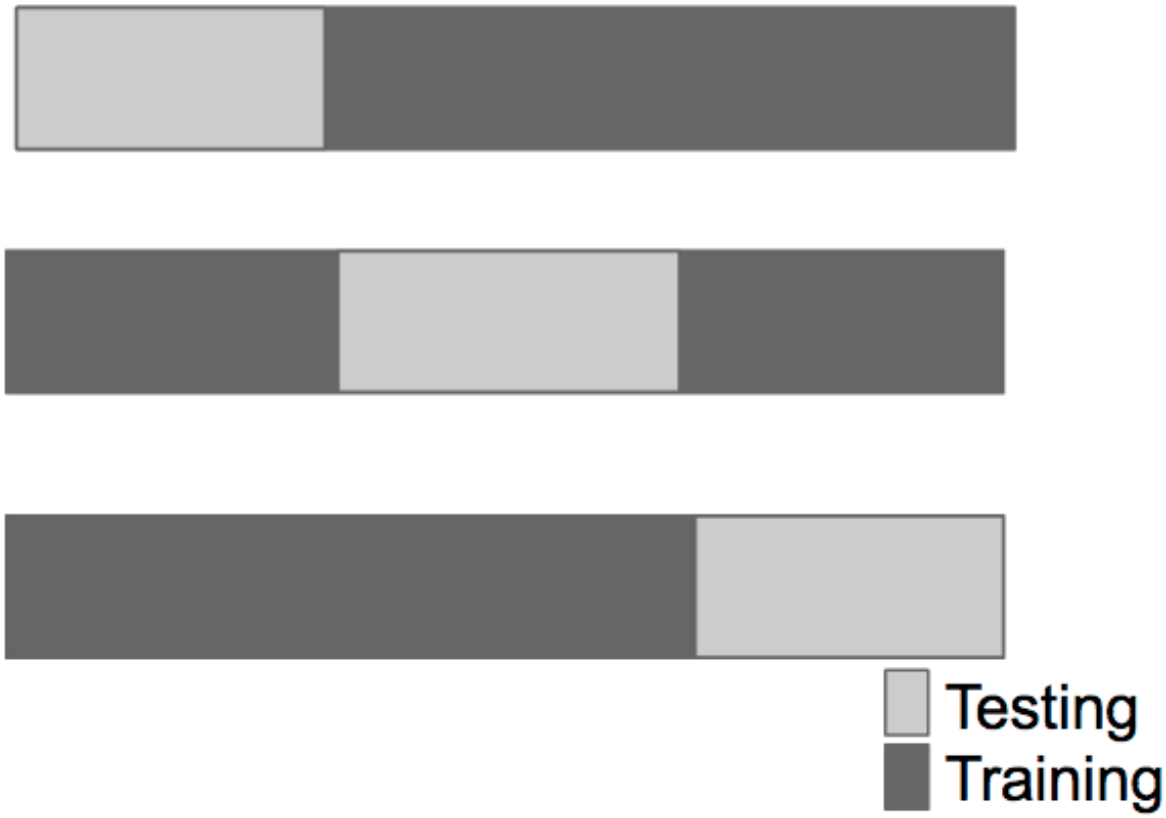


Figure 4: K-folds

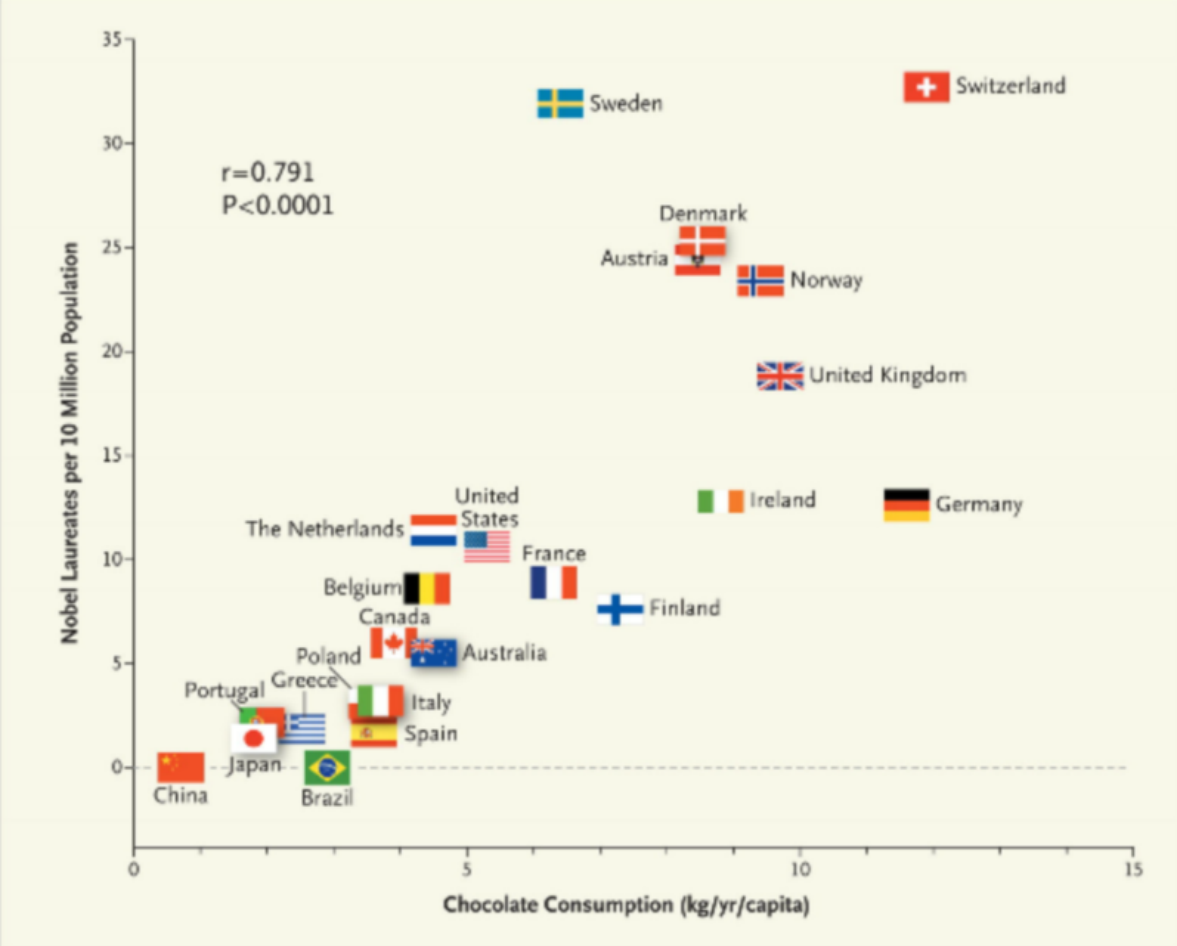
Figure 5: Leave One Out

  - Can be corrected, but it is complicated (**0.632 Bootstrap**)

- If you cross-validate to pick predictors estimate you must estiamte errors on independent data.

What Data Should You Use?

- If you want to predict something about X use data related to X (Use like to predict like)
  - To predict player performance use data about player performance
  - To predict movie preferences use data about movie preferences
  - To predict hospitalizations sue data about hospitalizations
- The closer the data is to the process you want to predict the better the predictions will be
- The looser connection the harder the prediction may be
  - *On Cotype DX* uses gene expression to predict one's longevity and effectiveness of treatments for those with breast cancer.
- Unrelated data is the most common mistake, **for example:**



– There are many alternate variables one could look at that are more realisticly correlated

Quiz 1

(Note 1-4 were multiple choice of material covered in the notes)

5. Suppose that we have created a machine learning algorithm that predicts whether a link will be clicked with 99% sensitivity and 99% specificity. The rate the link is clicked is 1/1000 of visits to a website. If we predict the link will be clicked on a specific visit, what is the probability it will actually be clicked?

```
sens <- 0.99  #TP/(TP+FN)
spec <- 0.99  #TN/(FP+TN)
rate <- 1/1000 #sum(TP+FN), 1-r is sum(FP+TN)
#ppv = TP/(TP + FP)

#rate = TP+FN,
#rate - TP = FN
#sens * (TP+FN) = TP,
#sens * (TP + rate - TP) = TP,
#TP = sens * rate
```

```r
TP <- sens * rate

#1 - rate = FP + TN,
#1 - rate - FP = TN

#spec * (FP+TN) = TN,
#spec * (FP+1 - rate - FP) = 1 - rate - FP
#spec * (1 - rate) = 1 - rate - FP
# FP = 1 - rate - (spec* (1-rate))
# FP = (1-rate)*(1-spec)
FP <- (1-rate)*(1-spec)
ppv <- TP/(TP + FP)
ppv
```

```
## [1] 0.09016393
```

The Caret Package

Caret Package

Caret Package

- Can be installed with `install.packages("caret")`, details about the package **can be found on cran**

Functionality

- Some preprocessing (cleaning)
  - `preProcess`
- Data splitting
  - `createDataPartition`
  - `createResample`
  - `createTimeSlices`
- Training/testing functions
  - `train`
  - `predict`
- Model comparison
  - `confusionMatrix`

Machine Learning Algorithms in Base R

- Linear discriminant analysis
- Regression
- Naive Bayes
- Support vector machines
- Classification and regression trees
- Random forests
- Boosting
- etc.
- The interface for these algorithms is slightly different

obj Class	Package	predict Function Syntax
lda	MASS	predict(obj) (no options needed)
glm	stats	predict(obj, type = "response")
gbm	gbm	predict(obj, type = "response", n.trees)
mda	mda	predict(obj, type = "posterior")
rpart	rpart	predict(obj, type = "prob")
Weka	RWeka	predict(obj, type = "probability")
LogitBoost	caTools	predict(obj, type = "raw", nIter)

- The `caret` package unifies these differences

SPAM Example: Data Splitting

```
library(caret)
```

```
## Loading required package: lattice
## Loading required package: ggplot2
##
## Attaching package: 'ggplot2'
## The following object is masked from 'package:kernlab':
##
##     alpha
```

```
library(kernlab) #For data
data(spam)
set.seed(32343)
inTrain <- createDataPartition(y = spam$type,
                               p = 0.75, #Proportion to subset
                               list = FALSE) # =F returns indecies
training <- spam[inTrain, ]
testing <- spam[-inTrain, ]
dim(spam)
```

```
## [1] 4601   58
```

```
dim(training)
```

```
## [1] 3451   58
```

```
dim(spam)[1]*0.75 #Showing it took 75%
```

```
## [1] 3450.75
```

```
set.seed(32343)
modelFit <- train(type ~ ., data = training, method = "glm")
modelFit
```

```
## Generalized Linear Model
##
## 3451 samples
##   57 predictor
##    2 classes: 'nonspam', 'spam'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 3451, 3451, 3451, 3451, 3451, 3451, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9159559  0.8227903
```

- `Resampling: Bootstrapped (25 reps)` indicates that it used the Bootstrap method, with 25 replicates. It corrects for the error that can occur when using the bootstrap method

```
modelFit$finalModel
```

```
##
## Call:  NULL
##
## Coefficients:
##       (Intercept)               make            address                all
##         -1.473e+00         -2.692e-01         -1.386e-01          8.483e-02
##             num3d                our               over             remove
##          3.634e+00          5.485e-01          6.465e-01          2.539e+00
```

```
##          internet             order              mail           receive
##         7.626e-01         4.876e-01         1.035e-01        -4.715e-01
##              will            people            report         addresses
##        -1.121e-01        -1.117e-01         6.115e-02         2.411e+00
##              free          business             email               you
##         9.280e-01         7.069e-01         8.624e-02         9.791e-02
##            credit              your              font             num000
##         9.980e-01         2.176e-01         1.327e-01         2.126e+00
##             money                hp               hpl            george
##         7.839e-01        -2.013e+00        -8.222e-01        -8.944e+00
##            num650               lab              labs            telnet
##         9.072e-02        -2.061e+00         8.513e-03         5.930e-01
##            num857              data            num415             num85
##         1.061e+00        -8.148e-01        -1.355e+01        -2.857e+00
##        technology            num1999             parts                pm
##         1.236e+00         6.600e-02        -5.405e-01        -1.218e+00
##            direct                cs           meeting          original
##        -3.825e-01        -4.398e+01        -3.209e+00        -1.245e+00
##           project                re               edu             table
##        -1.349e+00        -9.048e-01        -1.822e+00        -2.303e+00
##        conference      charSemicolon   charRoundbracket  charSquarebracket
##        -2.908e+00        -1.316e+00        -7.632e-02        -6.030e-01
##   charExclamation         charDollar          charHash         capitalAve
##         3.538e-01          5.103e+00         2.956e+00        -9.560e-03
##       capitalLong       capitalTotal
##         9.809e-03          7.312e-04
##
## Degrees of Freedom: 3450 Total (i.e. Null);  3393 Residual
## Null Deviance:       4628
## Residual Deviance: 1382  AIC: 1498
```

- This shows all how all the variables are weighted

SPAM Example: Prediction

```
predictions <- predict(modelFit, newdata = testing)
predictions[1:30]
```

```
##  [1] spam    spam    spam    spam    spam    nonspam spam    spam    spam
## [10] spam    spam    spam    spam    spam    spam    spam    spam    spam
## [19] spam    spam    spam    nonspam spam    nonspam spam    spam    spam
## [28] spam    spam    spam
## Levels: nonspam spam
```

SPAM Example: Confusion Matrix

```
confusionMatrix(predictions, testing$type)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction nonspam spam
##    nonspam     672   52
##    spam         25  401
##
##                Accuracy : 0.933
##                  95% CI : (0.917, 0.9468)
##     No Information Rate : 0.6061
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8583
##
##  Mcnemar's Test P-Value : 0.003047
##
##             Sensitivity : 0.9641
##             Specificity : 0.8852
##          Pos Pred Value : 0.9282
##          Neg Pred Value : 0.9413
##              Prevalence : 0.6061
##          Detection Rate : 0.5843
##    Detection Prevalence : 0.6296
##       Balanced Accuracy : 0.9247
##
##        'Positive' Class : nonspam
##
```

- First gives a table of the predicted vs. true value
- Gives summary statistics
  - Accuracy & 95% CI for the accuracy
  - No Information Rate is the average loss, L, of f over all combinations of y_i and x_j, expressed with the formula: $\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} L(y_i, f(x_j))$
  - The **Mcnemar's Test P-Value** has a null hypothesis that the error rates are equivelant

Further Information

- Caret tutorials:
  - **PDF caret tutorial**
  - **cran vignette PDF**
  - **A paper introducing the caret package**

Data Slicing

SPAM Example: Data Splitting

```
library(caret)
library(kernlab); data(spam)
inTrain <- createDataPartition(y = spam$type,
                               p = 0.75, list = FALSE)
training <- spam[inTrain, ]
testing <- spam[-inTrain, ]
```

SPAM Example: K-fold

```
set.seed(32323)
folds <- createFolds(y=spam$type, k = 10,
                     list = TRUE, returnTrain = TRUE) #Returns sample positions
sapply(folds, length)
```

```
## Fold01 Fold02 Fold03 Fold04 Fold05 Fold06 Fold07 Fold08 Fold09 Fold10
##   4141   4140   4141   4142   4140   4142   4141   4141   4140   4141
```

```
folds[[1]][1:10]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
#You can also have it return the test set
set.seed(32323)
folds <- createFolds(y=spam$type, k = 10,
                     list = TRUE, returnTrain = FALSE) #Returns sample positions
sapply(folds, length)
```

```
## Fold01 Fold02 Fold03 Fold04 Fold05 Fold06 Fold07 Fold08 Fold09 Fold10
##    460    461    460    459    461    459    460    460    461    460
```

```
folds[[1]][1:10]
```

```
##  [1] 24 27 32 40 41 43 55 58 63 68
```

SPAM Example: Resampling

```
set.seed(32323)
folds <- createResample(y = spam$type, times = 10,
                        list = TRUE)
sapply(folds, length)
```

```
## Resample01 Resample02 Resample03 Resample04 Resample05 Resample06 Resample07
##       4601       4601       4601       4601       4601       4601       4601
## Resample08 Resample09 Resample10
##       4601       4601       4601
```

```
folds[[1]][1:10] #Contains some resampled values
```

```
##  [1]  1  2  3  3  3  5  5  7  8 12
```

SPAM Example: Time Slices

```
set.seed(32323)
tme <- 1:1000
folds <- createTimeSlices(y = tme,
                          initialWindow = 20, #consecutive ... training set
                          horizon = 10)#Consecutive values in each test set
names(folds)
```

```
## [1] "train" "test"
```

```
folds$train[[1]]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
folds$test[[1]]
```

```
##  [1] 21 22 23 24 25 26 27 28 29 30
```

Training Options

```
## Still using SPAM set
library(caret)
library(kernlab); data(spam)
inTrain <- createDataPartition(y = spam$type, p = 0.75, list = FALSE)
training <- spam[inTrain, ]
testing <- spam[-inTrain, ]
modelFit <- train(type ~., data = training, method = "glm")
```

Train Options

```
args(caret:::train.default)
```

```
## function (x, y, method = "rf", preProcess = NULL, ..., weights = NULL,
##     metric = ifelse(is.factor(y), "Accuracy", "RMSE"), maximize = ifelse(metric %in%
##         c("RMSE", "logLoss", "MAE"), FALSE, TRUE), trControl = trainControl(),
##     tuneGrid = NULL, tuneLength = ifelse(trControl$method ==
##         "none", 1, 3))
## NULL
```

- One can change ...
  - `preProcess` to change preprocessing options (covered later)
  - `weights` to assign weights to the variables (Useful for unbalanced training set)

- `metric` to change what is measured, default is that Accuracy is measured for catagorigal variables and RMSE otherwise. Below are some of the other options:
  - RMSE = Root mean squared error
  - RSquared = R^2 from regression models
  - Accuracy = Fraction correct
  - Kappa = A measure of **concordance**
- `trControl` = Calls to `trainControl` function which has more of it's own settings:

```
args(trainControl)
```

```
## function (method = "boot", number = ifelse(grepl("cv", method),
##     10, 25), repeats = ifelse(grepl("[d_]cv$", method), 1, NA),
##     p = 0.75, search = "grid", initialWindow = NULL, horizon = 1,
##     fixedWindow = TRUE, skip = 0, verboseIter = FALSE, returnData = TRUE,
##     returnResamp = "final", savePredictions = FALSE, classProbs = FALSE,
##     summaryFunction = defaultSummary, selectionFunction = "best",
##     preProcOptions = list(thresh = 0.95, ICAcomp = 3, k = 5,
##         freqCut = 95/5, uniqueCut = 10, cutoff = 0.9), sampling = NULL,
##     index = NULL, indexOut = NULL, indexFinal = NULL, timingSamps = 0,
##     predictionBounds = rep(FALSE, 2), seeds = NA, adaptive = list(min = 5,
##         alpha = 0.05, method = "gls", complete = TRUE), trim = FALSE,
##     allowParallel = TRUE)
## NULL
```

```
- ‘method‘ will determine how it samples data, along with the ‘number‘ of times and how many t
- ‘initialWindow‘ and ‘horizon‘ is for time based data
- ‘savePredictions‘ if true will return all the predictors of each model
- ‘summaryFunction‘ will determine the kind of summary returned
- ‘preProcOptions‘ (Preprocessing options)
- ‘seeds‘ is available to set seeds for all the diffrent resampling layers, helpful when runni
```

trainControl Resampling

- `method`
  - *boot* = bootstrapping
  - *boot632* = bootstrapping with adjustment
  - *cv* = cross validation
  - *repeatedcv* = repeated cross validation
  - *LOOCV* = leave one out cross validation
- `number`

  - For boot/cross validation
  - Number of subsamples to take

- `repeats`
  - Number of times to repeat subsampling
  - If value is big this can slow things down
- **More info on model training and tuning**

Plotting Predictors

```
#The example in this lesson will be using wages data from the ISLR package
library(ISLR); data(Wage)
library(tidyverse)
library(caret)
summary(Wage)
```

```
##       year           age                     maritl           race
##  Min.   :2003   Min.   :18.00   1. Never Married: 648   1. White:2480
##  1st Qu.:2004   1st Qu.:33.75   2. Married      :2074   2. Black: 293
##  Median :2006   Median :42.00   3. Widowed      :  19   3. Asian: 190
##  Mean   :2006   Mean   :42.41   4. Divorced     : 204   4. Other:  37
##  3rd Qu.:2008   3rd Qu.:51.00   5. Separated    :  55
##  Max.   :2009   Max.   :80.00
##
##                education                     region               jobclass
##  1. < HS Grad      :268   2. Middle Atlantic   :3000   1. Industrial :1544
##  2. HS Grad        :971   1. New England       :   0   2. Information:1456
##  3. Some College   :650   3. East North Central:   0
##  4. College Grad   :685   4. West North Central:   0
##  5. Advanced Degree:426   5. South Atlantic    :   0
##                           6. East South Central:   0
##                           (Other)              :   0
##             health      health_ins      logwage           wage
##  1. <=Good     : 858   1. Yes:2083   Min.   :3.000   Min.   : 20.09
##  2. >=Very Good:2142   2. No : 917   1st Qu.:4.447   1st Qu.: 85.38
##                                      Median :4.653   Median :104.92
##                                      Mean   :4.654   Mean   :111.70
##                                      3rd Qu.:4.857   3rd Qu.:128.68
##                                      Max.   :5.763   Max.   :318.34
##
```

- From this we can see the data is all from Males in the Middle Atlantic region

```
## Creating training/test sets
set.seed(1618033)
inTrain <- createDataPartition(y = Wage$wage,
```

```
                                  p = 0.7, list = FALSE)
training <- Wage[inTrain, ]
testing <- Wage[-inTrain, ]
dim(training); dim(testing)
```

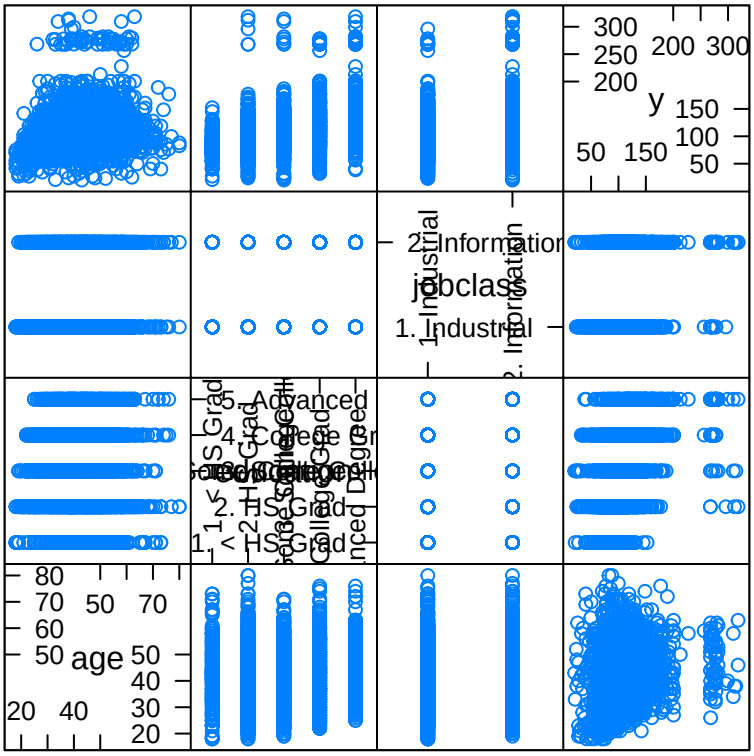
```
## [1] 2102   11
```

```
## [1] 898  11
```

Looking at the Data

featurePlot

```
featurePlot(x = training[, c("age", "education", "jobclass")],
            y = training$wage,
            plot = "pairs")
```



Scatter Plot Matrix

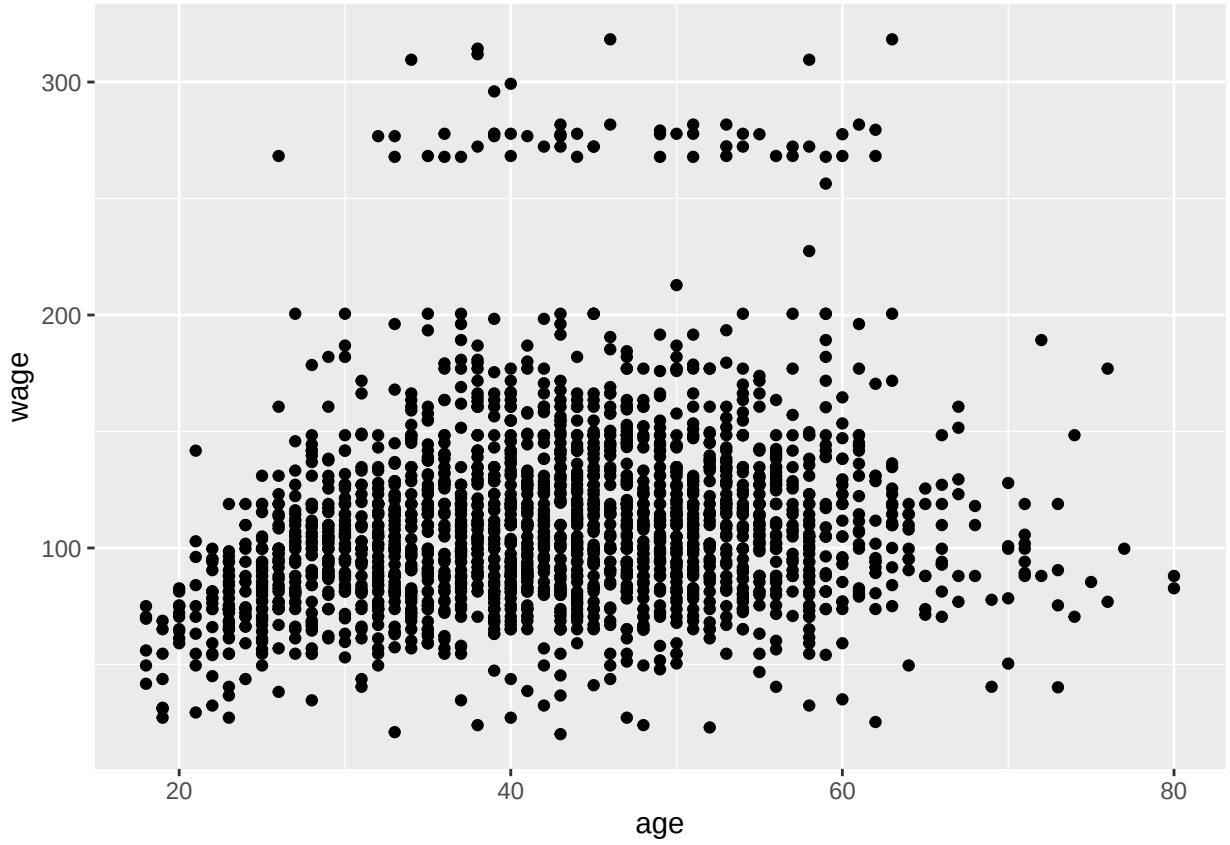
- When looking at this plot one is looking for trends in the data, for example the second column of the first row shows a semi-positive correlation to the X, when deciphering the mess of words one can see this is our y, `wage`, against `education`; indicating higher education might correlate to a higher wage.

- Two catagorical variables against eachother are hard to decipher meaning from as they largely overlap.

qplot

- Quick plots in style of `ggplot`

```
qplot(age, wage, data = training)
```

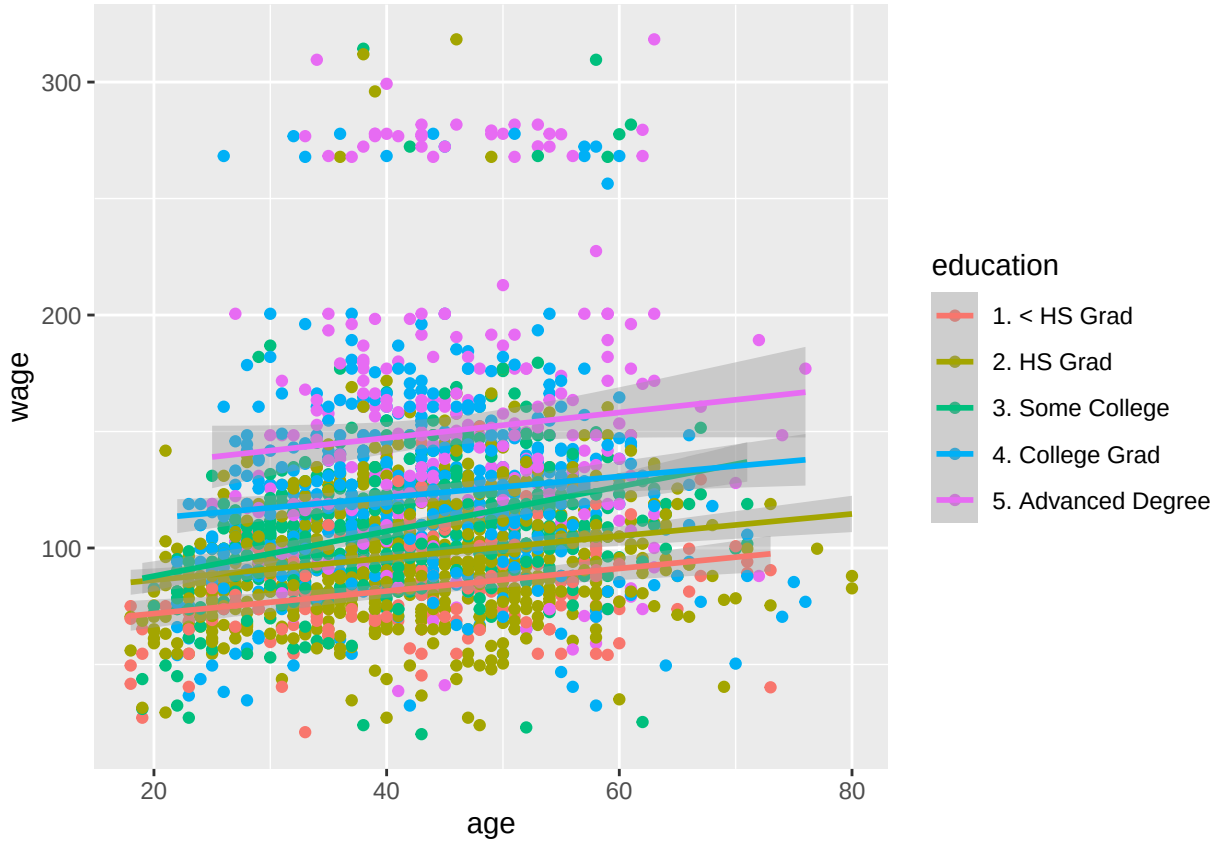


- The odd subset of wages that are away from the others may be cause for some concern, as such we'd want to investigate this before making the model

```
qplot(age, wage, colour = jobclass, data = training)
```

- You can also add regression smoothers to investigate differences more

```
plot <- qplot(age, wage, colour = education, data = training)
plot + geom_smooth(method = 'lm', formula = y ~ x)
```



Using cut2 to make factors

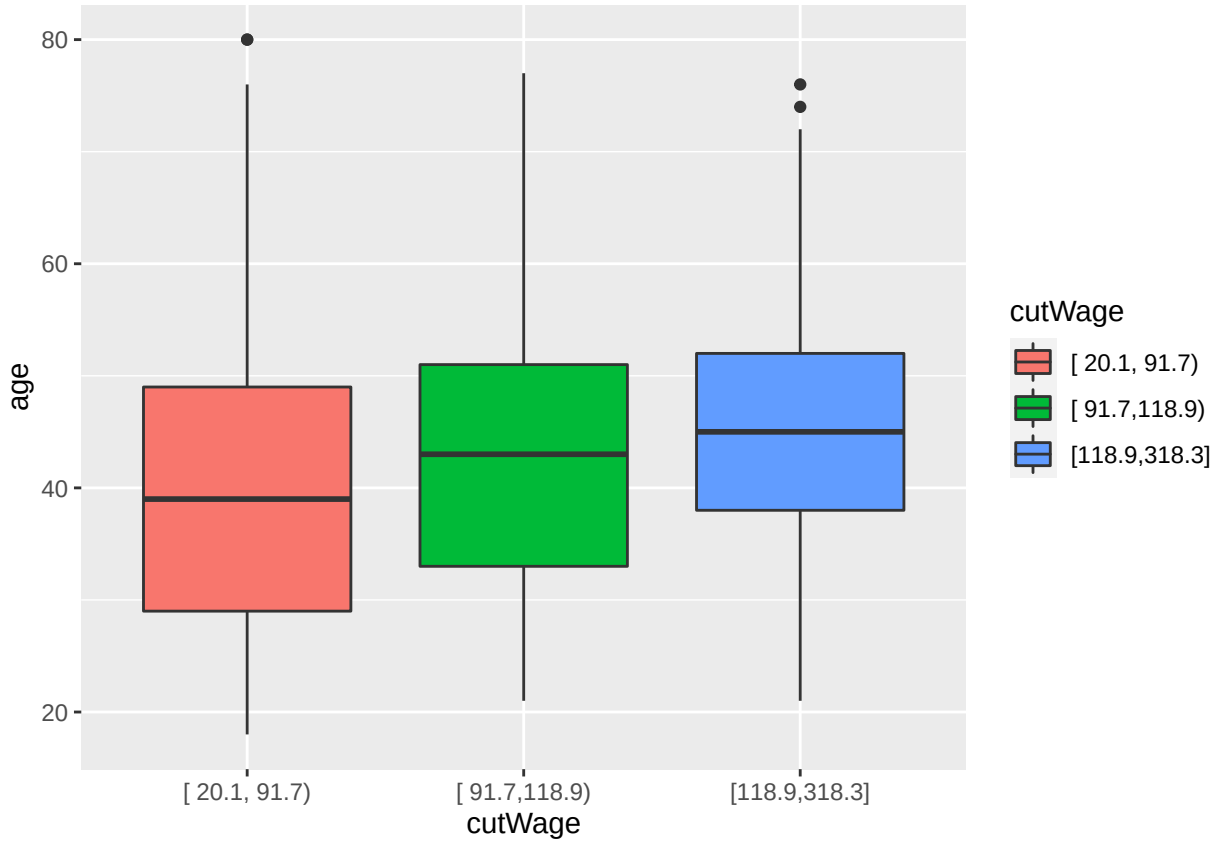
```
library(Hmisc)
```

```
## Loading required package: survival
##
## Attaching package: 'survival'
## The following object is masked from 'package:caret':
##
##     cluster
## Loading required package: Formula
##
## Attaching package: 'Hmisc'
## The following objects are masked from 'package:dplyr':
##
##     src, summarize
## The following objects are masked from 'package:base':
##
##     format.pval, units
```

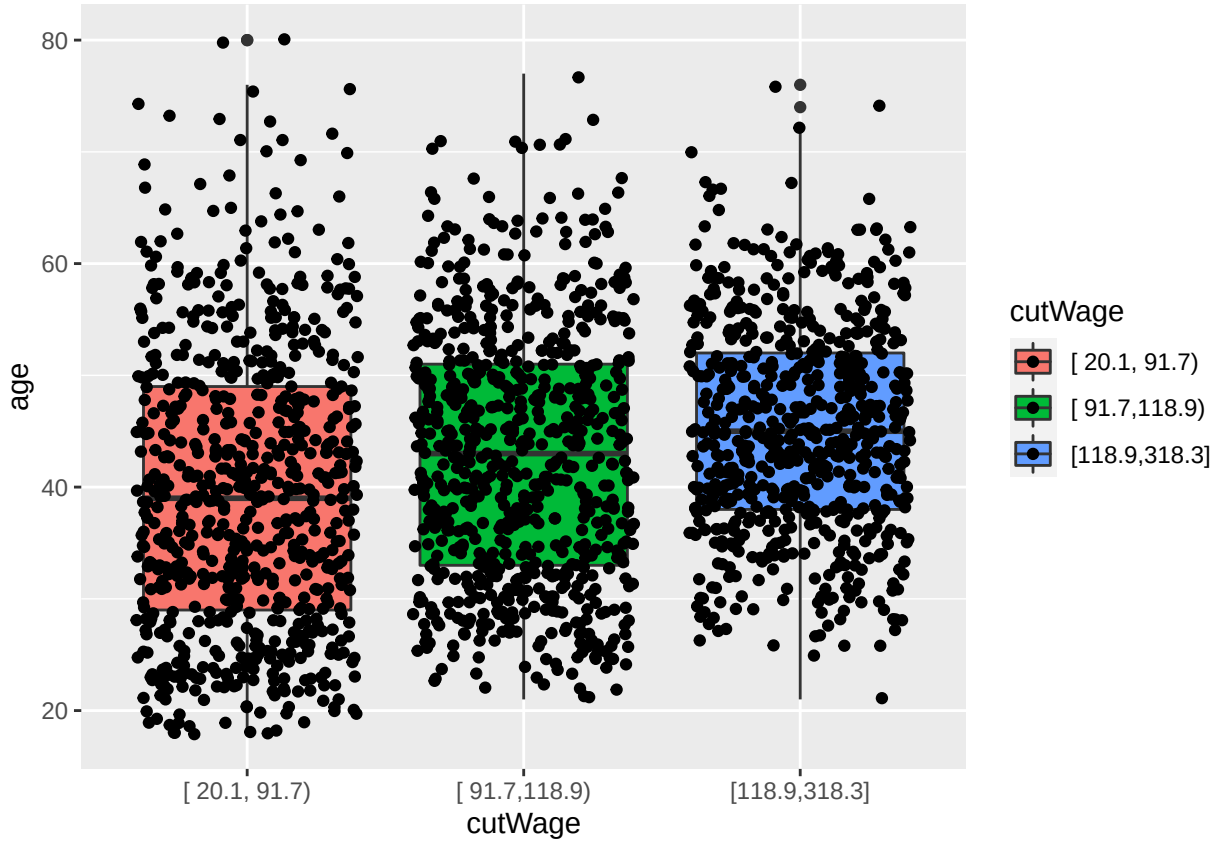
```
cutWage <- cut2(training$wage, g = 3)
table(cutWage)
```

```
## cutWage
## [ 20.1, 91.7) [ 91.7,118.9) [118.9,318.3]
##           704           727           671
```

```
bplot1 <- qplot(cutWage, age, data = training, fill = cutWage,
                geom = c("boxplot"))
bplot1
```




```
#Add points overlayed on boxplot
bplot2 <- qplot(cutWage, age, data = training, fill = cutWage,
                geom = c("boxplot", "jitter"))
bplot2
```



- Many points for each plot indicates the boxplots are well representing the data, if there were only a few then it would suggest the boxplots are not as representative

Tables

```
t1 <- table(cutWage, training$jobclass)
t1
```

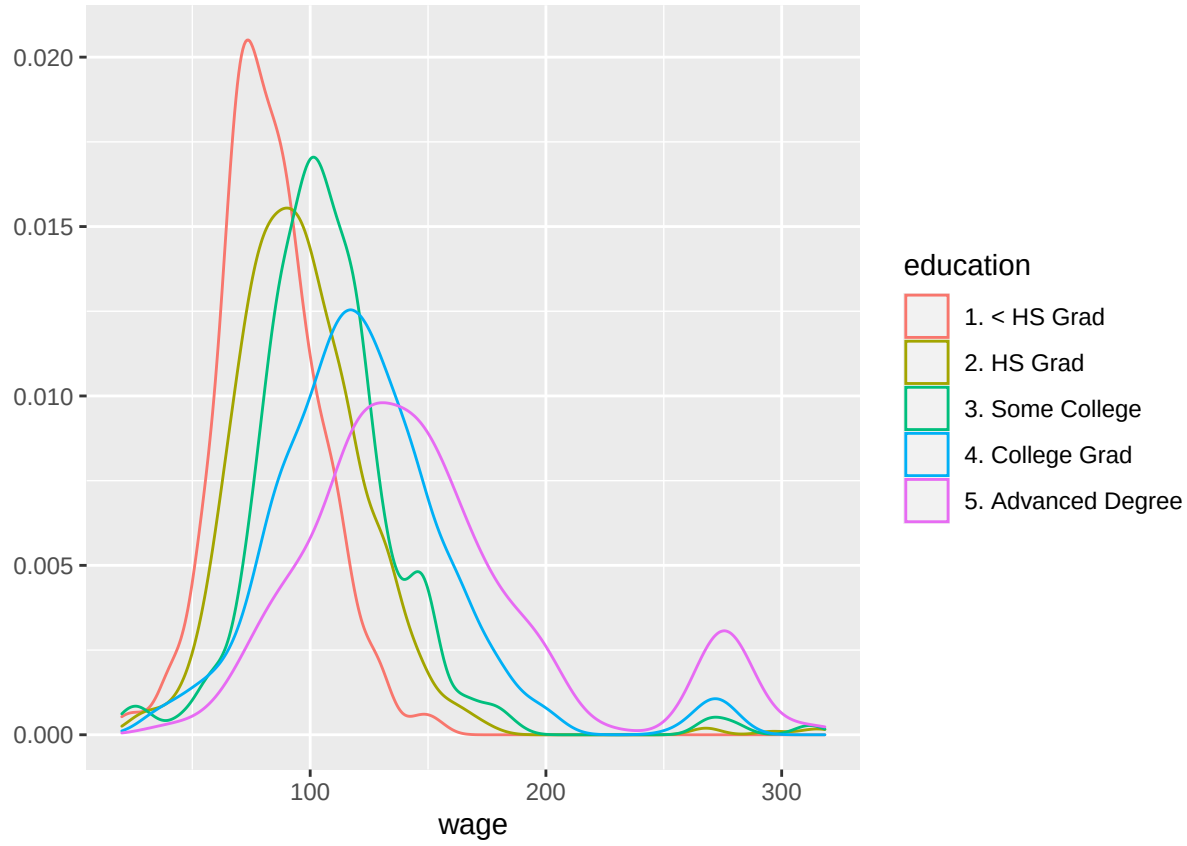
```
##
## cutWage          1. Industrial 2. Information
##   [ 20.1, 91.7)            439            265
##   [ 91.7,118.9)            380            347
##   [118.9,318.3]            266            405
```

```
prop.table(t1, 1) #Shows proportions, 1 for by row
```

```
##
## cutWage          1. Industrial 2. Information
##   [ 20.1, 91.7)     0.6235795      0.3764205
##   [ 91.7,118.9)     0.5226960      0.4773040
##   [118.9,318.3]     0.3964232      0.6035768
```

Density Plots

```
qplot(wage, colour = education, data = training, geom = "density")
```



Notes and Further Reading

- Make your plots only in the training set
  - Don't use the test set for exploration!
- Things one should be looking for
  - Imbalance in outcomes/predictors
  - Outliers
  - Groups of points not explained by a predictor
  - Skewed variables
- **ggplot2 tutorial**
- **caret visualizations**

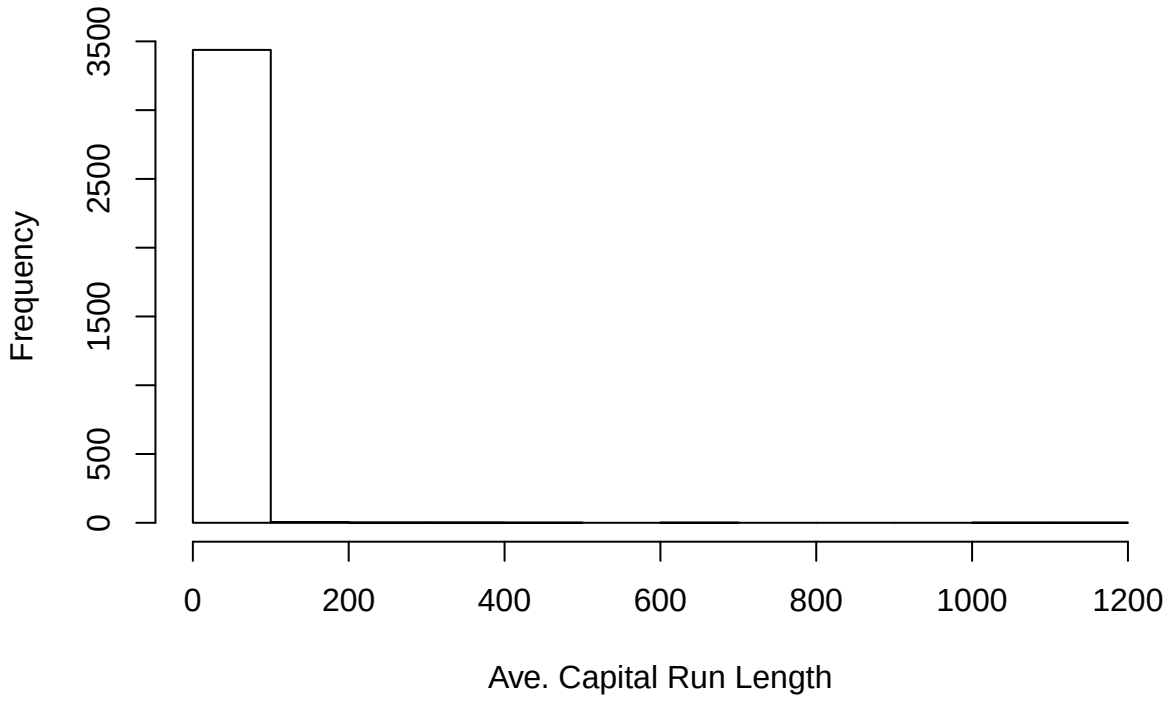
Preprocessing

Basic Preprocessing

- Sometimes variables need to be transformed to be more helpful for prediction algorithms

Why Preprocess? : Looking at SPAM Example

```
library(kernlab); data(spam)
library(caret)
set.seed(1618033)
inTrain <- createDataPartition(y = spam$type,
                               p = 0.75, list = FALSE)
training <- spam[inTrain, ]
testing <- spam[-inTrain, ]
hist(training$capitalAve, main = "", xlab = "Ave. Capital Run Length")
```



- There are a few lengths that are large causing this overall distribution to be quite skewed, so you may want to standardize the variable. This can be seen because of the large standard deviation, 31.51.

Standardizing

- Subtract the mean from each variable then divide by the sd.

```
trainCapAve <- training$capitalAve
trainCapAveS <- (trainCapAve - mean(trainCapAve))/sd(trainCapAve)
mean(trainCapAveS)
```

```
## [1] 1.410437e-17
```

```
sd(trainCapAveS)
```

```
## [1] 1
```

- Standardizing a data set will make the mean nearly 0 and the sd 1 as it's now measuring z-scores

- When evaluating on the test set one can only use parameters that were estiamted from the training set. In other words when we standardize the testing set with have to use the mean & sd of the training set. This will result in the mean and sd not be 1 and 0, respectively.

```
testCapAve <- testing$capitalAve
testCapAveS <- (testCapAve -
                  mean(trainCapAve))/sd(trainCapAve)
mean(testCapAveS)
```

```
## [1] 0.02486654
```

```
sd(testCapAveS)
```

```
## [1] 1.028051
```

Standardizing with preProcess Function

```
preObj <- preProcess(training[,-58], #Passing all variables except SPAM/HAM
                     method = c("center", #Subtracts mean
                                "scale")) #Divides by sd
trainCapAveS <- predict(preObj, training[,-58])$capitalAve
mean(trainCapAveS)
```

```
## [1] 1.410437e-17
```

```
sd(trainCapAveS)
```

```
## [1] 1
```

- You can then use this object returned by `preProcess` to pre process the test set

```
testCapAveS <- predict(preObj, testing[,-58])$capitalAve
mean(testCapAveS)
```

```
## [1] 0.02486654
```

```
sd(testCapAveS)
```

```
## [1] 1.028051
```

Standardizing with preProcess Argument

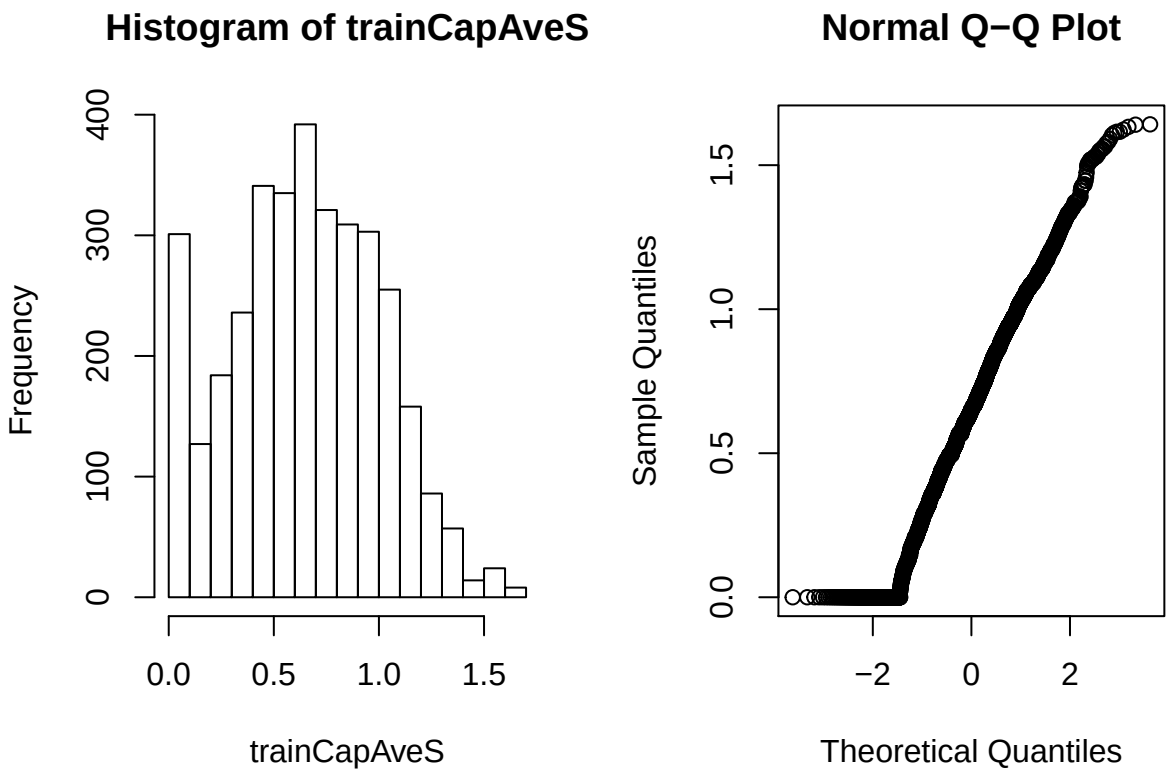
```
set.seed(32343)
modelFit <- train(type ~., data = training,
                  preProcess = c("center", "scale"),
                  method = "glm")
modelFit
```

```
## Generalized Linear Model
##
## 3451 samples
##   57 predictor
##    2 classes: 'nonspam', 'spam'
##
## Pre-processing: centered (57), scaled (57)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 3451, 3451, 3451, 3451, 3451, 3451, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9228539  0.8376812
```

Standardizing - Box-Cox Transforms

- BoxCox transforms are a set of transforms that take continous data and try to make them look like normal data. They do this by estimating a set of parameters based on maximum likelyhood

```
preObj <- preProcess(training[,-58], method = c("BoxCox"))
trainCapAveS <- predict(preObj, training[,-58])$capitalAve
par(mfrow = c(1,2))
hist(trainCapAveS); qqnorm(trainCapAveS)
```



- Note that there are still a stack of values at or around 0
  - This is because BoxCox does not take care of repeated variables

Imputing Data

```
set.seed(13343)

# Creating some NA values
training$capAve <- training$capitalAve
selectNA <- rbinom(dim(training)[1], size=1, prob = 0.05) == 1
training$capAve[selectNA] <- NA

# Impute and standardize
preObj <- preProcess(training[,-58],
                     method = "knnImpute") #K Nearest Neighbors Impute
capAve <- predict(preObj, training[,-58])$capAve

# Standardize true values
capAveTruth <- training$capitalAve
capAveTruth <- (capAveTruth - mean(capAveTruth))/sd(capAveTruth)

# How close was the imputting
quantile(capAve - capAveTruth)  #all
```

```
##            0%           25%           50%           75%          100%
## -2.0270107756  0.0002311304  0.0013492908  0.0018670773  0.1882803231
```

```
quantile((capAve - capAveTruth)[selectNA]) #only missing value
```

```
##           0%          25%          50%          75%         100%
## -2.027010776 -0.008848495  0.004294905  0.024778725  0.188280323
```

```
quantile((capAve - capAveTruth)[!selectNA]) #exclude missing
```

```
##            0%           25%           50%           75%          100%
## -0.7887893359  0.0002916346  0.0013363640  0.0018225519  0.0022534422
```

Notes & Further Reading

- Training and test sets must be pre-processed in the same way
- Test transformations will likely be imperfect
  - Especially if the test/training sets are collected at different times or in different ways
- Careful when transforming factor variables!
- **preprocessing with caret**

Covariate Creation

- Covariates are also called *predictors* or *features*, they are what variables are used and combined to predict the outcome you're looking at.
- The raw data usually takes form of an image or text file so you want to turn that into a quantifiable predictor. The goal is to describe the data as much as possible while also compressing the key components of the data. In spam example we look at things like:
  - Proportion of capital letters, `capitalAve`
  - Frequency of a given word, `you`, or character, `numDollar` (dollar signs).
- Sometimes after collecting the variables we want to transform the data to a more useful value, like squaring a value.

Level 1, Raw Data -> Covariates

- Depends heavily on application
- The balancing act is summarization vs. information loss
- Examples:
  - Text files: frequency fo words, frequency of phrases (**Google ngrams**), frequency of capital letters.

  - Images: Edges, corners, blobs, ridges (**computer vision feature detection**)
  - Webpages: Number and type of images, position fo elements, colors, videos (**A/B Testing**)
  - People: Height, weight, hair color, sex, country of orgin.

- The more knowledge of the system you have the better the job you will do at extracting features.
- When in doubt, err on the side of more features
- Can be automated, but use caution!

Level 2, Tidy Covariates -> New Covariates

- More necessary for some methods (regression, support vector machines (svms)) than for others (classification trees).
- Deciding how to create them should be done *only on the training set*
  - When applying the prediction you'll make these same mutations
- The best approach is through exploratory analysis (plotting/tables)
- New covariates should be added to data frames with recognizable names

Wage Example

```
library(caret)
library(ISLR); data(Wage)
set.seed(1618033)
inTrain <- createDataPartition(y = Wage$wage,
                               p = 0.7, list = FALSE)
training <- Wage[inTrain, ]
testing <- Wage[-inTrain, ]
```

Adding "Dummy Variables"

```
table(training$jobclass)
```

```
##
##  1. Industrial 2. Information
##           1085           1017
```

- Since jobs are either `Industrial` or `Information` we may want to melt this factor variable into two seperate logical columns

```
dummies <- dummyVars(wage ~ jobclass, data = training)
head(predict(dummies, newdata = training))
```

```
##        jobclass.1. Industrial jobclass.2. Information
## 231655                      1                       0
## 86582                       0                       1
## 155159                      0                       1
## 377954                      0                       1
## 81404                       0                       1
## 302778                      0                       1
```

Removing Zero Covariates

```
nsv <- nearZeroVar(training, saveMetrics = TRUE)
nsv
```

```
##            freqRatio percentUnique zeroVar   nzv
## year        1.031884    0.33301618   FALSE FALSE
## age         1.054054    2.90199810   FALSE FALSE
## maritl      3.205298    0.23786870   FALSE FALSE
## race        8.554455    0.19029496   FALSE FALSE
## education   1.316532    0.23786870   FALSE FALSE
## region      0.000000    0.04757374    TRUE  TRUE
## jobclass    1.066863    0.09514748   FALSE FALSE
## health      2.440262    0.09514748   FALSE FALSE
## health_ins  2.395800    0.09514748   FALSE FALSE
## logwage     1.189873   19.12464320   FALSE FALSE
## wage        1.189873   19.12464320   FALSE FALSE
```

- A `freqRatio` near 0 indicates that it rarely has a unique value.
- `TRUE` in either `zeroVar` or `nzv` indicates that you likely can remove that variable without losing any context.

Spline Basis

```
library(splines)
bsBasis <- bs(training$age, df = 3)
head(bsBasis,16)
```

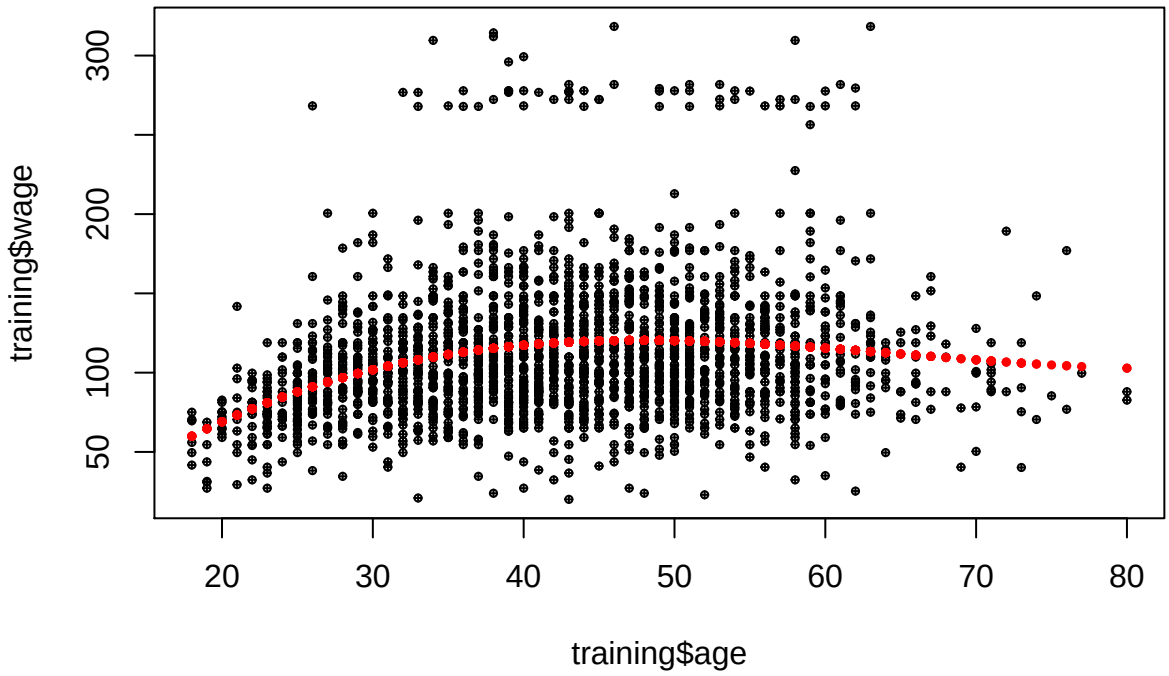
```
##                1          2           3
##  [1,] 0.00000000 0.00000000 0.000000000
##  [2,] 0.23685006 0.02537679 0.000906314
##  [3,] 0.43081384 0.29109043 0.065560908
##  [4,] 0.37763083 0.09063140 0.007250512
##  [5,] 0.33553758 0.40743849 0.164915579
##  [6,] 0.41633799 0.32117502 0.082587862
##  [7,] 0.42616898 0.14823269 0.017186399
##  [8,] 0.43333138 0.16370296 0.020614447
##  [9,] 0.44435820 0.22759810 0.038858212
## [10,] 0.30633413 0.42415495 0.195763821
## [11,] 0.44221829 0.19539878 0.028779665
```

```
## [12,] 0.36252559 0.38669397 0.137491189
## [13,] 0.27551945 0.43623913 0.230237320
## [14,] 0.44221829 0.19539878 0.028779665
## [15,] 0.01793746 0.20448709 0.777050955
## [16,] 0.44308684 0.24369776 0.044677923
```

- `bs` is a basis function that creates a polynomial variables of the `df` degree, with each column being a higher order.

- Including these covariates allows for more curved of a model fit

```
lm1 <- lm(wage ~ bsBasis, data = training)
plot(training$age, training$wage, pch = 10, cex = 0.5)
points(training$age, predict(lm1, newdata = training), col = "#FF0000", pch = 19, cex = 0.5)
```



Splines on the Test Set

- You have to create the predictions on the test set with the exact same percedure you used on the training set.

- As such we have to predict what the variables of the testing set will be with the prediction from the training set.

```
head(predict(bsBasis, age = testing$age), 10)
```

```
##                1          2          3
```

```
##  [1,] 0.0000000 0.00000000 0.000000000
##  [2,] 0.2368501 0.02537679 0.000906314
##  [3,] 0.4308138 0.29109043 0.065560908
##  [4,] 0.3776308 0.09063140 0.007250512
##  [5,] 0.3355376 0.40743849 0.164915579
##  [6,] 0.4163380 0.32117502 0.082587862
##  [7,] 0.4261690 0.14823269 0.017186399
##  [8,] 0.4333314 0.16370296 0.020614447
##  [9,] 0.4443582 0.22759810 0.038858212
## [10,] 0.3063341 0.42415495 0.195763821
```

Notes and Further Reading

- Level 1 feature creation (raw data to covariates)
  - Science is key. Google "feature extraction for [data type]"
  - Err on overcreation of features, you can always filter them out later in the process.
  - In some applications (images, voices) automated feature creation is possible/necessary
  - **A PDF on deep learning to auto create features for images and voice**
- Level 2 feature creation (covariates to new covariates)
  - The function `preProcess` in `caret` will handle some preprocessing
  - Create new covariates if you think they will improve fit
  - Use exploratory analysis on the training set for creating them
  - Be careful about overfitting!
- **preprocessing with caret**
- If you want to fit spline models, use the `gam` method in the `caret` package which allows smoothing fo multiple variables.
- More on feature creation/data tidying in the **Getting and Cleaning Data course**

Preprocessing with Principal Components Analysis (PCA)

- Often there will be multiple quantitative variables that are highly correlated, as such one may want to create a summary that contains most of the information from those quantitative variables

Correlated predictors

```
library(caret)
library(kernlab); data(spam)
set.seed(1618033)
inTrain <- createDataPartition(y = spam$type,
                               p = 0.75, list = FALSE)
training <- spam[inTrain, ]
testing <- spam[-inTrain, ]

M <- abs(cor(training[,-58]))#Remove SPAM/HAM value

#Every variable has a correlation of
# 1 with itself so we remove those
diag(M) <- 0
which(M > 0.8, arr.ind = TRUE)
```

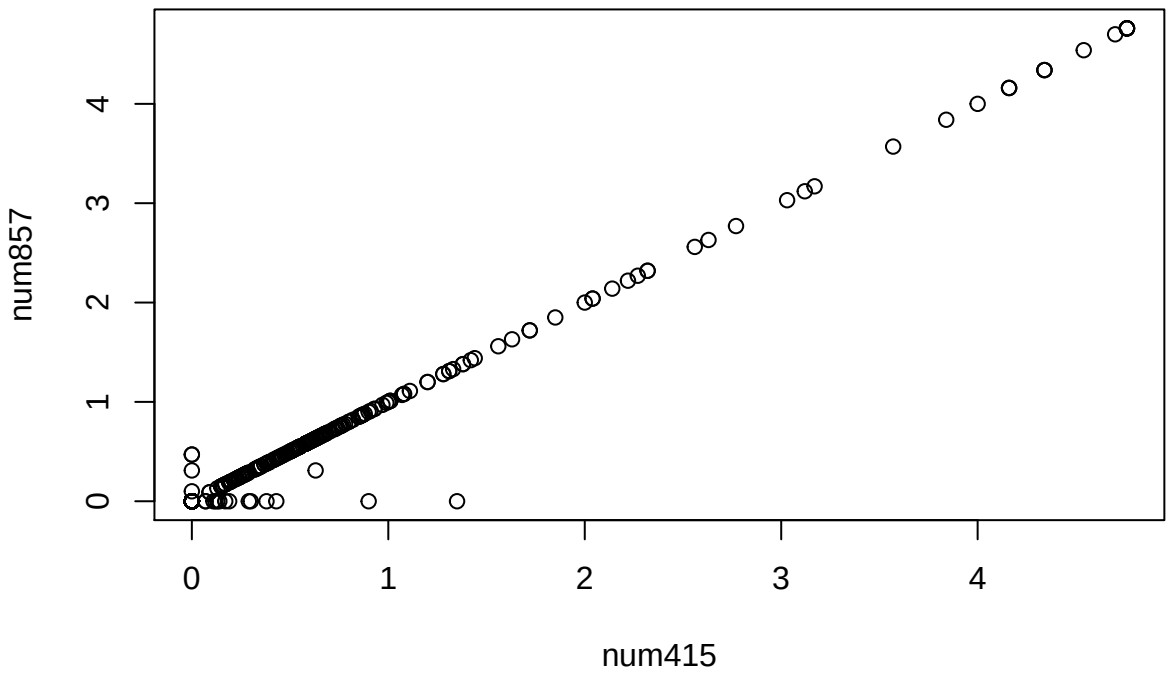
```
##        row col
## num415  34  32
## direct  40  32
## num857  32  34
## direct  40  34
## num857  32  40
## num415  34  40
```

- num415(col 32) and num857 (col 34) often appear together. It seems the word direct does too but we'll just look at the numbers in this example

```
rows <- c(34, 32, 40)
n <- names(spam)[rows]
n[1:2]
```

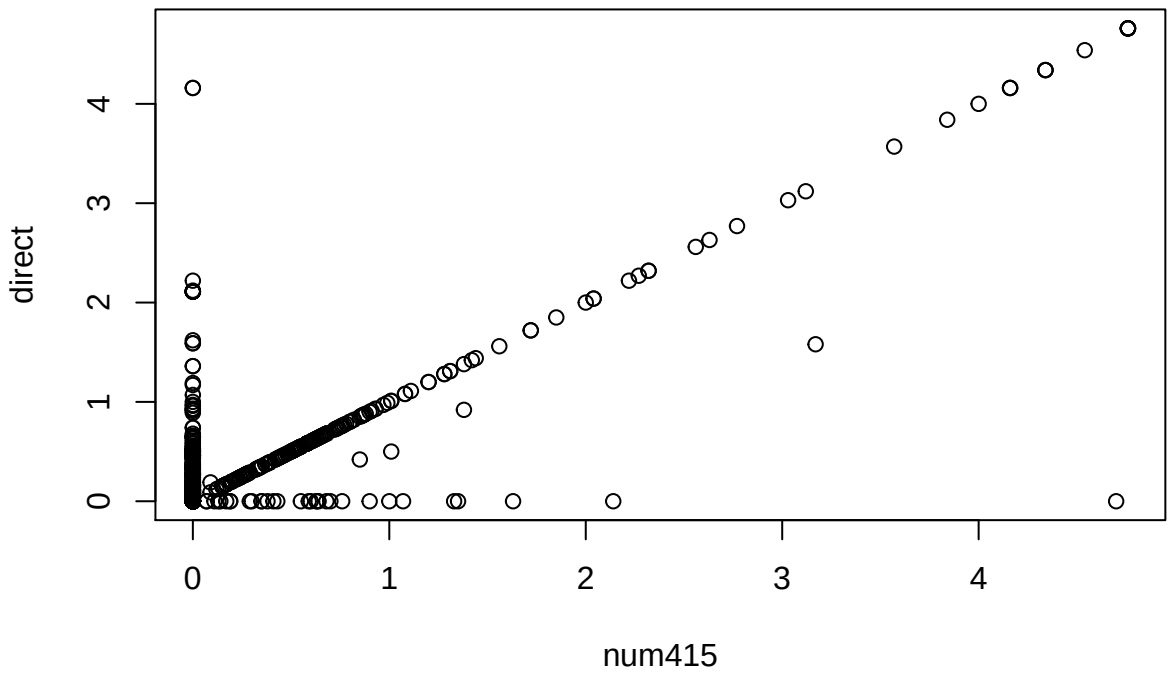
```
## [1] "num415" "num857"
```

```
plot(spam[,rows[1]], spam[,rows[2]], xlab = n[1], ylab = n[2])
```

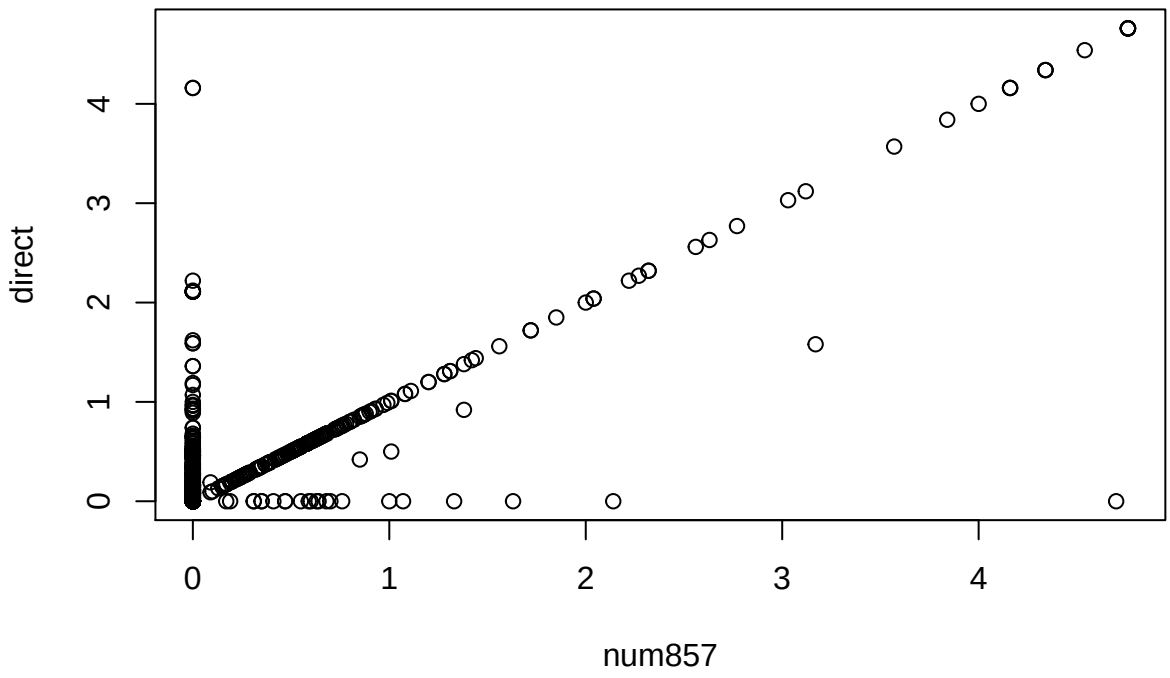


- Plotting them with eachother shows that they seem to be almost exactly 1-to-1. As such we could probably use less information to convey that both variables are present.

```
#Looking at plots of that direct variable too
plot(spam[,rows[1]], spam[,rows[3]], xlab = n[1], ylab = n[3])
```




```
plot(spam[,rows[2]], spam[,rows[3]], xlab = n[2], ylab = n[3])
```

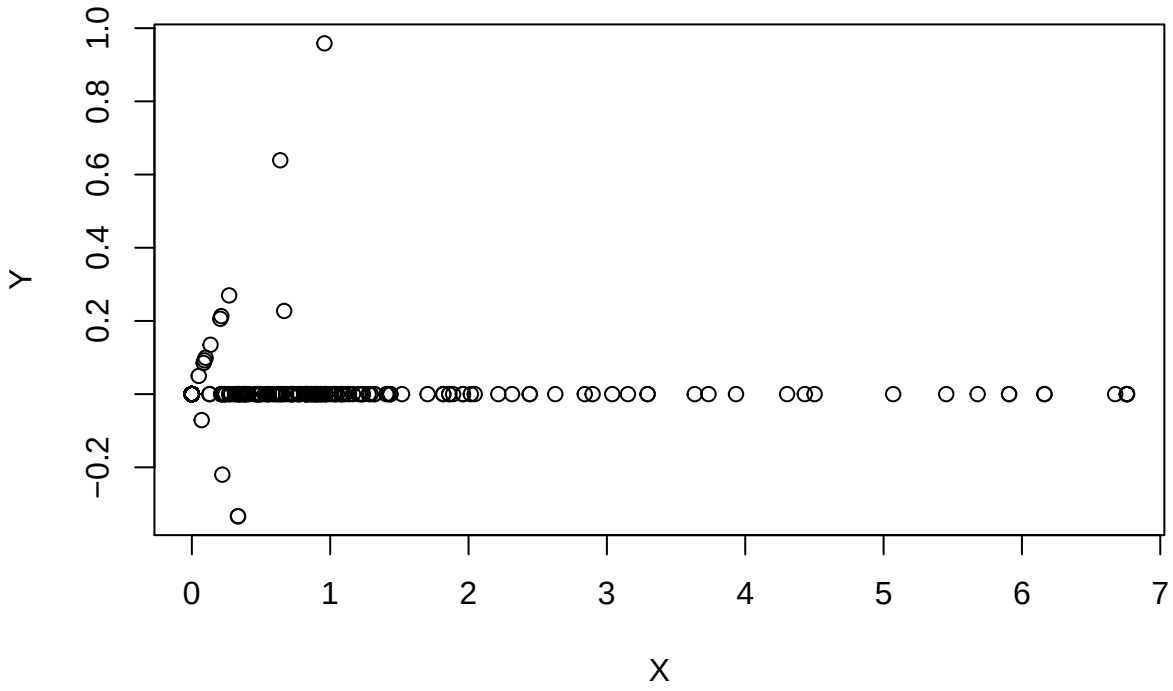


- `direct` may not be as strongly correlated but seeing the simularity in those two graphs builds a greater case for `num415` and `num857` being highly correlated.

We Could Rotate the Plot

- We'll be using `0.71` because it is a result from PCA later.

```
X <- 0.71*training$num415 + 0.71*training$num857
Y <- 0.71*training$num415 - 0.71*training$num857
plot(X,Y)
```



* We can see there is not a lot of variation on the Y axis, the difference
* Most of the variability occurs on the X axis, the sum, as such we may want to use the sum as a predictor since it captures most of the information from the two variables.

Basic PCA Idea

- We might not need every predictor
- A weighted combination fo predictors might be better
- We should pick this combination to capture the "most information" possible
- Benefits
  - Reduced numeber of predictors
  - Reduced noise (due to averaging)

Related Probelms

- You have multivariate variables $X_1, ..., X_n$ so $X_1 = (X_{11}, ..., X_{1m})$
  - Find a new set of multivariate variables that are uncorrelated and explain as much variance as possible.

  - If you put all the variables together in one matrix, find the best matrix created with fewer variables (lower rank) that explains the orginal data.

- The first goal is **statistical** and the second goal is **data compression**.

Related Solutions - PCA/SVD

- **I also have notes on this from the Exploratory Data Analysis course**

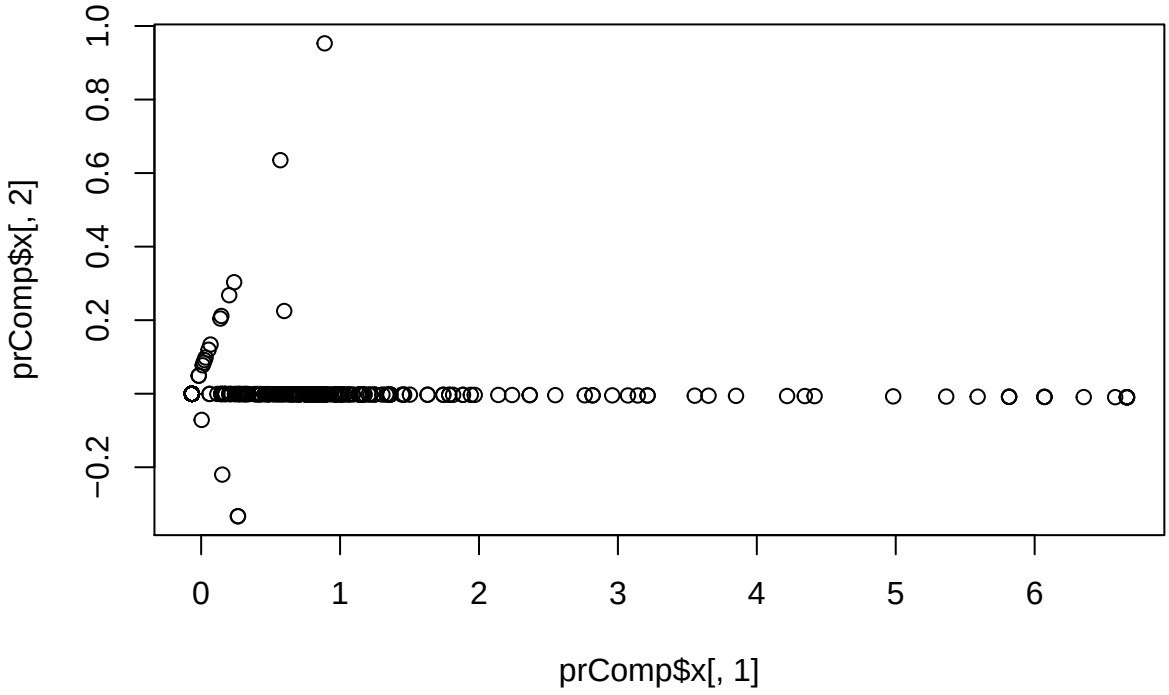
- SVD - Singular Value Decomposition
  - If X is a matrix with each variable in a column and each observation in a row then the SVD is a "matrix decomposition"
    $X = UDV^T$

  - Where the columns of U are orthogonal (left singular vectors), the columns of V are orthogonal (right singular vectors) and D is a diagonal matric (singular values).

- PCA - Principal Component Analysis
  - The principal components are equal tot eh right singular values if you first scale (subtract the mean, divide by the standard deviation) the variables.

Principal Components in R - prcomp

```
smallSpam <- spam[,c(34,32)]
prComp <- prcomp(smallSpam)
plot(prComp$x[,1], prComp$x[,2])
```



- We can look at how `prcomp` is summing the variables

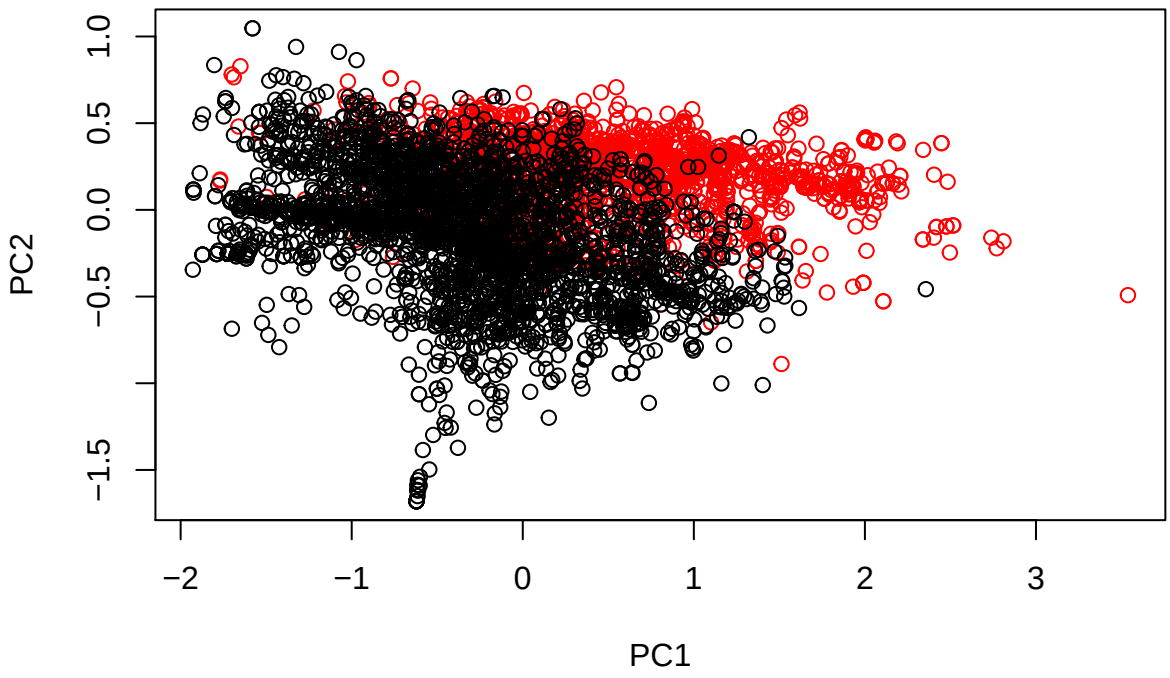
```
prComp$rotation
```

```
##               PC1        PC2
## num415 0.7080625  0.7061498
## num857 0.7061498 -0.7080625
```

- Taking the coefficients of `PC1` & `PC2` for the sum and difference respectively
- PCA allows one to perform this operation when dealing with more than just two variables

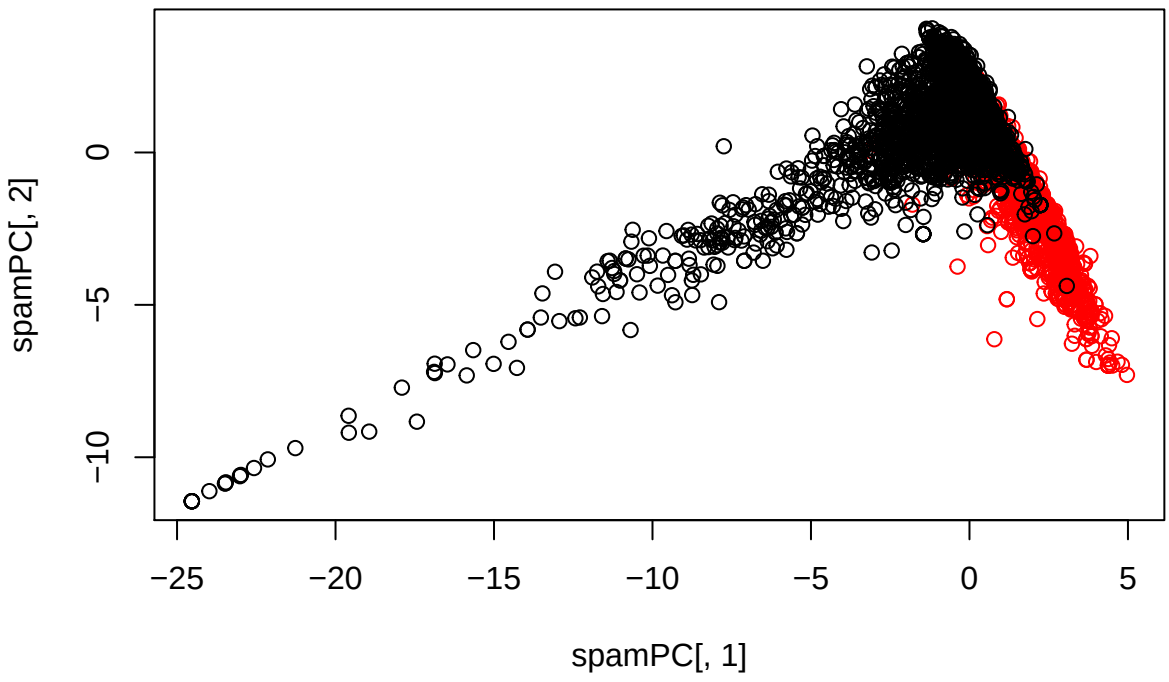
PCA on SPAM data

```
typeColor <- ((spam$type == "spam") * 1 + 1)# Red if SPAM, Black if HAM

# Apply log_10 to make data look more gaussian
prComp <- prcomp(log10(spam[,-58]+1))

plot(prComp$x[,1], prComp$x[,2], col = typeColor, xlab = "PC1", ylab = "PC2")
```



PCA with caret

```
preProc <- preProcess(log10(spam[,-58] + 1), method = "pca", pcaComp = 2)
spamPC <- predict(preProc, log10(spam[,-58] + 1))
plot(spamPC[,1], spamPC[,2], col = typeColor)
```



Preprocessing with PCA

```
preProc <- preProcess(log10(training[,-58] + 1),
                      method = "pca", pcaComp = 2)
trainPC <- predict(preProc, log10(training[,-58] + 1))
modelFit <- train(y = training$type, x = trainPC, method = "glm")
testPC <- predict(preProc, #Have to use training preProc procedure for test
                  log10(testing[, -58] + 1))
confusionMatrix(testing$type, predict(modelFit, testPC)) #Find result
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction nonspam spam
##    nonspam     661   36
##    spam         75  378
##
##                Accuracy : 0.9035
##                  95% CI : (0.8849, 0.9199)
##     No Information Rate : 0.64
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.7948
```

```
##
##  Mcnemar's Test P-Value : 0.00031
##
##             Sensitivity : 0.8981
##             Specificity : 0.9130
##          Pos Pred Value : 0.9484
##          Neg Pred Value : 0.8344
##              Prevalence : 0.6400
##          Detection Rate : 0.5748
##    Detection Prevalence : 0.6061
##       Balanced Accuracy : 0.9056
##
##        'Positive' Class : nonspam
##
```

Alternative: Built in PCA to train (sets # of PCs)

```
modelFit <- train(type ~ ., method = "glm",
                  preProcess = "pca", data = training)
confusionMatrix(testing$type, predict(modelFit, testing))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction nonspam spam
##    nonspam     675   22
##    spam         56  397
##
##                Accuracy : 0.9322
##                  95% CI : (0.9161, 0.946)
##     No Information Rate : 0.6357
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8561
##
##  Mcnemar's Test P-Value : 0.0001866
##
##             Sensitivity : 0.9234
##             Specificity : 0.9475
##          Pos Pred Value : 0.9684
##          Neg Pred Value : 0.8764
##              Prevalence : 0.6357
##          Detection Rate : 0.5870
##    Detection Prevalence : 0.6061
##       Balanced Accuracy : 0.9354
##
##        'Positive' Class : nonspam
##
```

Final Notes on PCs

- Most useful for linear-type models
- Can make it harder to itnerpret predictors
- Watch out for outliers!
  - Transform first (with logs/Box Cox)
  - Plot predictors to identify problems
- More info: **Elements of Statistical Learning**

Predicting

Predicting with Regression

- **Covered more fully in my notes for Regression Models**

Key Ideas

- Fit a Simple Regression Model
- Plug in new covariates and multiply by the coefficients
- Useful when the linear model is (nearly correct) **Pros:**
- Easy to implement
- Easy to interpret
  Cons:
- Often poor performance in nonlinear settings

Example: Old Faithful Eruptions

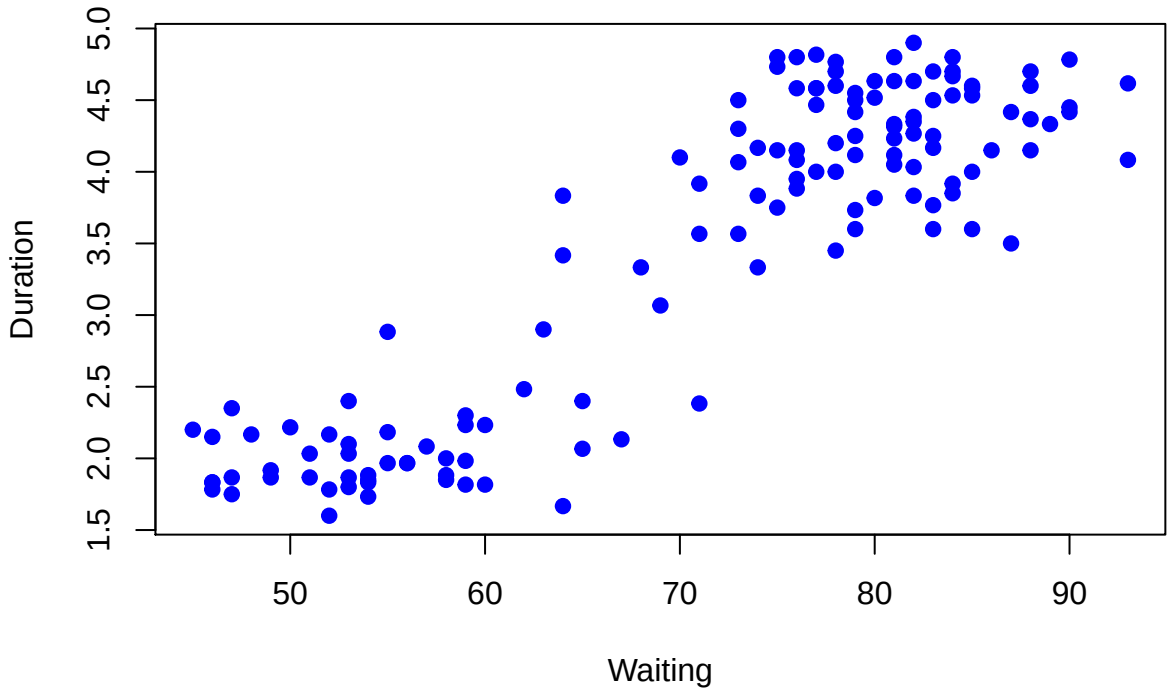
```
data(faithful)
library(caret)
set.seed(333)
inTrain <- createDataPartition(y = faithful$waiting, p = 0.5, list = FALSE)

trainFaith <- faithful[inTrain, ]
testFaith <- faithful[-inTrain, ]
head(trainFaith)
```

```
##   eruptions waiting
## 1     3.600      79
## 3     3.333      74
```

```
## 5     4.533      85
## 6     2.883      55
## 7     4.700      88
## 8     3.600      85
```

- Plot of eruption duration versus waiting time

```
plot(trainFaith$waiting, trainFaith$eruptions, pch = 19, col = "#0000FF",
     xlab = "Waiting", ylab = "Duration")
```



- One can see a line that may be able to go through these points that shows longer waiting correlates to longer erruptions

Fit a Linear Model

- Formula we fit is with an Eruption Duration, ED, and Wait Time, WT, and a gaussian distributed error term, e:
  $ED_i = b_0 + b_1 WT_i + e_i$

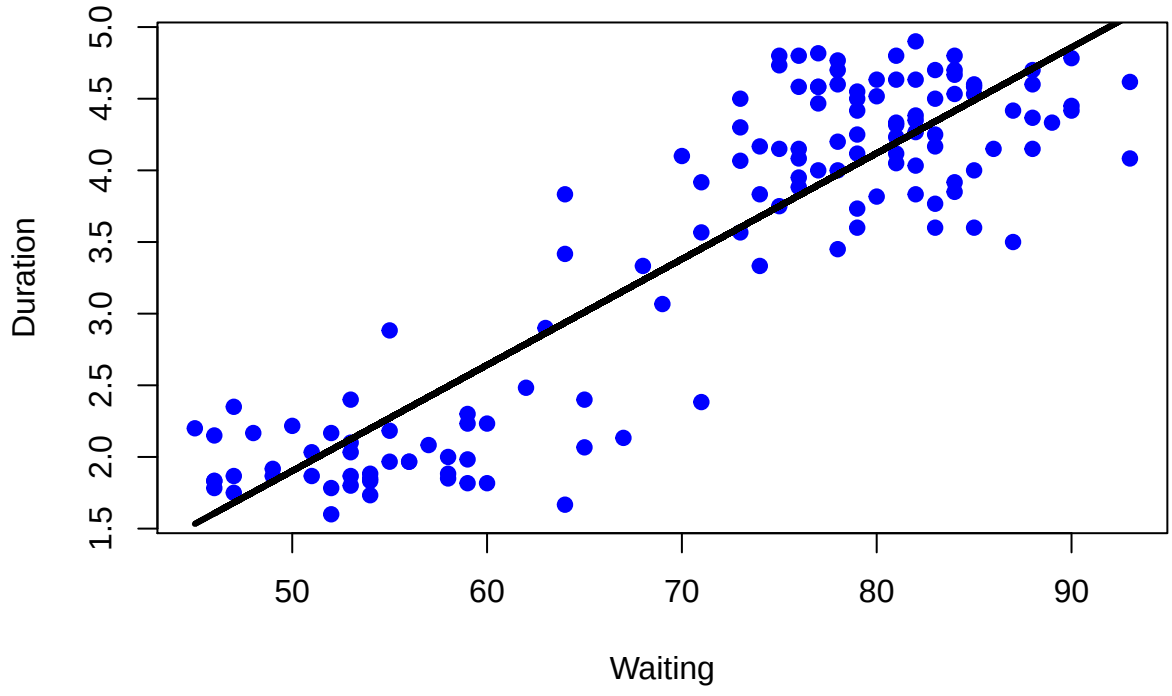
```
lm1 <- lm(eruptions ~ waiting, data = trainFaith)
summary(lm1)
```

```
##
## Call:
## lm(formula = eruptions ~ waiting, data = trainFaith)
```

```
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.26990 -0.34789  0.03979  0.36589  1.05020
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.792739   0.227869  -7.867 1.04e-12 ***
## waiting      0.073901   0.003148  23.474  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.495 on 135 degrees of freedom
## Multiple R-squared:  0.8032, Adjusted R-squared:  0.8018
## F-statistic:   551 on 1 and 135 DF,  p-value: < 2.2e-16
```

- These b_0 and b_1 values are given by the respective rows in the `Estimate` column.

```
plot(trainFaith$waiting, trainFaith$eruptions, pch = 19, col = "#0000FF",
     xlab = "Waiting", ylab = "Duration")
lines(trainFaith$waiting, lm1$fitted, lwd = 3)
```



Predict a New Value

- Explicitly

```
newValue <- 80
coef(lm1)[1] + coef(lm1)[2]*newValue
```

```
## (Intercept)
##    4.119307
```

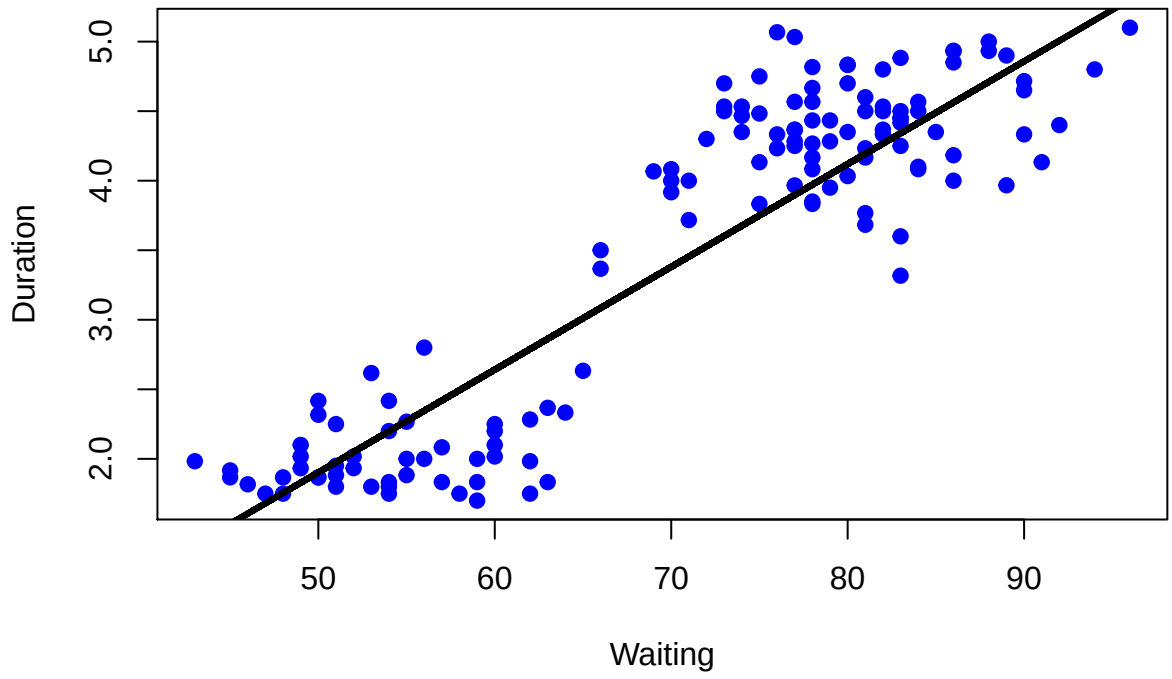
- With `predict`

```
#New value has to be named the same as explanatory variable
newValue <- data.frame(waiting = 80)
predict(lm1, newValue)
```

```
##        1
## 4.119307
```

Plot Predictions on the Test Set

```
plot(testFaith$waiting, testFaith$eruptions, pch = 19, col = "#0000FF",
     xlab = "Waiting", ylab = "Duration")
lines(testFaith$waiting,
      predict(lm1,
              newdata = testFaith), #Have to use info from train to predict
      lwd = 3)
```



Get Training/Test Set Errors

```
# Calculate RMSE on training
sqrt(sum(
  (lm1$fitted - trainFaith$eruptions)^2))
```

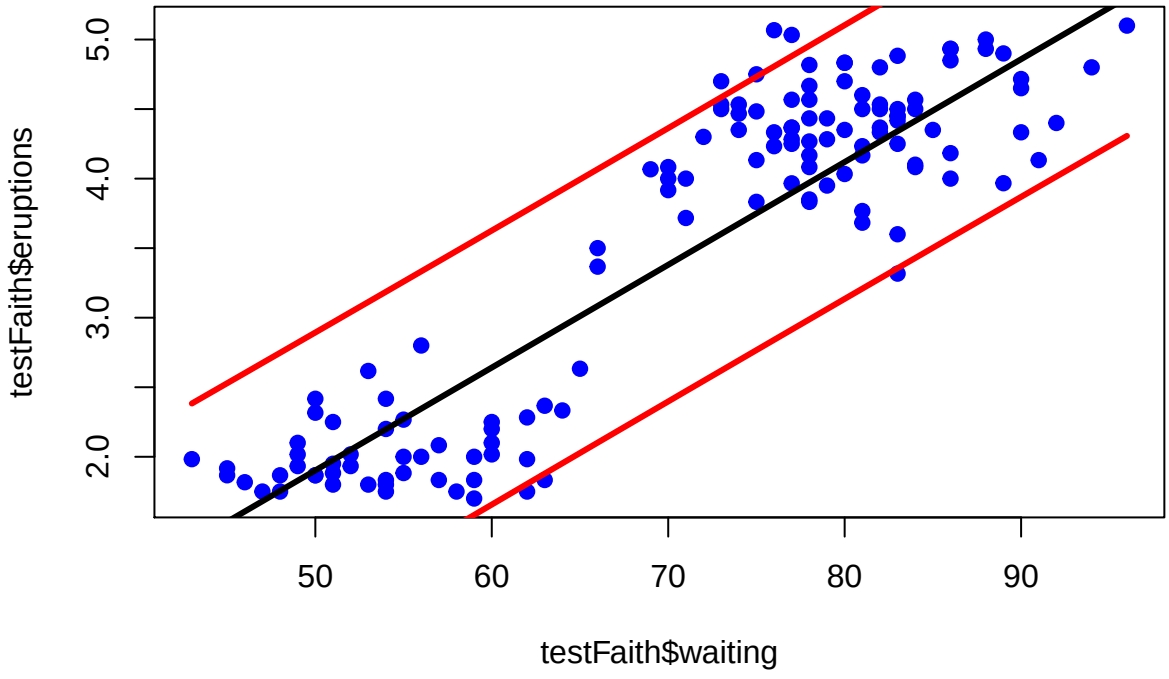
```
## [1] 5.75186
```

```
# Calculate RMSE on test
sqrt(sum(
  (predict(lm1, newdata = testFaith) - testFaith$eruptions)^2))
```

```
## [1] 5.838559
```

Prediction Intervals

```
pred1 <- predict(lm1, newdata = testFaith,
                 interval = "prediction") #Returns intervals
ord <- order(testFaith$waiting)
plot(testFaith$waiting, testFaith$eruptions, pch = 19, col = "#0000FF")
matlines(testFaith$waiting[ord], pred1[ord,], type= "l",
         col = c(1,2,2), lty = c(1,1,1), lwd = 3)
```



* Red lines are the 95% CI of the prediction

Same Process in caret Package

```
modFit <- train(eruptions ~ waiting, data = trainFaith, method = "lm")
summary(modFit$finalModel)
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.26990 -0.34789  0.03979  0.36589  1.05020
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.792739   0.227869  -7.867 1.04e-12 ***
## waiting      0.073901   0.003148  23.474  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.495 on 135 degrees of freedom
## Multiple R-squared:  0.8032, Adjusted R-squared:  0.8018
## F-statistic:   551 on 1 and 135 DF,  p-value: < 2.2e-16
```

Notes and Further Reading

- Regression models with multiple covariates can be included
- Often useful in combination with other models
- Further Reading:
  - **Elements of statistical learning**
  - **Modern applied statistics with S**
  - **Introduction to statistical learning**

Predicting with Regression Multiple Covariates

- We'll be using the `Wage` data

```
library(ISLR); data(Wage)
library(ggplot2)
library(caret)
```

- We're going to subset out the variable we're trying to predict, `logwage`

```
wage <- subset(Wage, select = -c(logwage))
summary(wage)
```

```
##       year           age                     maritl           race
##  Min.   :2003   Min.   :18.00   1. Never Married: 648   1. White:2480
##  1st Qu.:2004   1st Qu.:33.75   2. Married      :2074   2. Black: 293
##  Median :2006   Median :42.00   3. Widowed      :  19   3. Asian: 190
##  Mean   :2006   Mean   :42.41   4. Divorced     : 204   4. Other:  37
##  3rd Qu.:2008   3rd Qu.:51.00   5. Separated    :  55
##  Max.   :2009   Max.   :80.00
##
##                education                     region               jobclass
##  1. < HS Grad      :268   2. Middle Atlantic   :3000   1. Industrial :1544
##  2. HS Grad        :971   1. New England       :   0   2. Information:1456
##  3. Some College   :650   3. East North Central:   0
##  4. College Grad   :685   4. West North Central:   0
##  5. Advanced Degree:426   5. South Atlantic    :   0
##                           6. East South Central:   0
##                           (Other)              :   0
##             health      health_ins        wage
##  1. <=Good     : 858   1. Yes:2083   Min.   : 20.09
##  2. >=Very Good:2142   2. No : 917   1st Qu.: 85.38
##                                      Median :104.92
##                                      Mean   :111.70
##                                      3rd Qu.:128.68
##                                      Max.   :318.34
##
```

- Get training/test sets

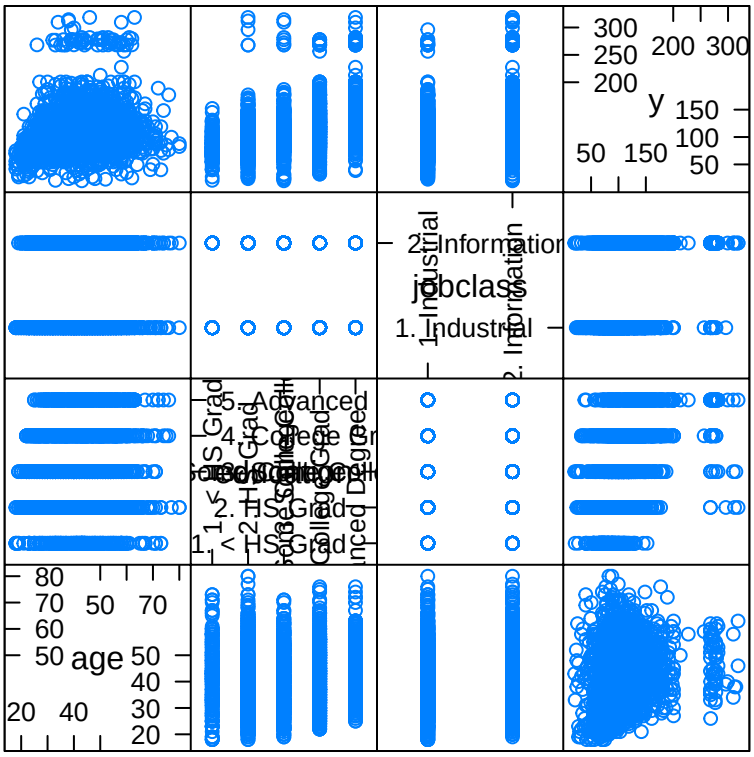
```
set.seed(1618033)
inTrain <- createDataPartition(y = wage$wage,
                               p = 0.7, list = FALSE)
training <- wage[inTrain, ]
testing <- wage[-inTrain, ]
dim(training); dim(testing)
```

```
## [1] 2102   10
```

```
## [1] 898  10
```

- Feature plot, sometimes helpful to detect trends

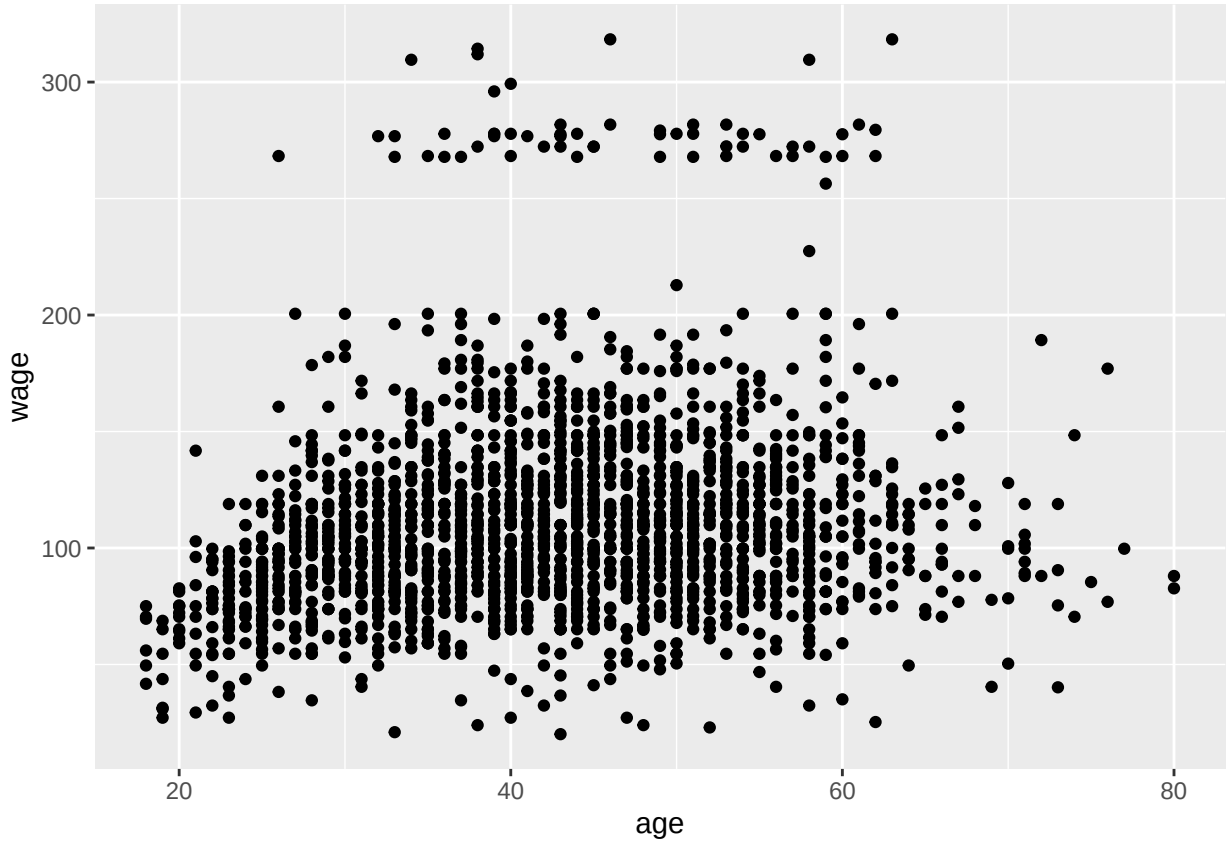
```
featurePlot(x = training[,c("age", "education", "jobclass")],
            y = training$wage,
            plot = "pairs")
```



Scatter Plot Matrix

- This lecture is largely covering some exploratory analysis done before on these data so I'm going to put them all here.

```
qplot(age, wage, data = training)
```




```
qplot(age, wage, colour = jobclass, data = training)
```


```
qplot(age, wage, colour = education, data = training)
```

Fit a Linear Model

$ED_i = b_0 + b_1 age + b_2 I(Jobclass_i = "Information") + \sum_{k=1}^{4} \eta_k I(education_i = levelk)$
* $I(Jobclass_i = "Information")$ is how an indicator variable is indicated in mathematical notation. Likewise for the $\eta_k I(education_i = levelk)$ which is checking for the 5 catagories (only 4 DF)

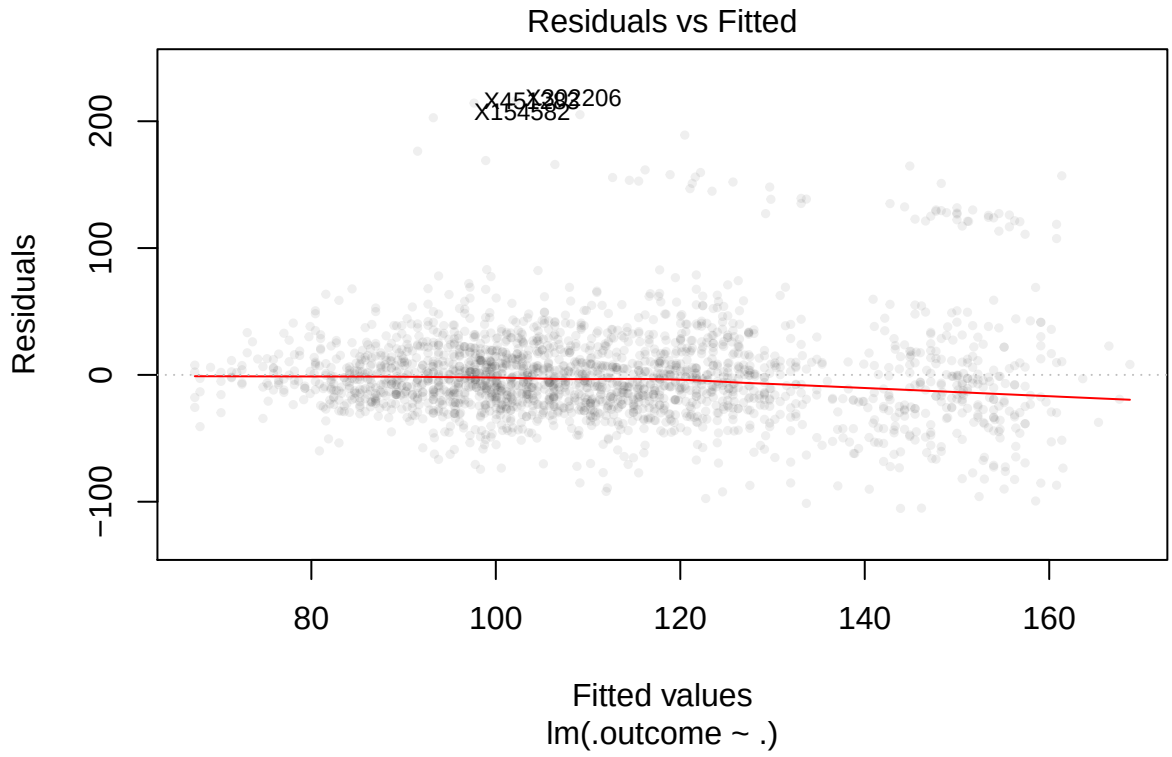
```
modFit <- train(wage ~ age + jobclass + education,
                method = "lm", data = training)
finMod <- modFit$finalModel
print(modFit)
```

```
## Linear Regression
##
## 2102 samples
##    3 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 2102, 2102, 2102, 2102, 2102, 2102, ...
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   36.87177  0.2378928  25.27525
```

```
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

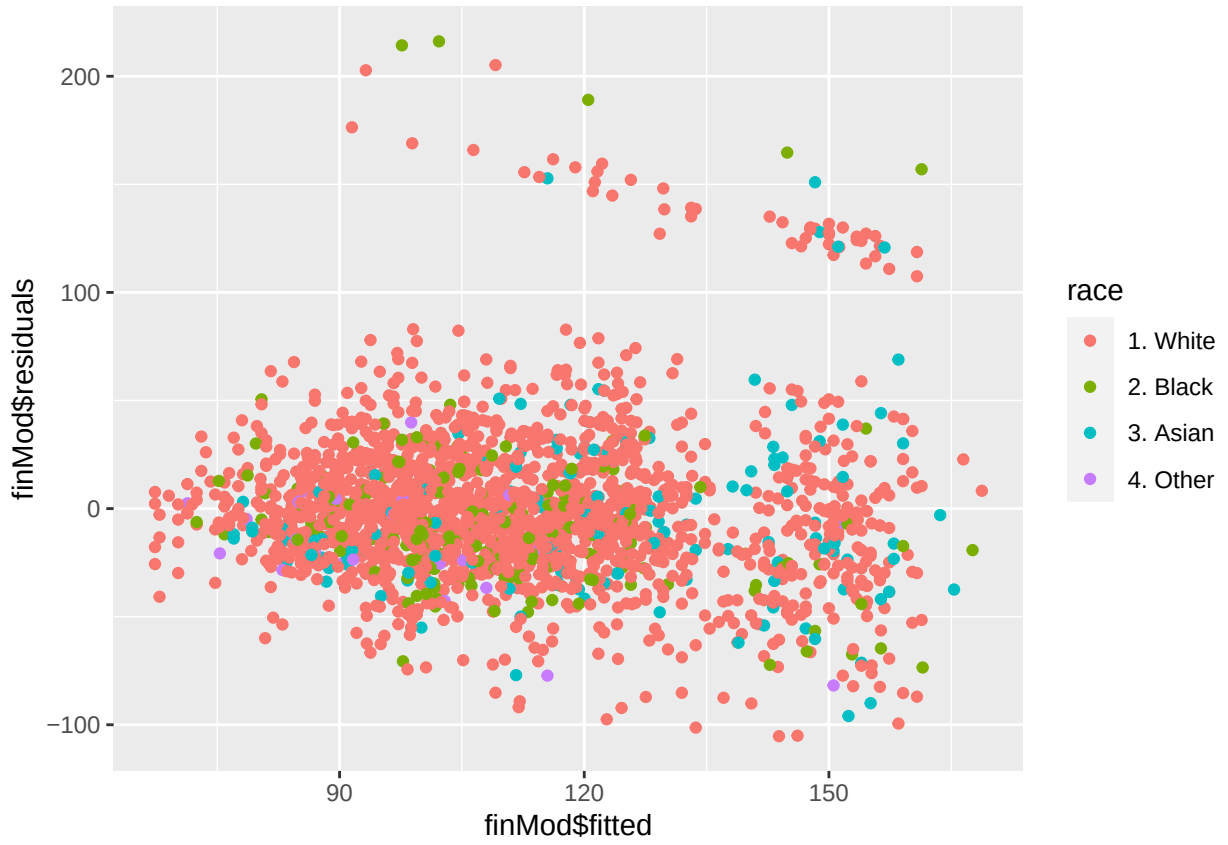
Diagnostics

```
plot(finMod, 1, pch = 19, cex = 0.5, col = "#00000010")
```



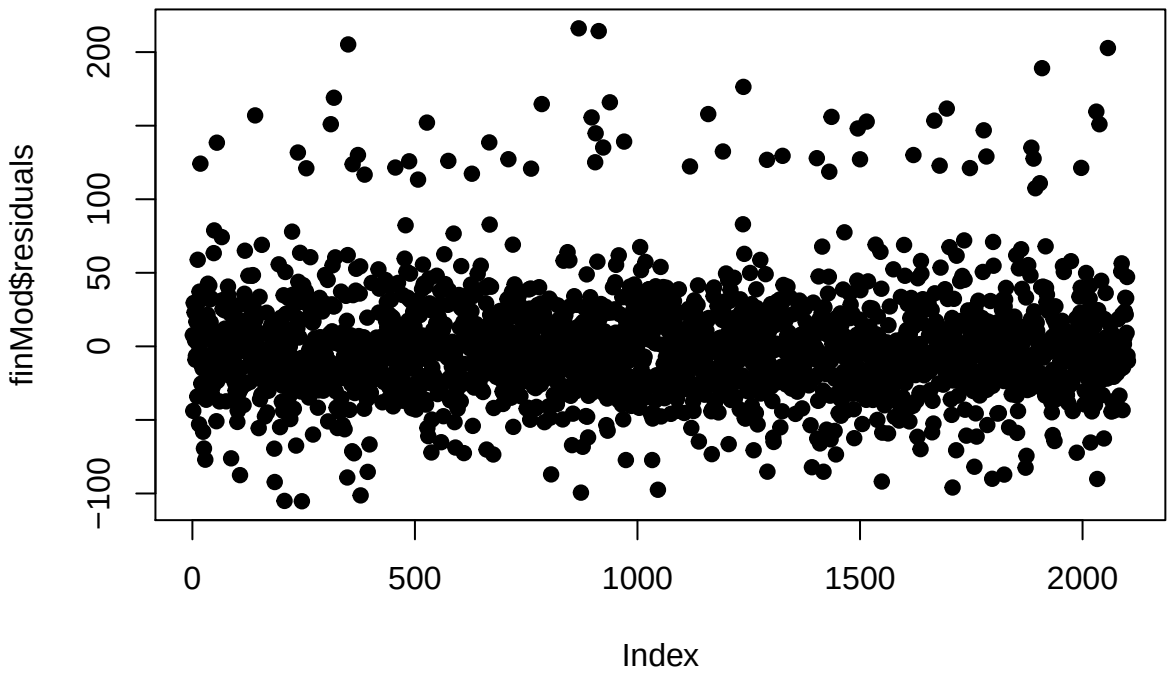
- We can plot by variables not included in the model to help identify further outliers

```
qplot(finMod$fitted, finMod$residuals, colour = race, data = training)
```



- Plotting by index can sometimes give insight to entry issues

```
plot(finMod$residuals, pch = 19)
```



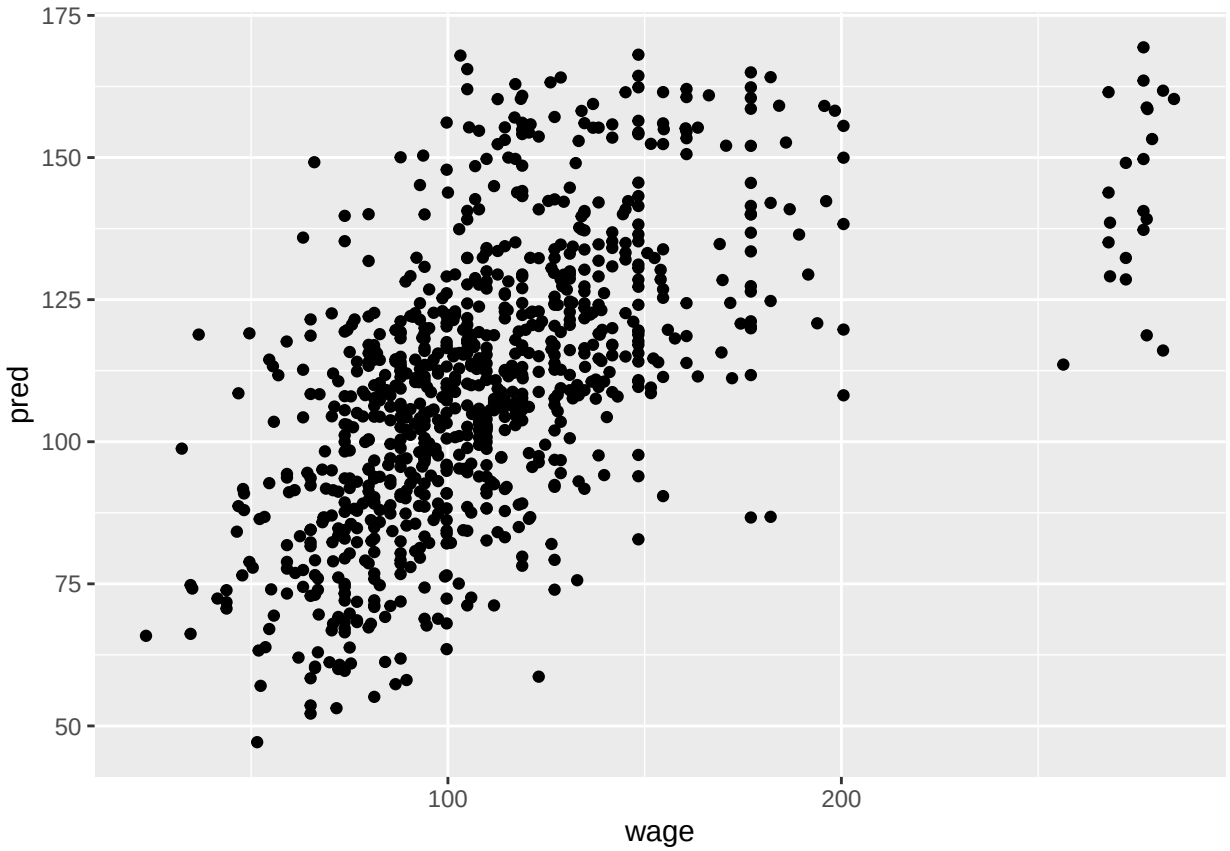
- If a trend was seen here it could suggest there is a varaible missing from the model.

Plotting Predicted Versus Truth in Test Set

```
pred <- predict(modFit, testing)
qplot(wage, pred, colour = year, data = testing)
```

* Ideally we'd have an identity line that our prediction matched the truth, obviously that typically won't be the case
* This is more used as a "post-mortum" tool on your analysis as adjusting after this would be like using the test to train.

```
#Using all covariates
modFitAll <- train(wage ~ ., data = training, method = "lm")
pred <- predict(modFitAll, testing)
qplot(wage, pred, data = testing)
```



* Useful if you don't want to do some sort of model selection in advance

Notes and Further Reading

- Regression models are often useful in combinaiton with other models.
- Further Reading:
  - **Elements of statistical learning**
  - **Modern applied statistics with S**
  - **Introduction to statistical learning**

Quiz 2

1. Load the Alzheimer's disease data:

```
library(AppliedPredictiveModeling)
data(AlzheimerDisease)
```

- What commands will create non-overlapping training and test sets with about 50% of the observations assigned to each?

```
adData = data.frame(diagnosis,predictors)
trainIndex = createDataPartition(diagnosis, p = 0.50,list=FALSE)
```

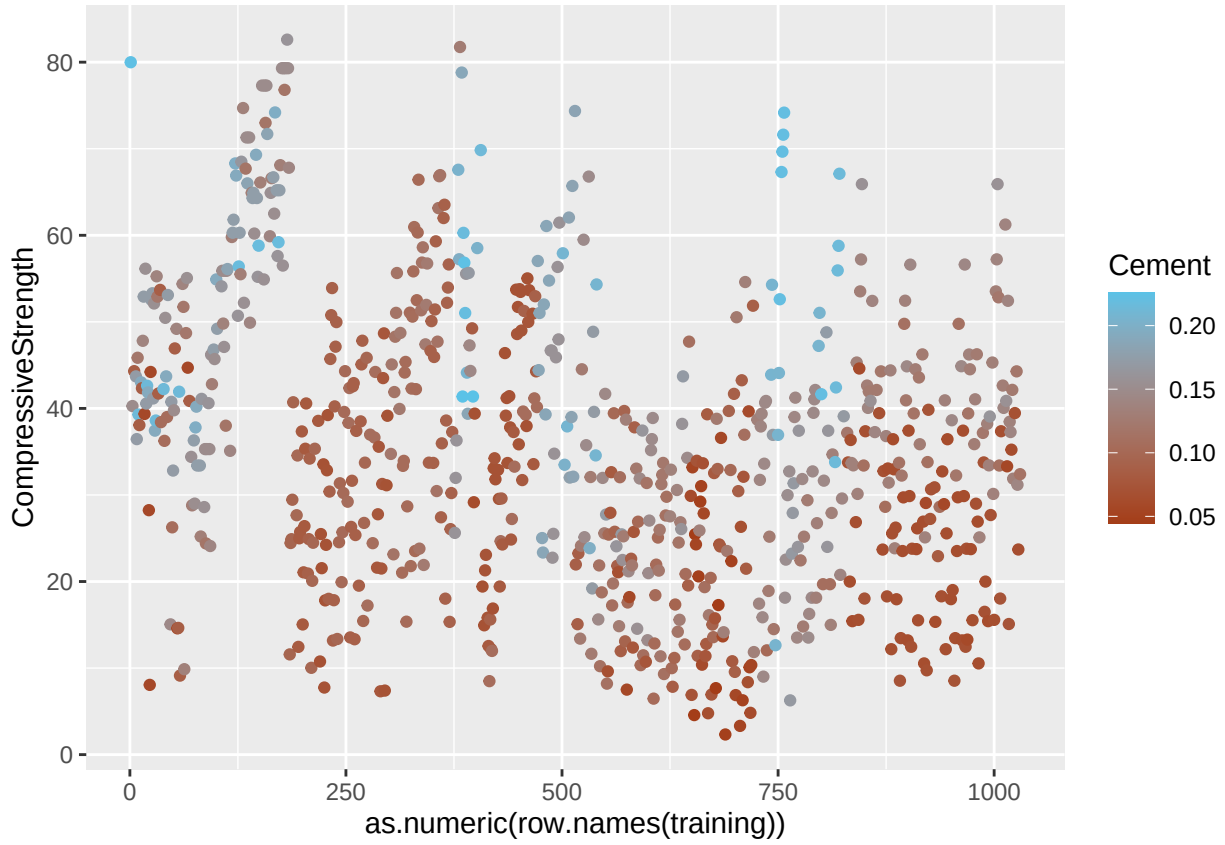
```
training = adData[trainIndex,]
testing = adData[-trainIndex,]
```

2. Load (and subset) the cement data with the following commands:

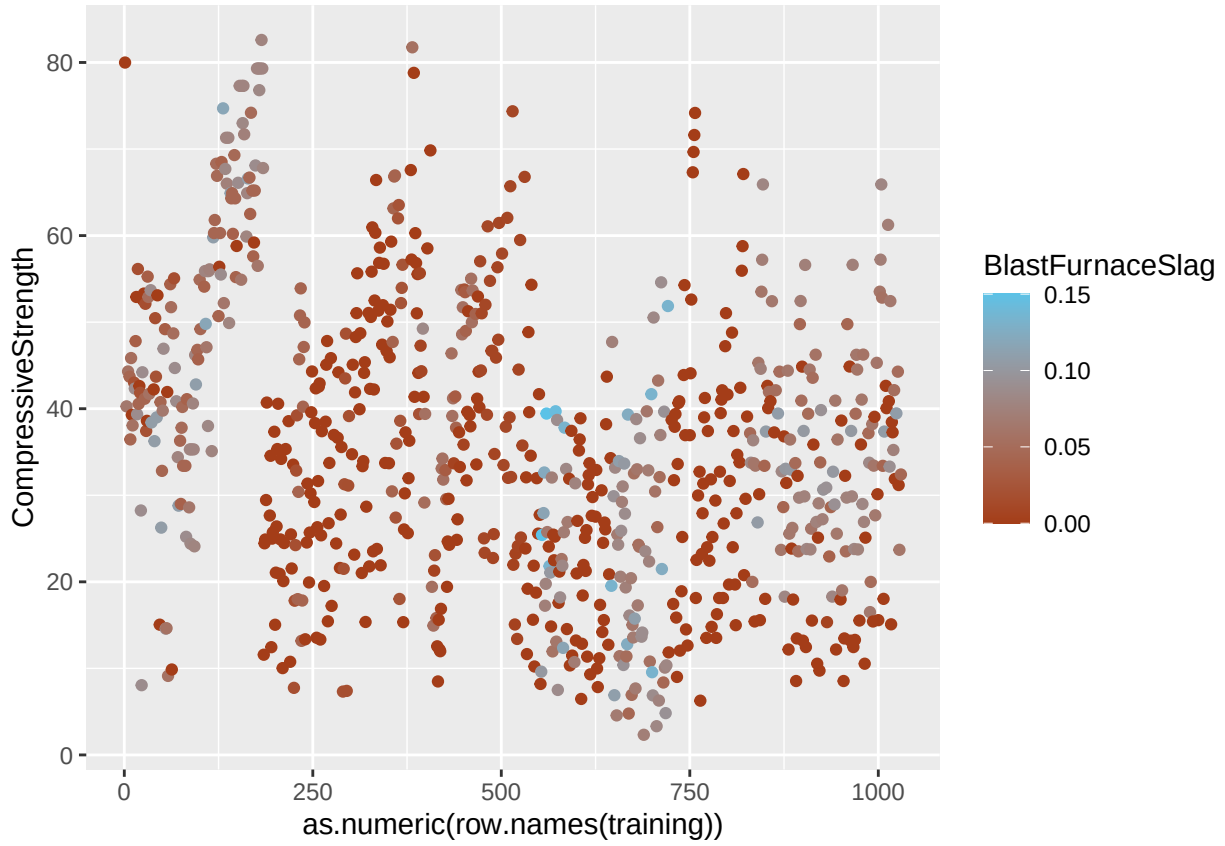
```
library(AppliedPredictiveModeling)
data(concrete)
library(caret)
set.seed(1000)
inTrain = createDataPartition(mixtures$CompressiveStrength, p = 3/4)[[1]]
training = mixtures[inTrain,]
testing = mixtures[-inTrain,]
```

Make a plot of the outcome (CompressiveStrength) versus the index of the samples. Color by each of the variables in the data set.

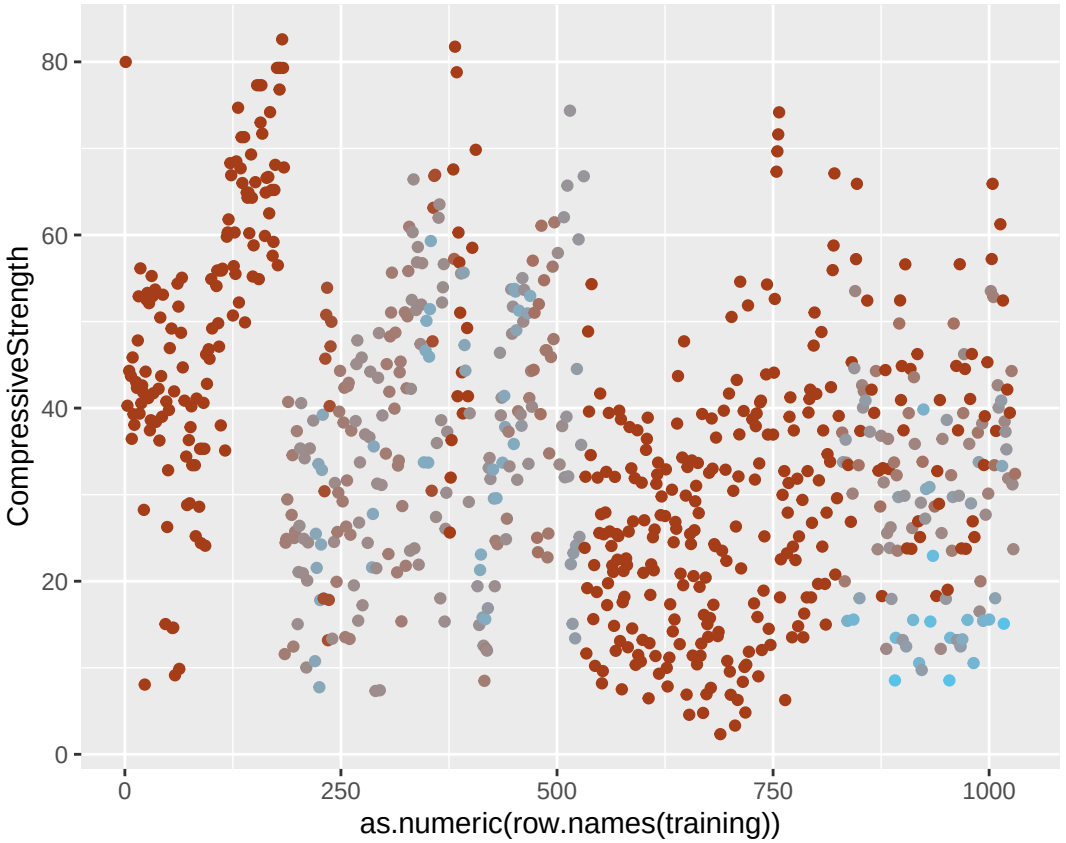
```
library(tidyverse)
library(RColorBrewer)
plot <- ggplot(training,
               aes(as.numeric(row.names(training)), CompressiveStrength)) +
  scale_color_gradient(low = "#A43D18", high = "#5BC2E7")
plot + geom_point(aes(col = Cement))
```



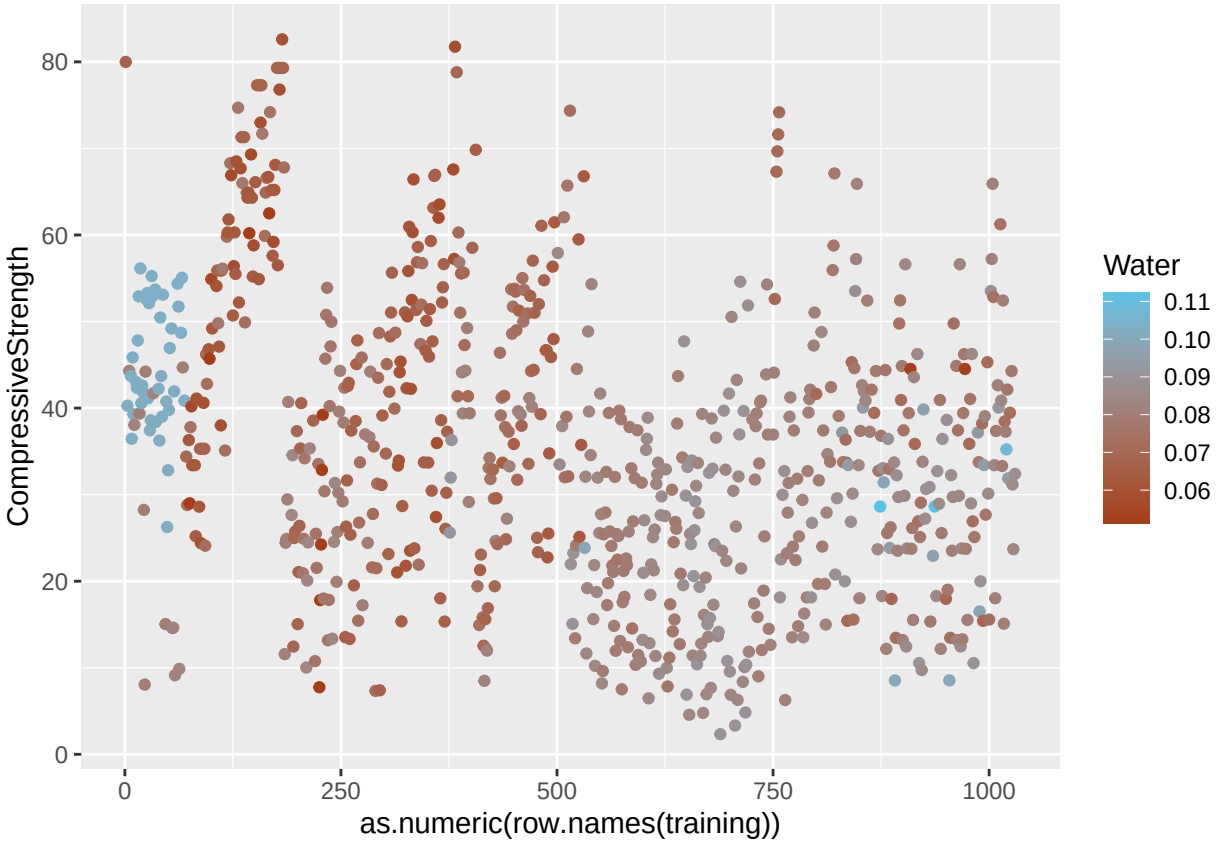
```
plot + geom_point(aes(col = BlastFurnaceSlag))
```



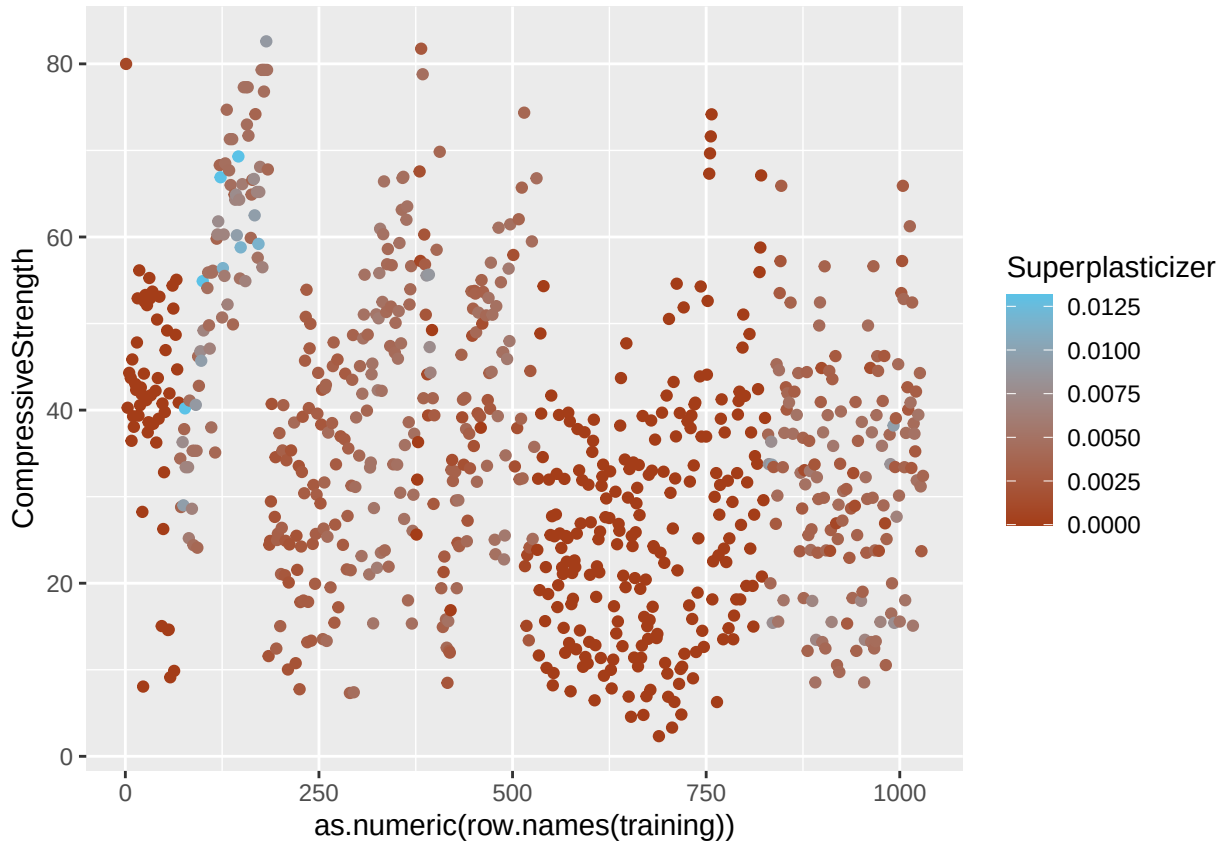

```
plot + geom_point(aes(col = FlyAsh))
```



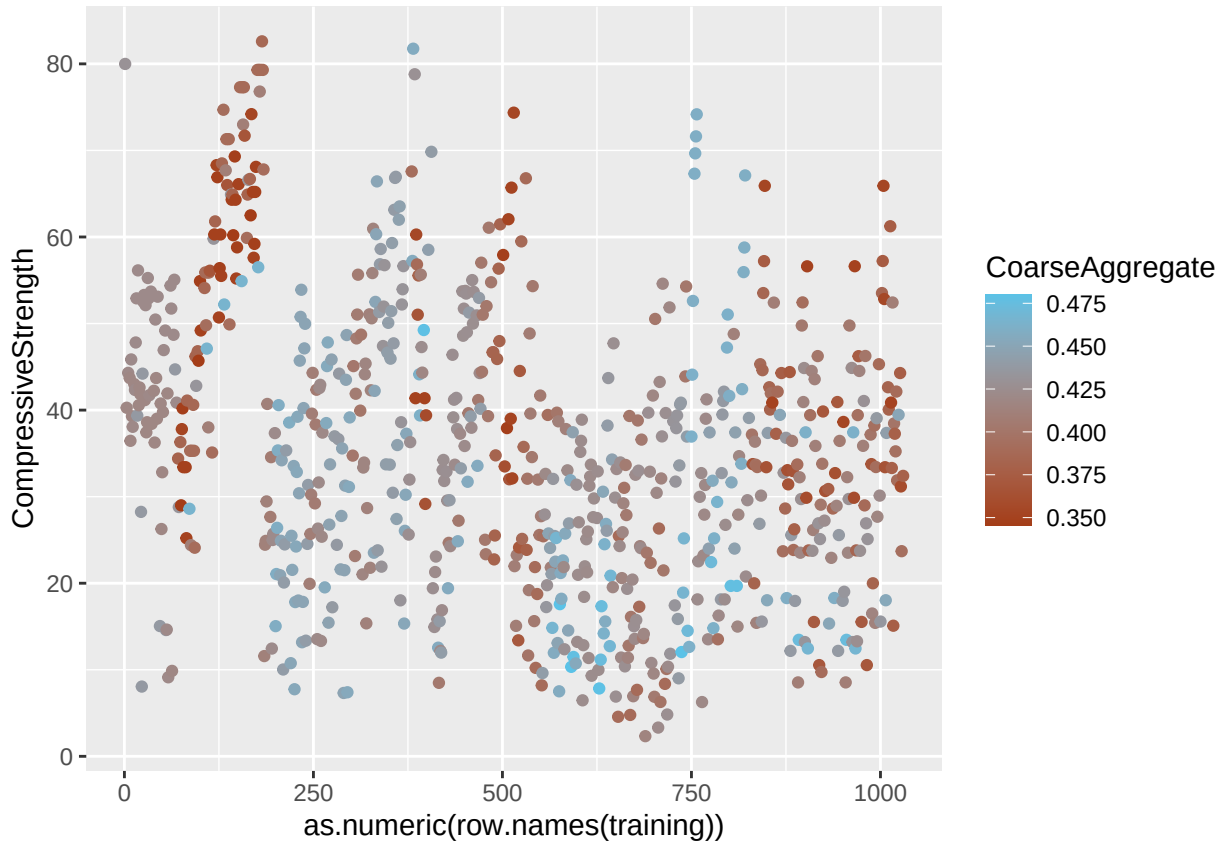

```
plot + geom_point(aes(col = Water))
```



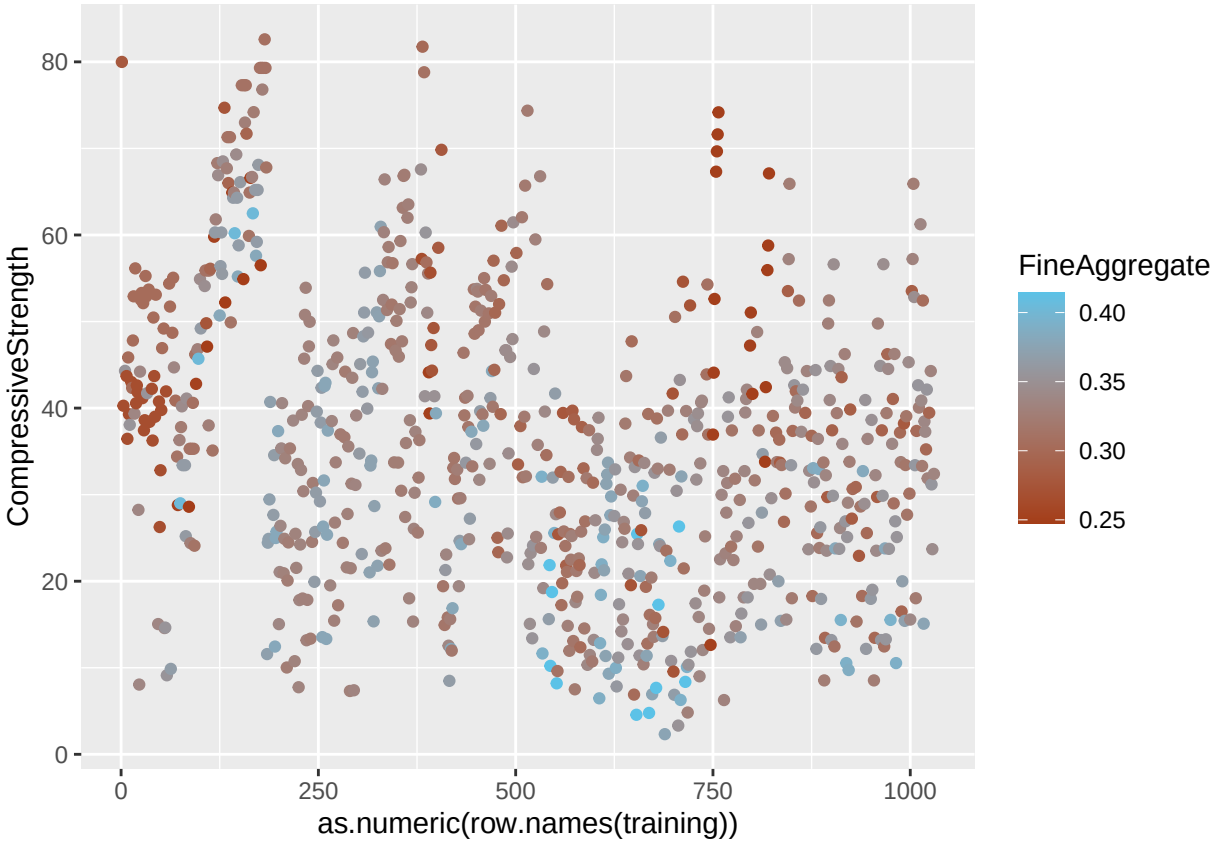

```
plot + geom_point(aes(col = Superplasticizer))
```



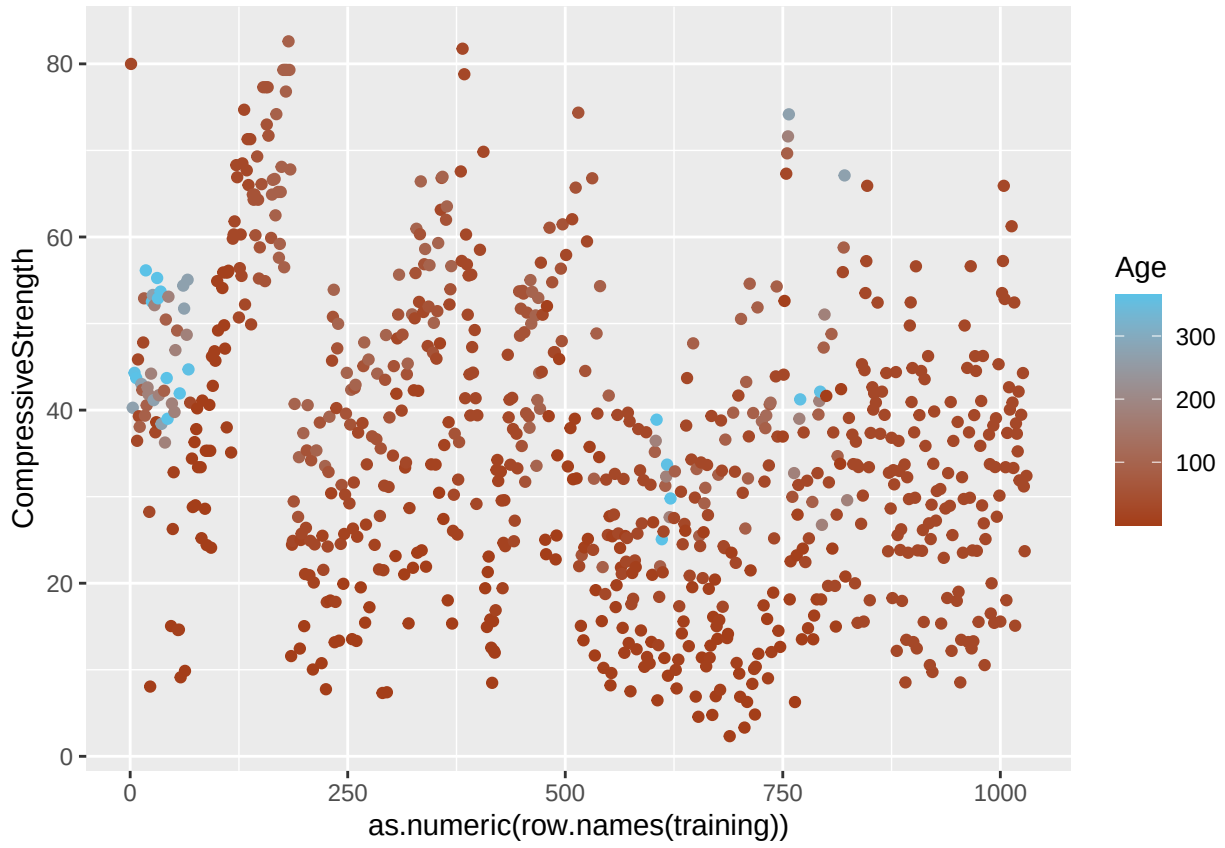

```
plot + geom_point(aes(col = CoarseAggregate))
```



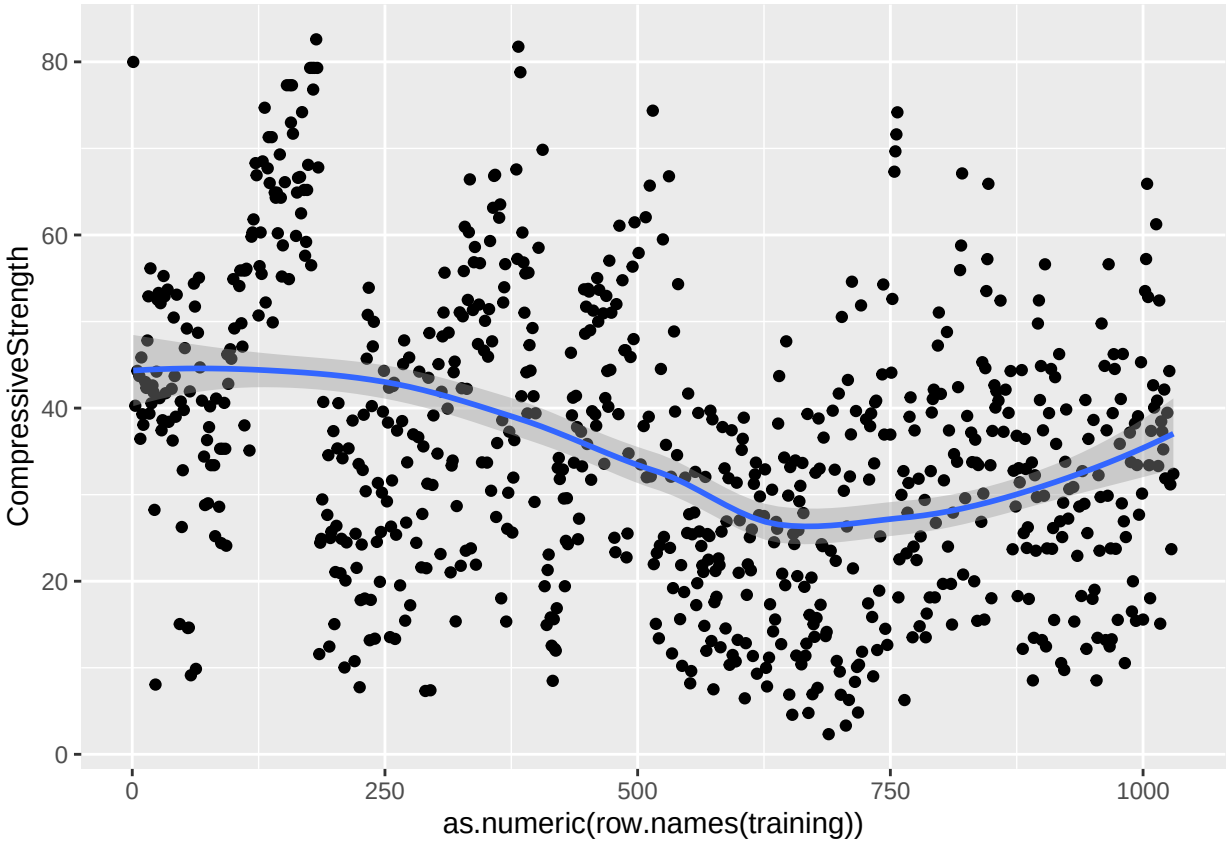

```
plot + geom_point(aes(col = FineAggregate))
```




```
plot + geom_point(aes(col = Age))
```



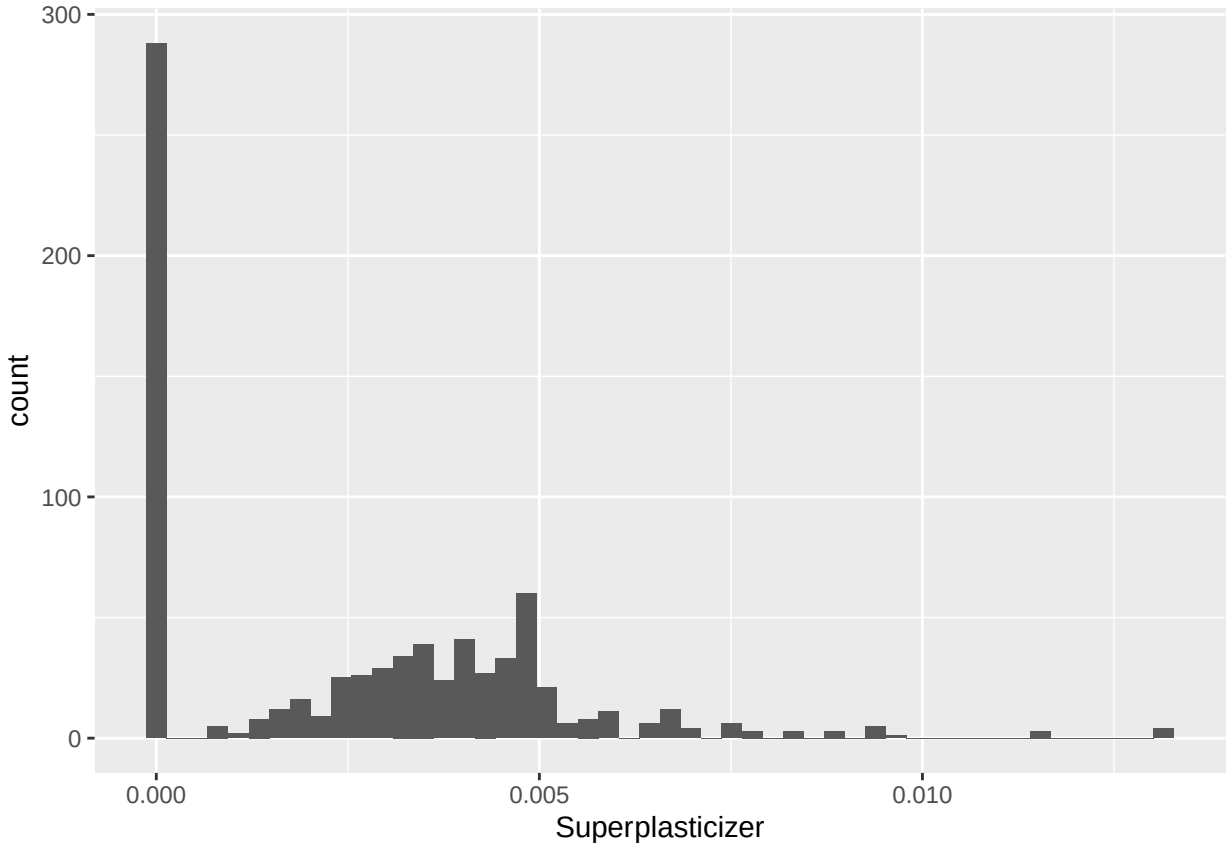

```
plot + geom_point() + geom_smooth()
```



What do you notice in these plots?
* There is a non-random pattern in the plot of the outcome versus index that does not appear to be perfectly explained by any predictor suggesting a variable may be missing.

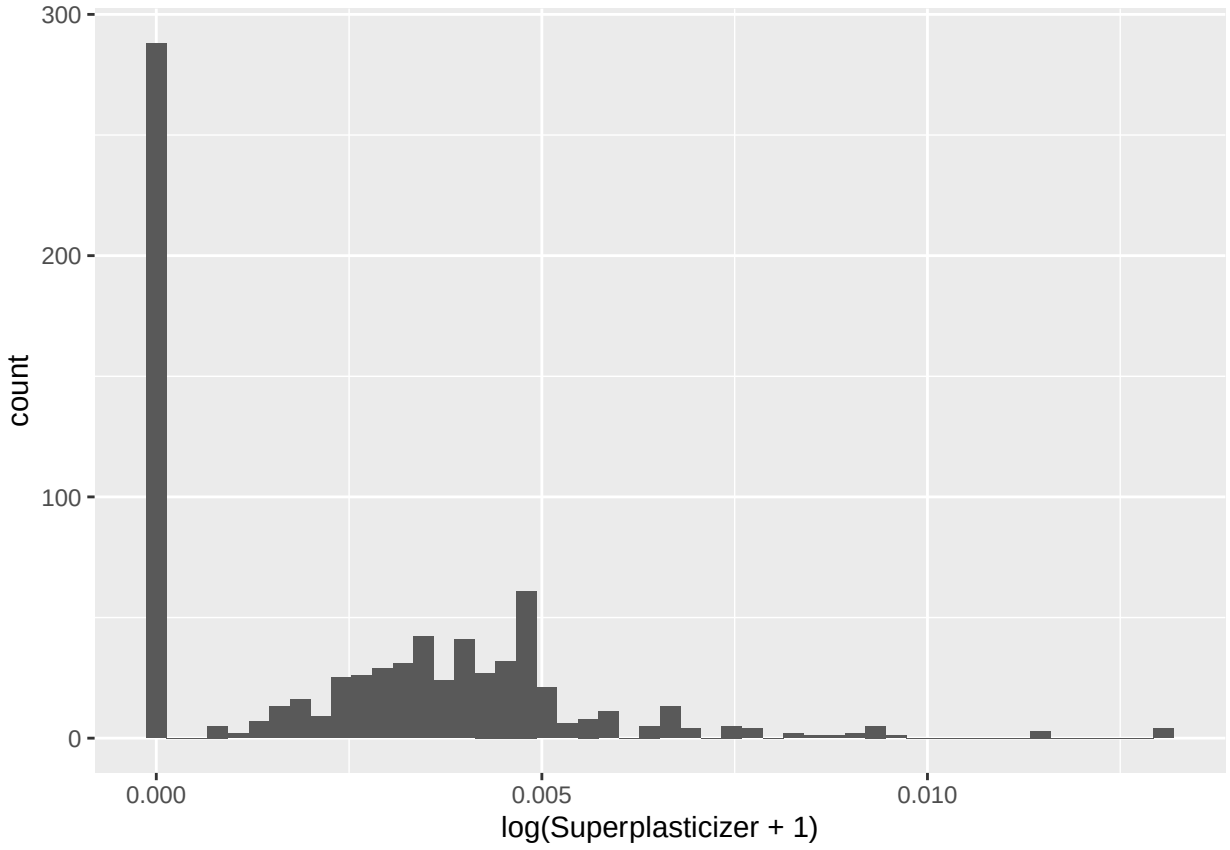
3. Make a histogram and confirm the `Superplasticizer` variable is skewed.

```
plot <- ggplot(training, aes(Superplasticizer)) +
  geom_histogram(bins = 50)
plot
```



Normally you might use the log transform to try to make the data more symmetric. Why would that be a poor choice for this variable?

```
plot <- ggplot(training, aes(log(Superplasticizer + 1))) +
  geom_histogram(bins = 50)
plot
```



- The log transform does not reduce the skewness of the non-zero values of `Superplasticizer`

4. Load (and subset) the Alzheimer's disease data:

```
library(caret)
library(AppliedPredictiveModeling)
set.seed(3433)
data(AlzheimerDisease)
adData = data.frame(diagnosis,predictors)
inTrain = createDataPartition(adData$diagnosis, p = 3/4)[[1]]
training = adData[ inTrain,]
testing = adData[-inTrain,]
```

Find all the predictor variables in the training set that begin with IL.

```
begIL <- grepl("^IL", colnames(training))
```

Perform principal componetns on these variables with the `preProcess()` function from the `caret` package. Calculate the number of principal components needed to capture 80% of the variance. How many are there?

```
ILset <- training[,begIL]
preProc <- preProcess(ILset, method = "pca", thresh = 0.8)
preProc
```

```
## Created from 251 samples and 12 variables
```

```
##
## Pre-processing:
##   - centered (12)
##   - ignored (0)
##   - principal component signal extraction (12)
##   - scaled (12)
##
## PCA needed 7 components to capture 80 percent of the variance
```

5. With the Alzheimer's training data set consisting of only the predictors with variable names beginning with IL and the diagnosis. Build two predictive models, one using the predictors as they are and one using PCA with principal components explaining 80% of the variance in the predictors. Use `method = "glm"` in the train function.

```
trainIL <- cbind(diagnosis = training$diagnosis, ILset)
preProc <- preProcess(trainIL, method = "pca", thresh = 0.8)
ILPC <- predict(preProc, trainIL)

NonPCA <- train(diagnosis ~., data = trainIL, method = "glm")
WithPCA <- train(diagnosis ~., data = trainIL, method = "glm",
                 preProcess = "pca", trControl = trainControl(
                   preProcOptions = list(thresh = 0.8)))
```

What is the accuracy of each method in the test set? Which is more accurate?

```
nopcres <- confusionMatrix(testing$diagnosis,
                           predict(NonPCA, testing))
pcres <- confusionMatrix(testing$diagnosis,
                         predict(WithPCA, testing))
round(nopcres$overall[1], 2)
```

```
## Accuracy
##     0.65
```

```
round(pcres$overall[1], 2)
```

```
## Accuracy
##     0.72
```

Predicting with Trees, Random Forests, & Model Based Predictions

Trees

Predicting with Trees

Key Ideas

- Iteratively split variables into groups
- Evaluate "homogeneity" of outcome within each group
- Split again if necessary until groups are homogeneous or small enough
  Pros:
- Easy to interpret
- Better performance in nonlinear settings
  Cons:
- Without pruning/cross-validation can lead to overfitting
- Harder to estimate uncertainty
- Results may be variable

Example Tree

- This tree appeared in the NY Times during the 2008 elections when Obama was running against Clinton for the democratic election
- The tree asks the most likely split that would result from a question first, then asked any further questions until all the counties were subsetted.

Basic Algorithm

1. Start with all variables in one group
2. Find the variable/split that best separates the outcomes
3. Divide the data into two groups ("leaves") on that split ("node")
4. Within each split, find the best variable/split that separates the outcomes
5. Continue until the groups are too small or sufficiently "pure"

Measures of Impurity

- Impurity has different measurements that are based on the following formula:
  $\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i in Leaf m} \mathbb{1}(y_i = k)$
- Where:
  - N_m is the number of objects in leaf m
  - $\sum_{x_i in Leaf m} \mathbb{1}(y_i = k)$ is the count of how many times that class k appears in leaf m

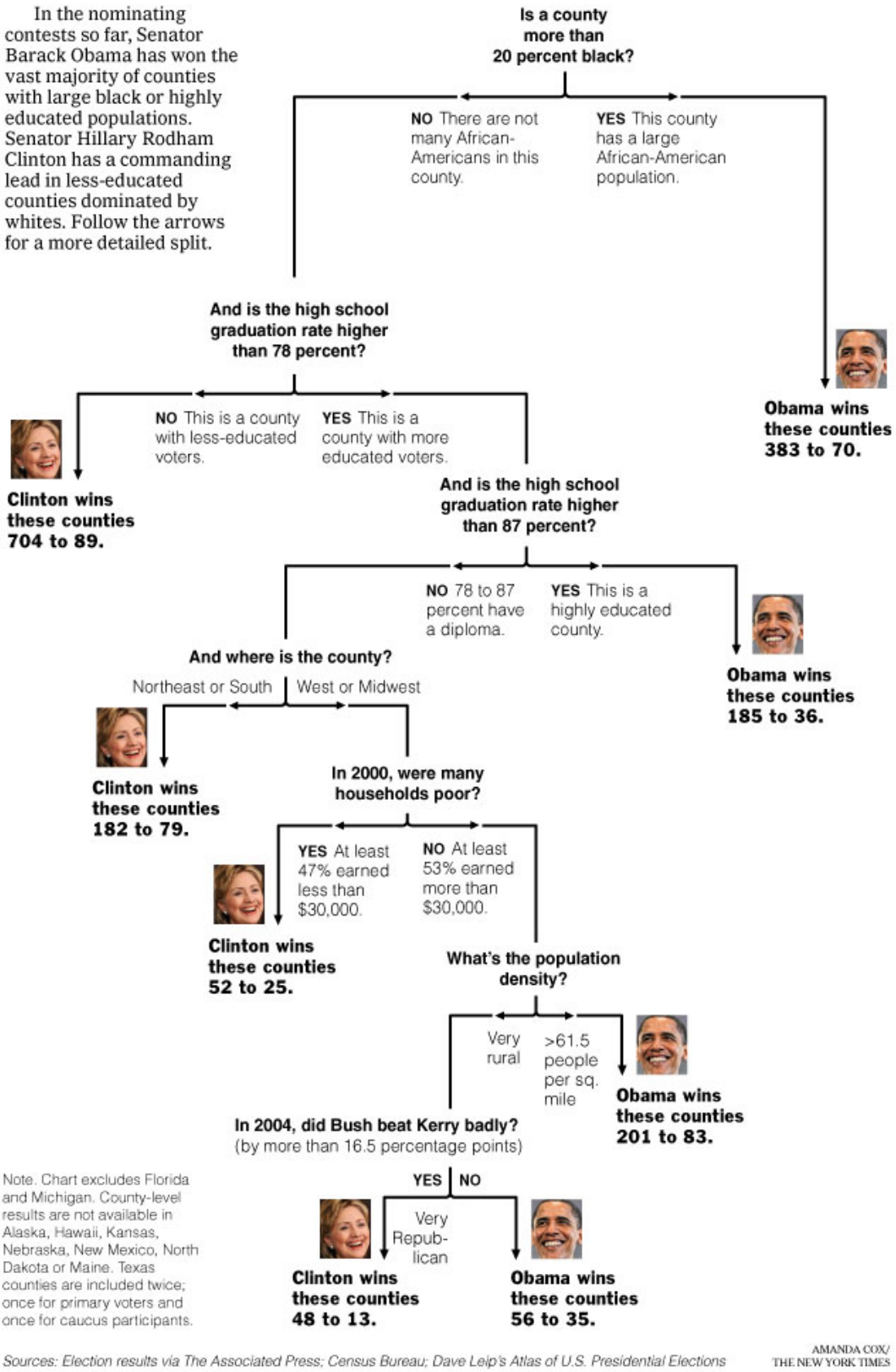


Figure 6: Decision Tree: The Obama-Clinton Divide

Misclassification Error:
$1 - \hat{p}_{mk(m)}$
* Where $k(m)$ is the most common k
* 0 = perfect purity
* 0.5 = no purity (perfectly balanced means no homogenity)

Gini Index:
$\sum_{k \neq k'} \hat{p}_{mk} \times \hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^{K} p_{mk}^2$
* Essentially 1 minus the sum of the squared probability that an object belongs to any of the different classes
* 0 = perfect purity
* 0.5 = no purity

Deviance/Information Gain:
$-\sum_{k=1}^{K} \hat{p}_{mk} \log_2 \hat{p}_{mk}$
* Deviance uses log base e, Information is as above
* 0 = perfect purity
* 1 = no purity

- **Wikipedia**

Example

- Say we have an object of 16 points, 15 blue & 1 red
  - This **Misclassification Error** would be $1 - \frac{15}{16} = 1/16 = 0.0625$
  - The **Gini Index** would be $1 - (\frac{1}{16}^2 + \frac{15}{16}^2) \approx 0.1172$
  - The **Information Gain** would be $-[\frac{1}{16} \times \log_2(\frac{1}{16}) + \frac{15}{16} \times \log_2(\frac{15}{16})] \approx 0.3373$
- Say we have an object of 16 points, 8 blue & 8 red
  - This **Misclassification Error** would be $1 - \frac{8}{16} = 8/16 = 0.5$
  - The **Gini Index** would be $1 - (\frac{8}{16}^2 + \frac{8}{16}^2) = 0.5$
  - The **Information Gain** would be $-[\frac{8}{16} \times \log_2(\frac{8}{16}) + \frac{8}{16} \times \log_2(\frac{8}{16})] = 1$

Example: Iris Data

```
data(iris)
library(tidyverse)
```

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
table(iris$Species) #What we're predicting
```

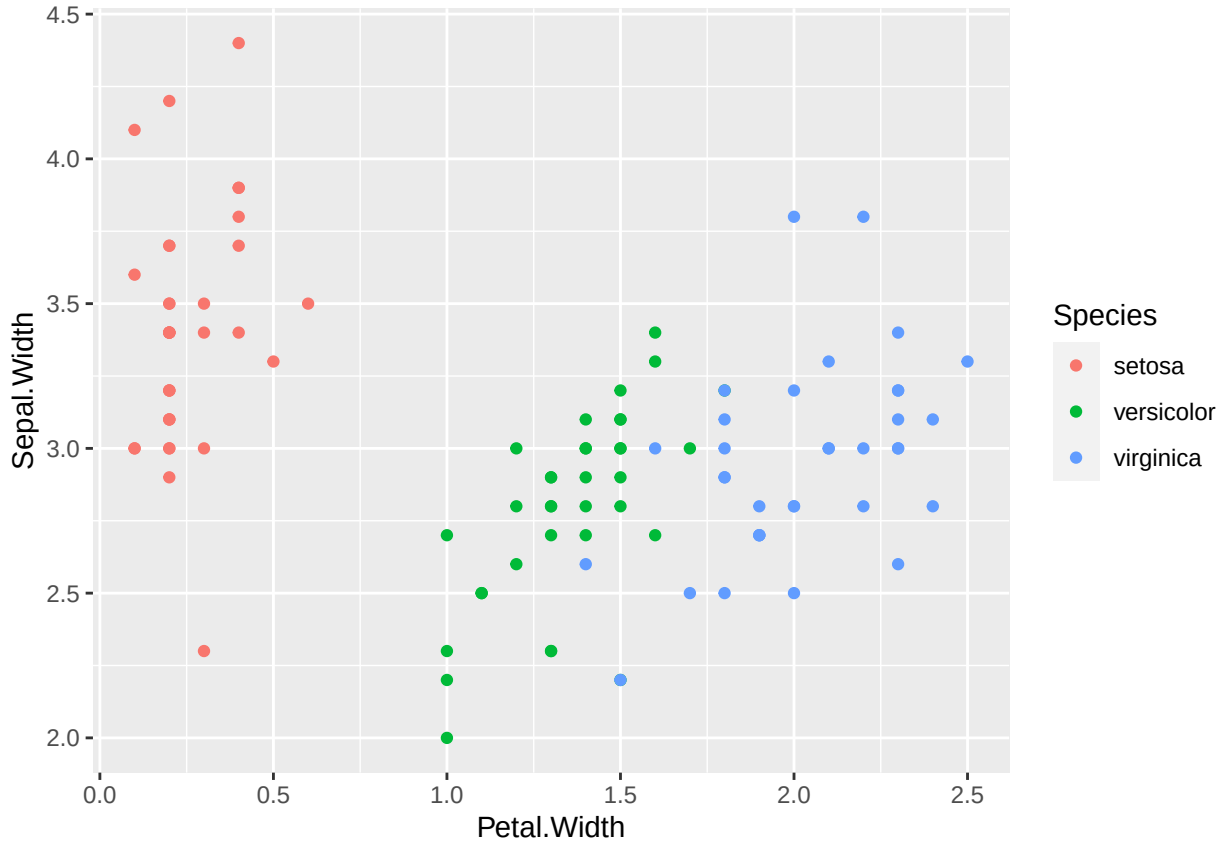
```
##
##     setosa versicolor  virginica
##         50         50         50
```

```
#Create training/test sets
set.seed(1618033)
inTrain <- createDataPartition(y = iris$Species,
                               p = 0.7, list = FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
dim(training); dim(testing)
```

```
## [1] 105   5
```

```
## [1] 45  5
```

```
#Plot Iris petal/sepal width
qplot(Petal.Width, Sepal.Width, colour = Species, data = training)
```



- It can be seen here that there is a distinct cluster for `setosa`, and a semi-distinct cluster for `versicolor` & `virginica`

```
#Training
library(caret)
modFit <- train(Species ~.,
                method = "rpart", #R package for reg. & classification trees
                data = training)
```

Viewing Model

```
print(modFit$finalModel)
```

```
## n= 105
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 105 70 setosa (0.33333333 0.33333333 0.33333333)
##   2) Petal.Length< 2.45 35  0 setosa (1.00000000 0.00000000 0.00000000) *
##   3) Petal.Length>=2.45 70 35 versicolor (0.00000000 0.50000000 0.50000000)
##     6) Petal.Length< 4.85 33  1 versicolor (0.00000000 0.96969697 0.03030303) *
##     7) Petal.Length>=4.85 37  3 virginica (0.00000000 0.08108108 0.91891892) *
```

- 1-7 gives a decision tree for determining the prediction.

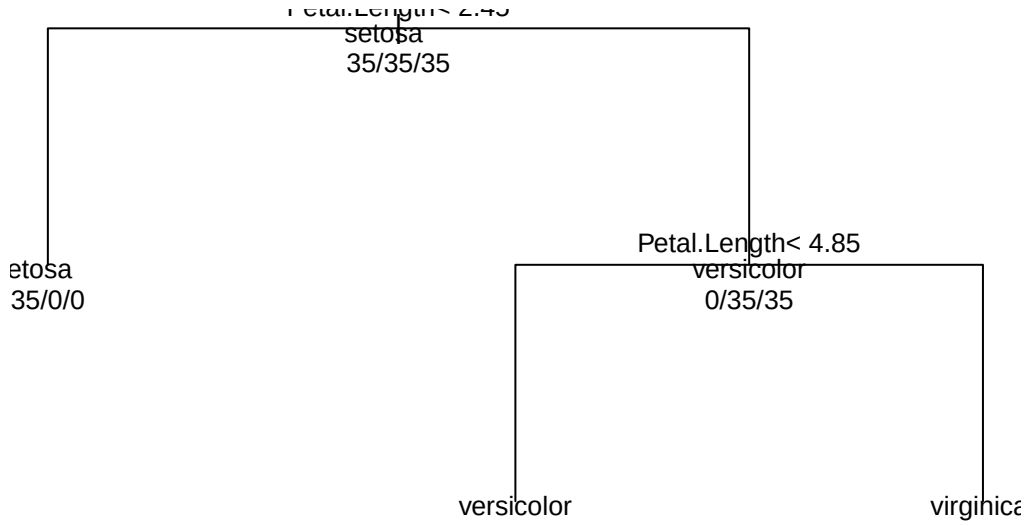
2) If `Petal.Length` is less than 2.45 all of these belong to `setosa`

3) If `Petal.Length` is greater than or equal to 2.45 it's a 50/50 split between the other two species

4) If `Petal.Length` is less than 4.85 there is about a 0.97 probability that specis is `versicolor`

Plotting Tree

```
#Dendrogram
plot(modFit$finalModel, uniform = TRUE, main = "Classification Tree")
text(modFit$finalModel, use.n = TRUE, all = TRUE, cex = 0.8)
```

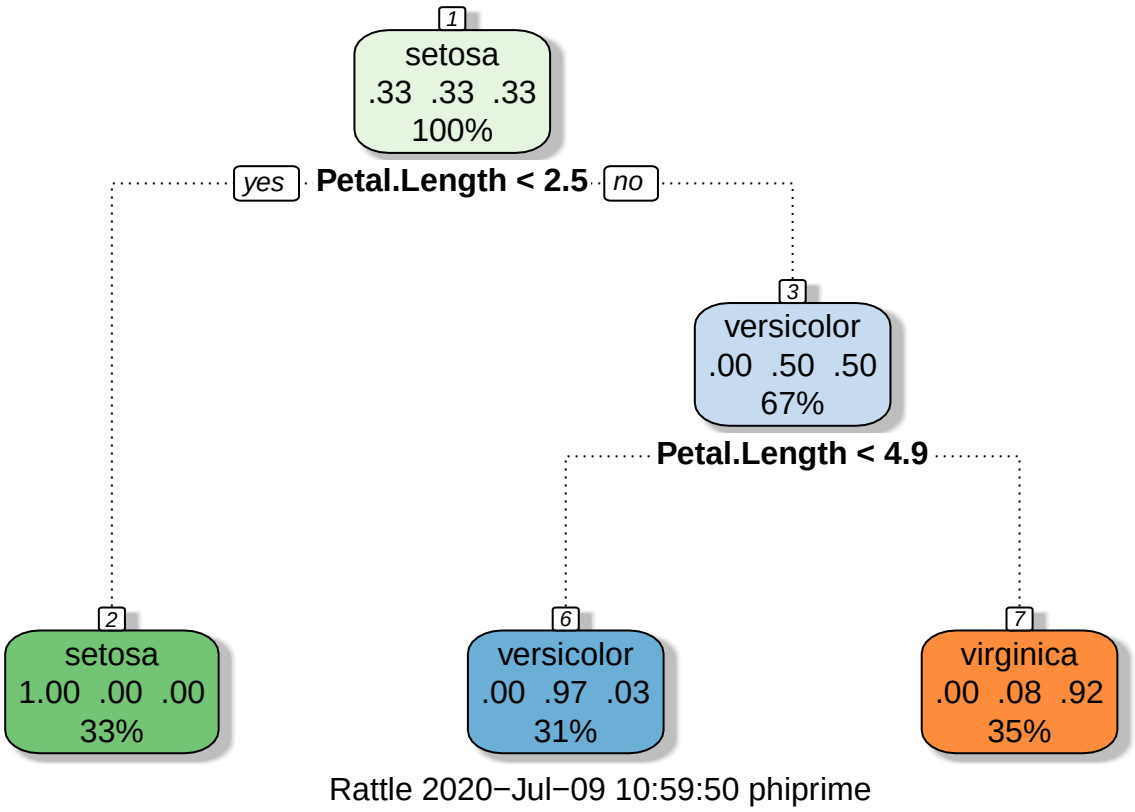



```
#Prettier version
library(rattle)
```

```
## Loading required package: bitops
```

```
## Rattle: A free graphical interface for data science with R.
## Version 5.4.0 Copyright (c) 2006-2020 Togaware Pty Ltd.
## Type 'rattle()' to shake, rattle, and roll your data.
```

```
fancyRpartPlot(modFit$finalModel)
```



Predicting New Values

```
#Gives factors since that's what it was predicting
predict(modFit, newdata = testing)
```

```
##  [1] setosa     setosa     setosa     setosa     setosa     setosa
##  [7] setosa     setosa     setosa     setosa     setosa     setosa
## [13] setosa     setosa     setosa     versicolor versicolor versicolor
## [19] versicolor virginica  versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
## [31] virginica  virginica  virginica  virginica  virginica  virginica
## [37] versicolor virginica  virginica  virginica  virginica  versicolor
## [43] virginica  virginica  virginica
## Levels: setosa versicolor virginica
```

Notes and Further Resources

- Classification trees are non-linear models
  - They use interactions between variables
  - Data transformations may be less important (monotone transformations, doesn't change order of values; will give same data splits)

  - Trees can also be used for regression problems (continuous outcome)

- Note that there are multiple tree building options in R, both in the caret package (`party`, `rpart`) and out of the caret package (`tree`)

- Further Resources
  - **Introduction to statistical learning**
  - **Elements of Statistical Learning**
  - **Classification and regression trees**

Bagging (Bootstrap Aggregating)

- When you fit complicated models, sometimes averaging those models together will give you a smoother model fit that gives you a better balance between potential bias & varaince

Key Idea

- Basic idea:
  1. Resample cases (with replacement) and recalculate predictions
  2. Average or majority vote

- Similar bias to fitting any of those models individually

- Reduced variance

- More useful for non-linear functions

Example: Ozone data

- Note: Ozone data was in the `ElemStatLearn` package which has been archived by CRAN, as such the following code chunk is needed to install the archived version of the package

```
url <- "http://cran.r-project.org/src/contrib/Archive/ElemStatLearn/ElemStatLearn_2015.6.26.ta
pkgFile <- "ElemStatLearn_2015.6.26.tar.gz"
download.file(url = url, destfile = pkgFile)
#There are no dependencies for ElemStatLearn
install.packages(pkgs = pkgFile, type = "source", repos = NULL)
```

```
library(ElemStatLearn)
```

```
##
## Attaching package: 'ElemStatLearn'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     spam
```

```r
data(ozone, package = "ElemStatLearn")
ozone <- ozone[order(ozone$ozone),]
head(ozone)
```

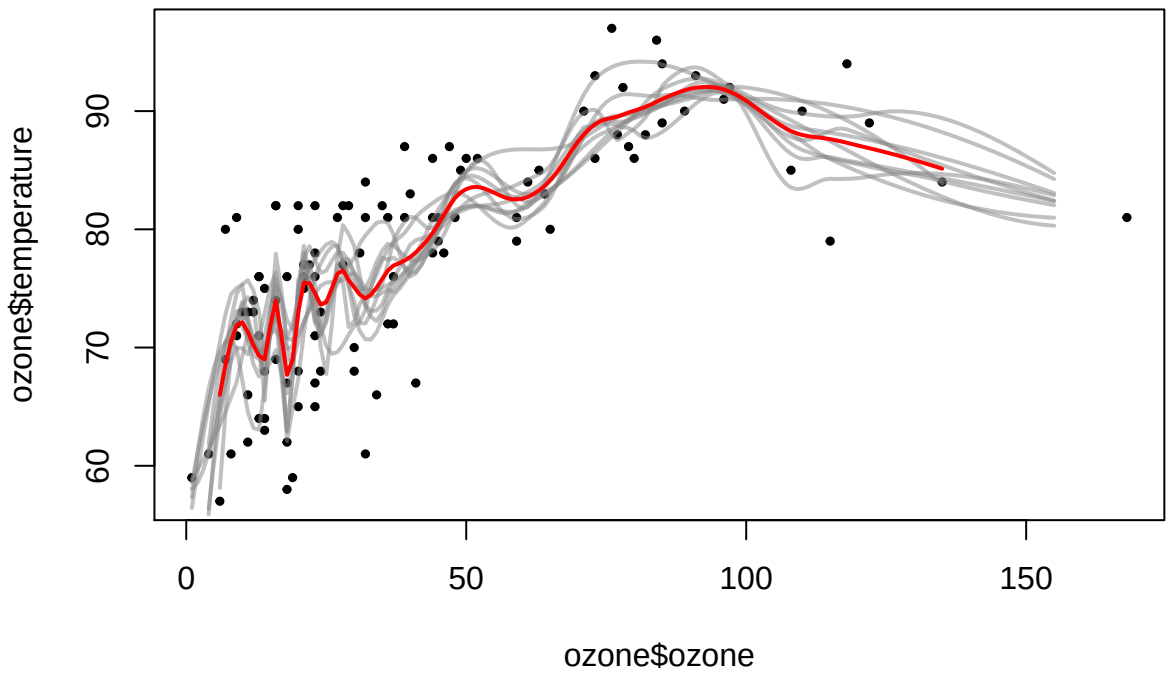
```
##     ozone radiation temperature wind
## 17      1         8          59  9.7
## 19      4        25          61  9.7
## 14      6        78          57 18.4
## 45      7        48          80 14.3
## 106     7        49          69 10.3
## 7       8        19          61 20.1
```

- We're going to try to predict `temperature` as a function of `ozone`

Bagged Loess

- We're going to sample with replacement from the entire data set
- Store & order it by `ozone`
- Create a loess (lo-ess) model
- Then predict 1:155 using the loess model and store it in our matrix, `ll`.
- We'l repeat this 10 times

```r
set.seed(1618033)
ll <- matrix(NA, nrow = 10, ncol = 155)
for (i in 1:10) {
  ss <- sample(1:dim(ozone)[1], replace = TRUE)
  ozone0 <- ozone[ss,]
  ozone0 <- ozone0[order(ozone0$ozone),]
  loess0 <- loess(temperature ~ ozone, data = ozone0, span = 0.2)
  ll[i,] <- predict(loess0, newdata = data.frame(ozone = 1:155))
}

plot(ozone$ozone, ozone$temperature, pch = 19, cex = 0.5)
for (i in 1:10) {lines(1:155, ll[i,], col = "#88888888", lwd = 2)}
lines(1:155, apply(ll, 2, mean), col = "#FF0000", lwd = 2)
```



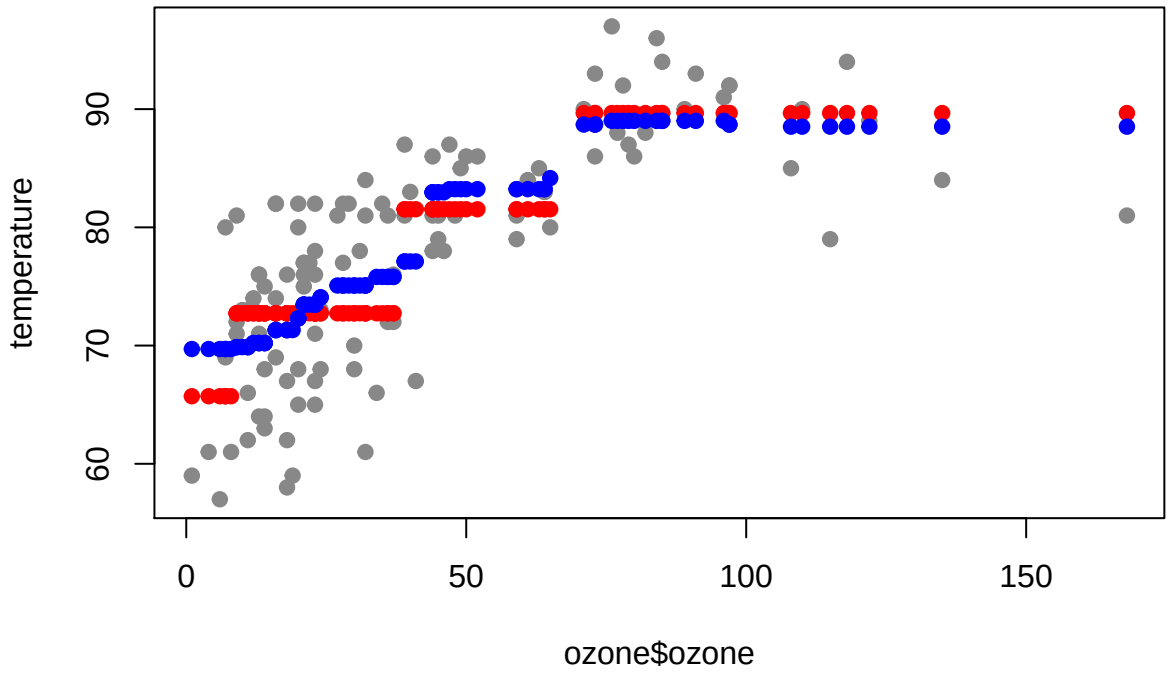
- Grey lines are the 10 fits with the resampled data set.
  - They capture a lot of the noise from the dataset but maybe a little *too* much
- The red line is the average of the 10 fits.

Bagging in caret

- Some models perform bagging for you, in the `train` function consider the `method` parameters:
  - `bagEarth`
  - `treebag`
  - `bagFDA`
- Alternatively you can bag any model you choose using the `bag` function
  - A bit of an advanced use so read the documentation if doing this

```
predictors = data.frame(ozone = ozone$ozone)
temperature = ozone$temperature
treebag <- bag(predictors, temperature, B = 10, #Number of replications
               bagControl = bagControl(#Informs how the model is to be fit
#Function to fit the model, could be 'train'
                        fit = ctreeBag$fit,
```

```
#How new values will be predicted
                        predict = ctreeBag$pred,
#How predictions will be put together, such as averaging
                        aggregate = ctreeBag$aggregate))

#Plotting custom Bagging
plot(ozone$ozone, temperature, col = "#888888", pch = 19)
points(ozone$ozone, predict(treebag$fits[[1]]$fit, predictors),
       pch = 19, col = "#FF0000")
points(ozone$ozone, predict(treebag, predictors), pch = 19, col = "#0000FF")
```



* Red values is fit from first regression tree
* Blue is the average of the 10 models

Parts of Bagging

```
ctreeBag$fit
```

```
## function (x, y, ...)
## {
##     loadNamespace("party")
##     data <- as.data.frame(x, stringsAsFactors = TRUE)
##     data$y <- y
##     party::ctree(y ~ ., data = data)
```

```
## }
## <bytecode: 0x55f5eead8ea8>
## <environment: namespace:caret>
```

- returns the `ctree` function

```
ctreeBag$pred
```

```
## function (object, x)
## {
##     if (!is.data.frame(x))
##         x <- as.data.frame(x, stringsAsFactors = TRUE)
##     obsLevels <- levels(object@data@get("response")[, 1])
##     if (!is.null(obsLevels)) {
##         rawProbs <- party::treeresponse(object, x)
##         probMatrix <- matrix(unlist(rawProbs), ncol = length(obsLevels),
##             byrow = TRUE)
##         out <- data.frame(probMatrix)
##         colnames(out) <- obsLevels
##         rownames(out) <- NULL
##     }
##     else out <- unlist(party::treeresponse(object, x))
##     out
## }
## <bytecode: 0x55f5eead9928>
## <environment: namespace:caret>
```

- creates `rawProbs` and uses that to make `probMatrix`, it then returns the observed levels or the response, depending on what's applicable for the object

```
ctreeBag$aggregate
```

```
## function (x, type = "class")
## {
##     if (is.matrix(x[[1]]) | is.data.frame(x[[1]])) {
##         pooled <- x[[1]] & NA
##         classes <- colnames(pooled)
##         for (i in 1:ncol(pooled)) {
##             tmp <- lapply(x, function(y, col) y[, col], col = i)
##             tmp <- do.call("rbind", tmp)
##             pooled[, i] <- apply(tmp, 2, median)
##         }
##         if (type == "class") {
##             out <- factor(classes[apply(pooled, 1, which.max)],
##                 levels = classes)
##         }
##         else out <- as.data.frame(pooled, stringsAsFactors = TRUE)
##     }
##     else {
##         x <- matrix(unlist(x), ncol = length(x))
```

```
##         out <- apply(x, 1, median)
##     }
##     out
## }
## <bytecode: 0x55f5eeac78b8>
## <environment: namespace:caret>
```

- takes predictions and puts them together (aggregates) in some way. Taking the median by default

Notes and Further Resources

- Bagging is most useful for nonlinear models
- Often used with trees - an extension is random forests
- Several models use bagging in `caret`'s `train` function
- Further resources:
  - **Bagging**
  - **PDF on Bagging and boosting**
- **Elements of Statistical Learning**

Random Forests

Random Forests

1. Bootstrap sample
2. At each split, bootstrap variables

- As such only a subset of variables are considered at each split

3. Grow multiple trees and vote
   Pros:

- Accuracy
  Cons:
- Speed
- Interpretability
- Overfitting

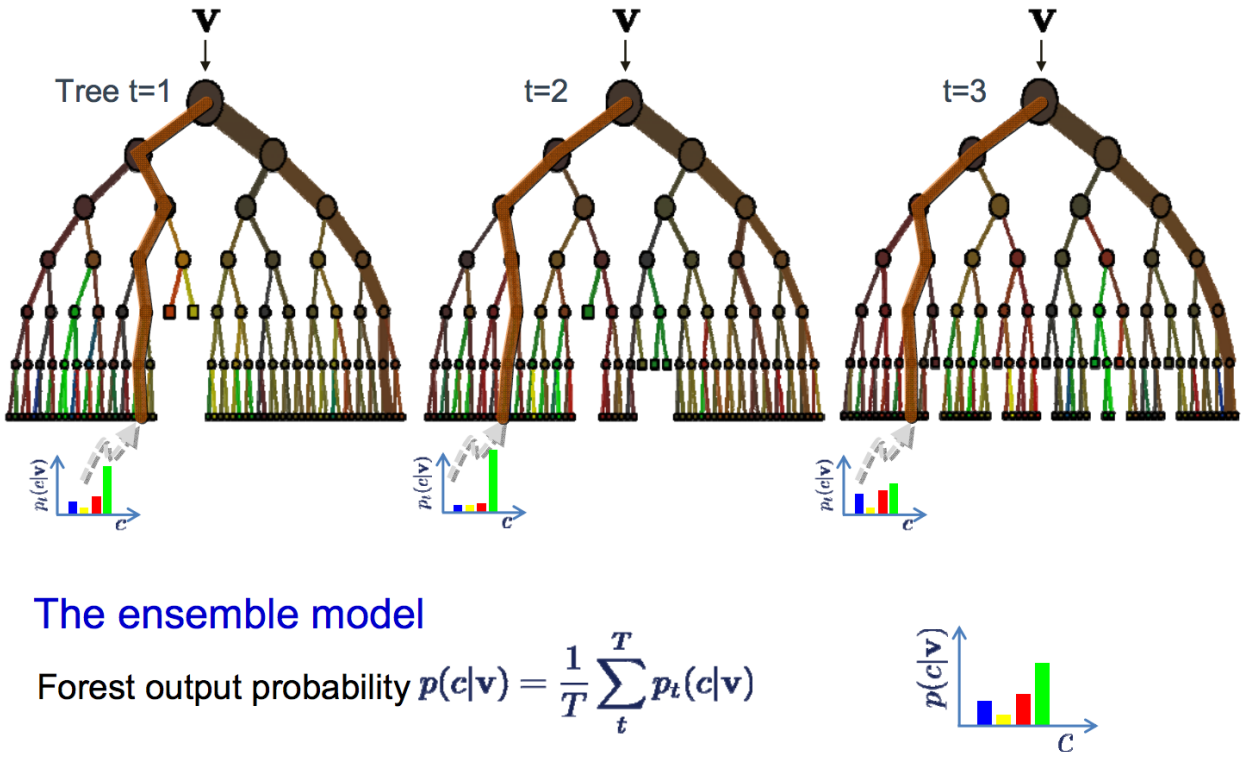


Figure 7: Visual Example

- Each **v** is based on a bootstrap sample, as is the sample taken at each node
- We then get an observation (thicker brown line) and see what leaf it ends on in each tree, which contains probabilities of each value.
- We then get these probabilities and average them together
  $p(c|v) = \frac{1}{T}\sum_t^T p_t(c|v)$
- Where T is the total number of trees

Example: Iris Data

```
data(iris)
library(tidyverse)
library(caret)
set.seed(1618)
inTrain <- createDataPartition(y = iris$Species,
                               p = 0.7, list = FALSE)
training <- iris[inTrain, ]
testing <- iris[-inTrain, ]
```

```
modFit <- train(Species ~., data = training,
                method = "rf", #random forests
```

```
                prox = TRUE) #Produces extra info for building models
modFit
```

```
## Random Forest
##
## 105 samples
##   4 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 105, 105, 105, 105, 105, 105, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##   2     0.9380351  0.9053613
##   3     0.9373653  0.9043924
##   4     0.9383177  0.9058721
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 4.
```

- `mtry` is the number of tries, or repeated trees that it builds

Getting a Single Tree

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:rattle':
##
##     importance
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
getTree(modFit$finalModel, k = 2)#2nd tree
```

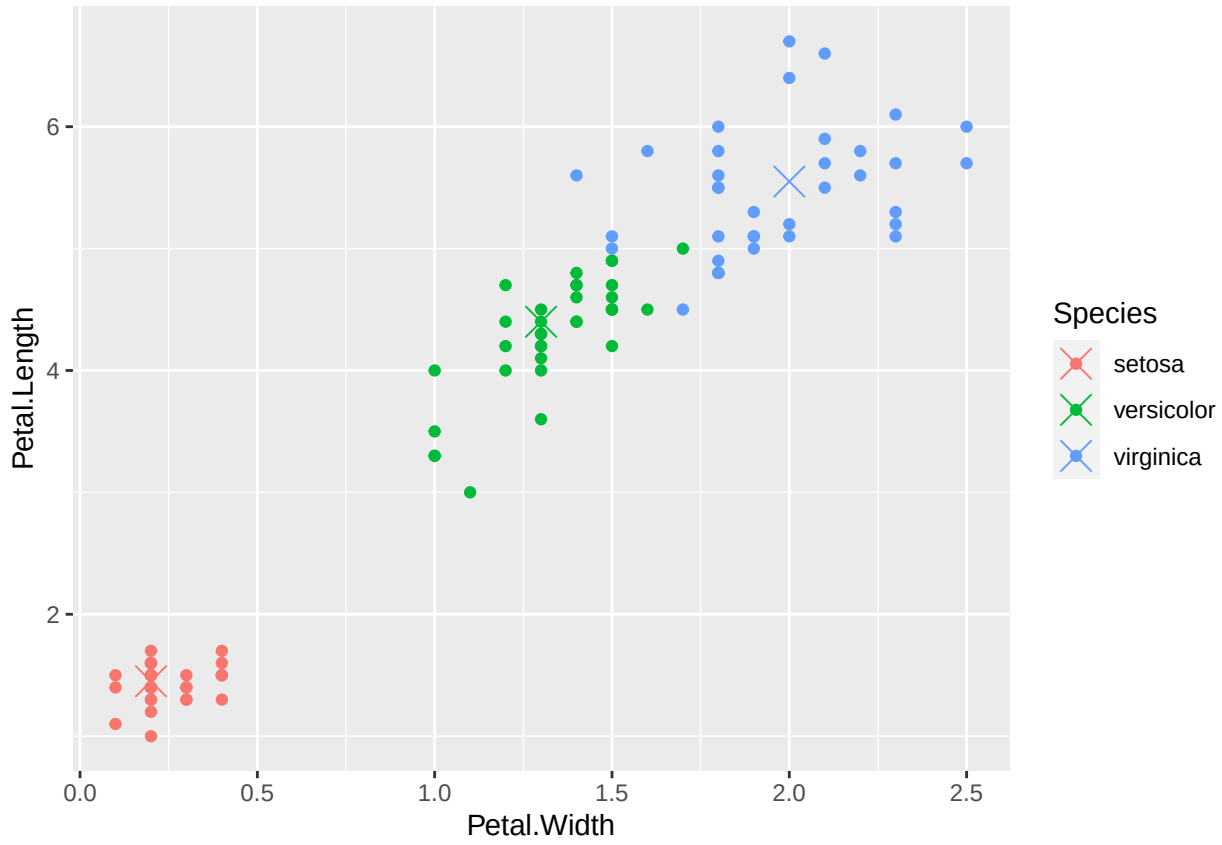
```
##    left daughter right daughter split var split point status prediction
## 1              2              3         3        2.35      1          0
```

```
## 2              0              0         0        0.00     -1          1
## 3              4              5         3        5.05      1          0
## 4              6              7         4        1.65      1          0
## 5              0              0         0        0.00     -1          3
## 6              8              9         3        4.95      1          0
## 7             10             11         2        2.90      1          0
## 8              0              0         0        0.00     -1          2
## 9              0              0         0        0.00     -1          3
## 10             0              0         0        0.00     -1          3
## 11             0              0         0        0.00     -1          2
```

- Each row corresponds to a particular split.
  - `left/right daughter` indicates what row is the next node
  - `split var` is what variable is being used to determine the split
  - `split point` is the value we're examining for the split (less than this value goes to the left, greater to the right)
  - `status` is a 1 if this node splits, and -1 otherwise
  - `prediction` is what it predicts the value to be if we're at a leaf (`status == -1`)

Viewing Class "Centers"

```
irisP <- classCenter(training[,c(3,4)], training$Species,
                     modFit$finalModel$prox)#We have this from `prox = TRUE`
irisP <- as.data.frame(irisP)
irisP$Species <- rownames(irisP)

plot <- qplot(Petal.Width, Petal.Length, col = Species, data = training)
plot + geom_point(aes(x = Petal.Width, y = Petal.Length, col = Species),
                  size = 5, shape = 4, data = irisP)
```

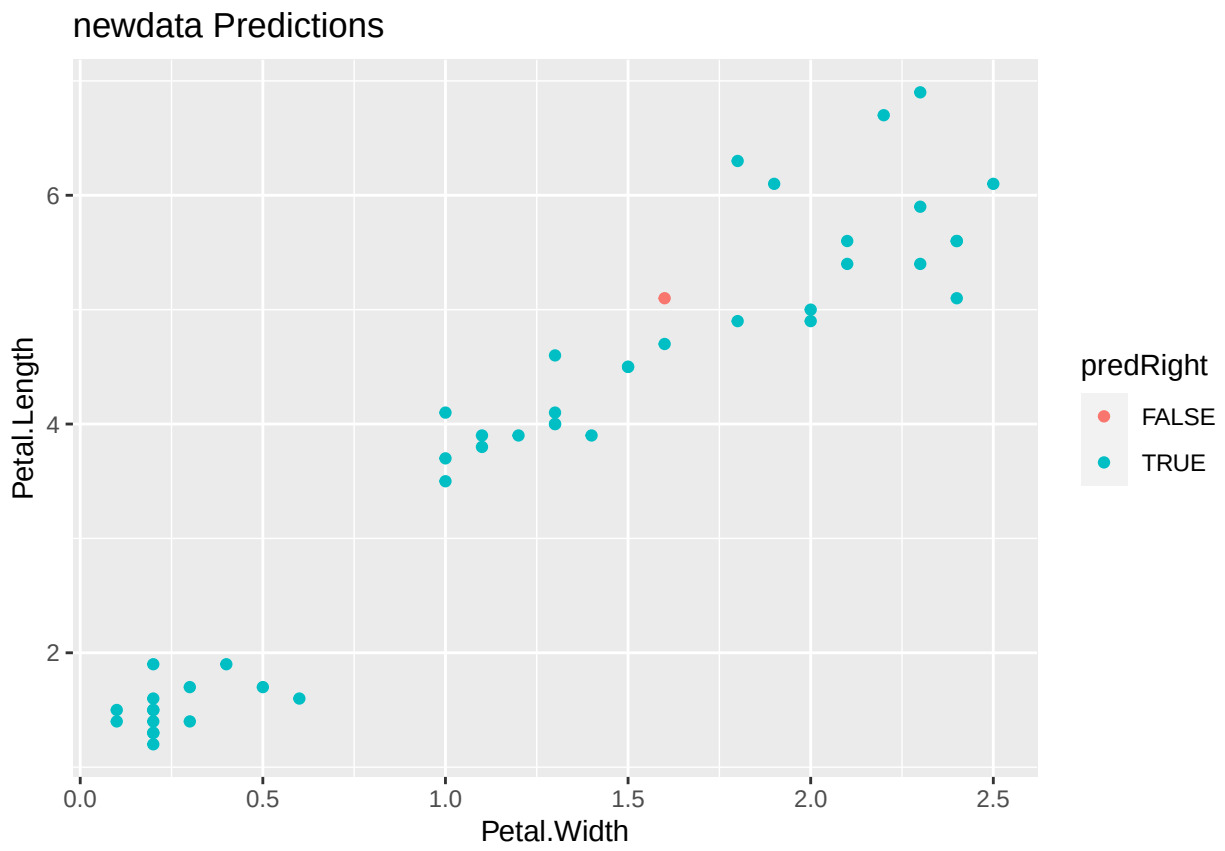


Predicting New Values

```
pred <- predict(modFit, testing)
testing$predRight <- pred == testing$Species
table(pred, testing$Species)
```

```
##
## pred         setosa versicolor virginica
##   setosa         15          0         0
##   versicolor      0         14         0
##   virginica       0          1        15
```

```
#Viewing which values we missed
qplot(Petal.Width, Petal.Length, colour = predRight,
      data = testing, main = "newdata Predictions")
```



Notes & Further Resources

- Random forests are usually one of the two top performing algorithms along with boosting in prediction contests.
- Random forests are difficult to interpret but often very accurate.
- Care should be taken to avoid overfitting (see **rfcv (PDF)** function)

Further Resources:
+ **Random forests (written by discoverer)**
+ **Random forest Wikipedia**
+ **Elements of Statistical Learning**

Boosting

- With *random forests* this is one of the most accurate out-of-the-box classifiers one can use.

Basic Idea

- General

1. Take lots of (possibly) weak predictors
2. Weight them and add them up
3. Get a stronger predictor

- Mathematical

1) Start with a set of classifiers, $h_1, ..., h_T$

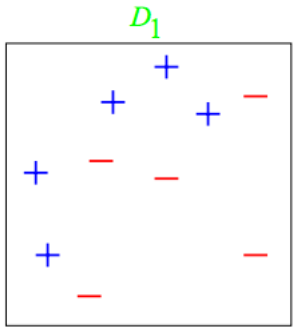
- Examples: All possible trees, all possible regression models, all possible cutoffs

2) Create a classifier that combines classification functions: $f(x) = sgn(\sum_{t=1}^{T} \alpha_t h_t(x))$

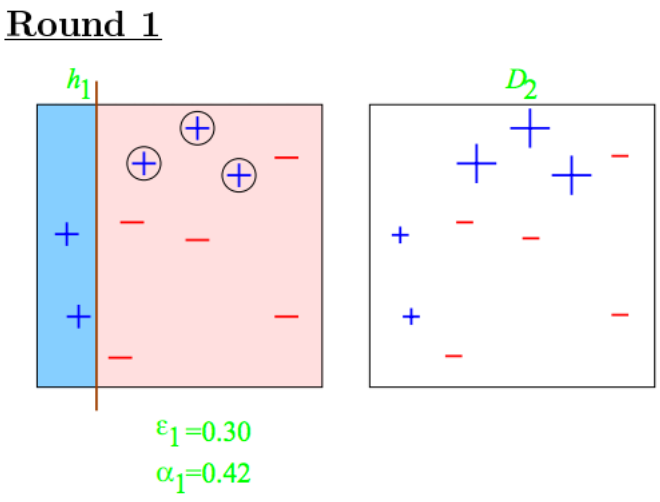
- Where:
  - α_t is the weight
  - $h_t(x)$ is the classifier
  - sgn is the **sign function**, essentially a unit step function centered at 0, and if $x = 0$, $sgn(x) = 0$
- Goal is to minimize error (on training set)
- Iterative, select one h at each step
- Calculate weights based on errors
- Upweight missed classifications and select next h
- **Adaboost on Wikipedia**
- **Boosting Tutorial PDF**

Visual Example

- Given a set of points, D_1, plotted by two variables

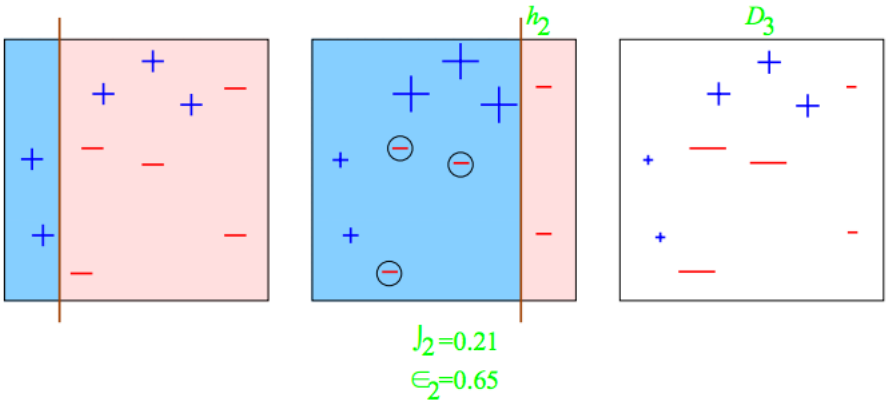


- We build a simple classifier to seperate the points

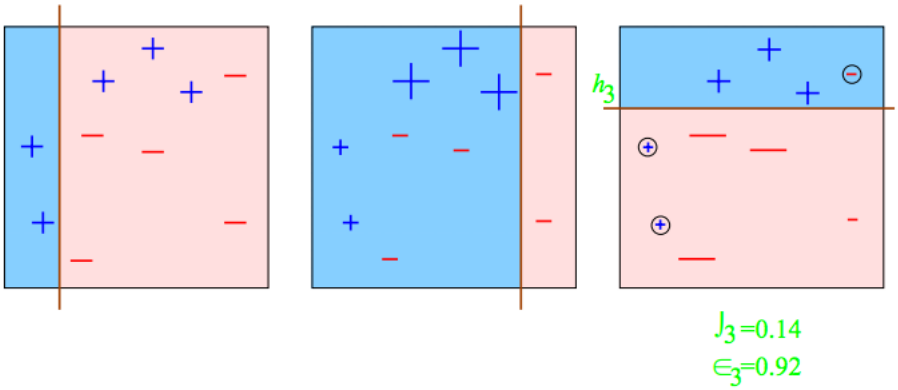


- We then up-weight the misclassified points and build the next classifier

Round 2

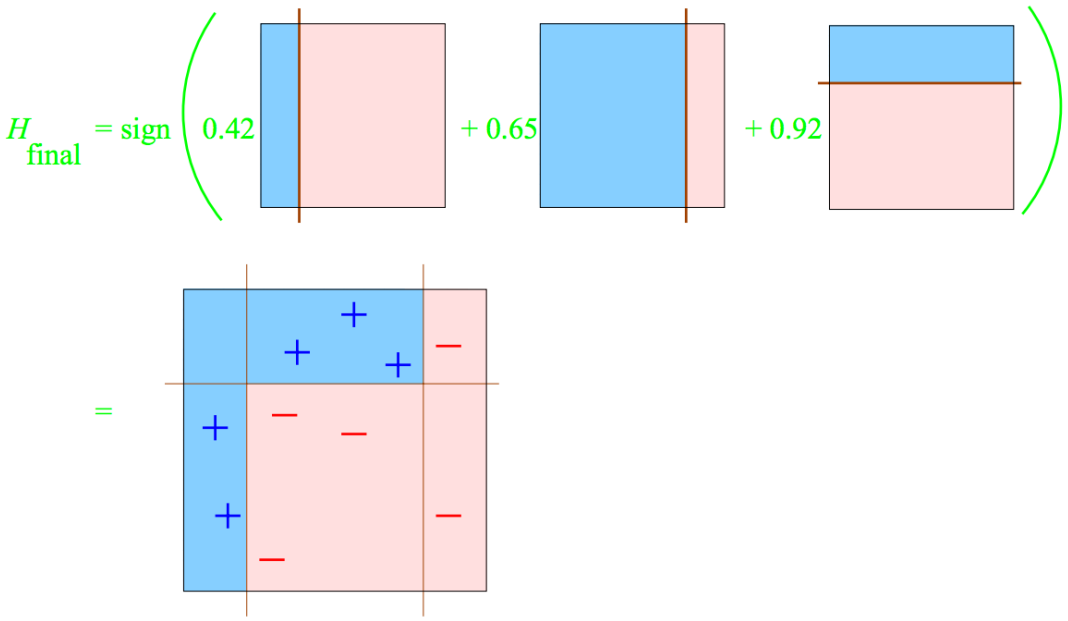


Round 3



- In round 2 we seperate the weighted points, then up-weight the misclassified points again
- In round 3 we seperate the result of round 2's weights with a horizontal line, since we already did two vertical lines and doing more vertical wouldn't seperate them well.
- We then weight all these classifiers based on their α_t which is the value of the error function that is a minimum. In this case it is $\log_e(\frac{1-p(misclassified)}{p(misclassified)})/2$ which for round $1 = log(\frac{0.7}{0.3})/2 \approx 0.424$

Final Hypothesis



Boosting in R

- Boosting can be used with any subset of classifiers
- One large subclass is **gradient boosting**.
- R has multiple boosting libraries. Differences include the choice of basic classification functions and combination rules.
  - **gbm** - boosting with trees.
  - **mboost** - model based boosting.
  - **ada** - statistical boosting based on **additive logistic regression**
  - **gamBoost** - for boosting generalized additive models.
- Most of these are available in the caret package

Wage Example

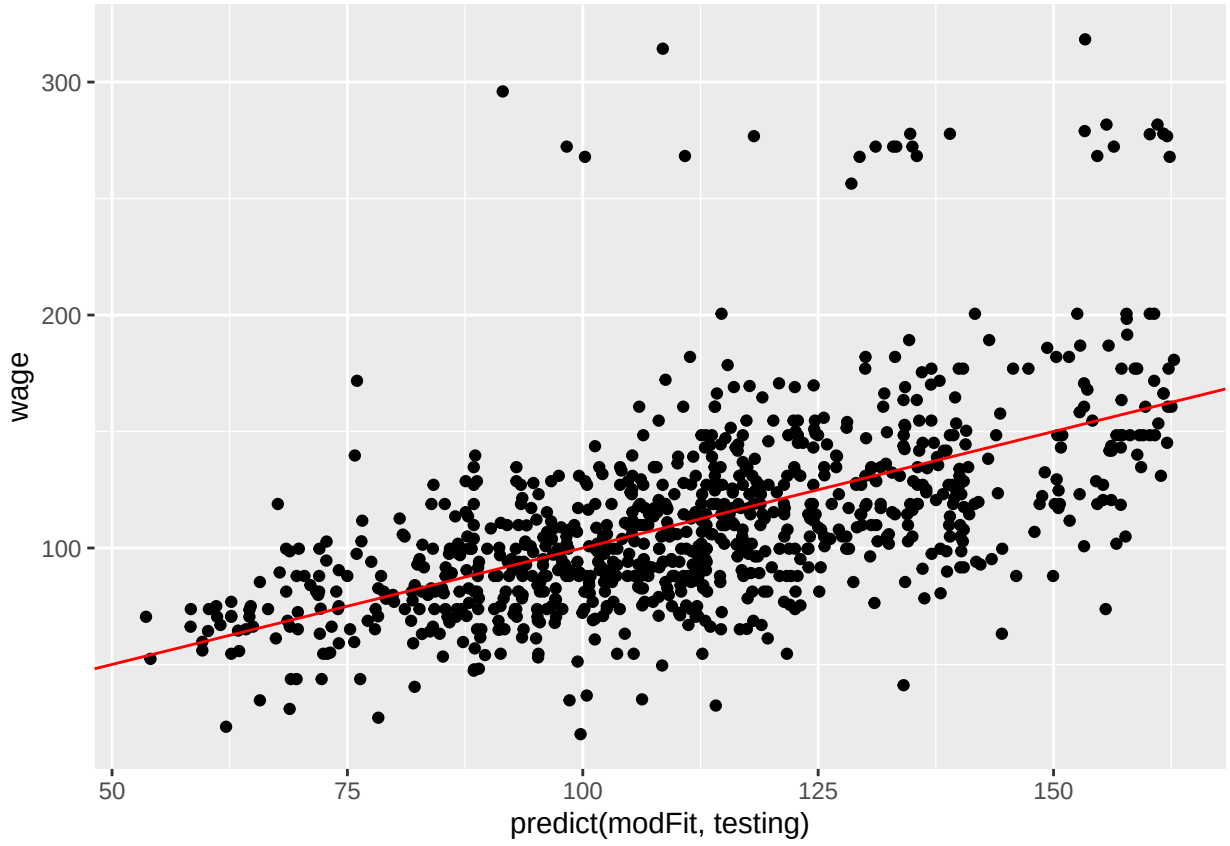
```
library(ISLR); data(Wage)
library(tidyverse)
```

```r
library(caret)
wage <- subset(Wage, select =
          -c(logwage))#Including logwage would just give a direct predictor
set.seed(1618)
inTrain <- createDataPartition(y = wage$wage,
                               p = 0.7, list = FALSE)
training <- wage[inTrain,]
testing <- wage[-inTrain,]
```

```r
modFit <- train(wage ~., method = "gbm", #boosting w/ trees
                data = training,
                verbose = FALSE) #Produces more output
print(modFit)
```

```
## Stochastic Gradient Boosting
##
## 2102 samples
##    9 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 2102, 2102, 2102, 2102, 2102, 2102, ...
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  RMSE      Rsquared   MAE
##   1                   50      34.62540  0.3136505  23.54452
##   1                  100      34.12601  0.3233824  23.20182
##   1                  150      34.07535  0.3247305  23.19608
##   2                   50      34.16029  0.3227701  23.16469
##   2                  100      34.09284  0.3244626  23.15279
##   2                  150      34.18130  0.3218321  23.25438
##   3                   50      34.08376  0.3244280  23.09039
##   3                  100      34.20015  0.3210006  23.24545
##   3                  150      34.35968  0.3163684  23.40135
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were n.trees = 150, interaction.depth =
##  1, shrinkage = 0.1 and n.minobsinnode = 10.
```

```r
# Plot the results
qplot(predict(modFit, testing), wage, data = testing) +
  geom_abline(slope = 1, intercept = 0, col = "#FF0000")
```



- Points will fall on the red line if we were to predict the exactly correct wage.

Notes and Further Reading

- A couple of nice tutorials for boosting:
  - **Freund and Shapire**
  - **Ron Meir**
- Boosting, random forests, and model ensembling are the most common tools that win Kaggle and other prediction contests.
  - **PDF about Netflix Prize winner**

Reminder to Commit (09), Delete this line *AFTER* Committing

Model Baded Predictions

Model Based Predictions

Reminder to Commit (10), Delete this line *AFTER* Committing

Quiz 3

Reminder to Commit (Q3), Delete this line *AFTER* Committing

Regularized Regression and Combining Predictors

Regularized Regression

Combining Predictors

Reminder to Commit (11), Delete this line *AFTER* Committing

Forecasting

Unsupervised Prediction

Reminder to Commit (12), Delete this line *AFTER* Committing

Quiz 4

Reminder to Commit (Q4), Delete this line *AFTER* Committing

Course Project

Reminder to Commit (P1), Delete this line *BEFORE* Committing