# Practical Machine Learning

# Week 1:

Why use prediction: We use prediction to be able to determine something about a set of data based on the features of that data.

# Importance of Steps

1. The first step, and the most important is having a good question. A good question is specific and is clear about what is being asked, and allows you to collect data.
2. The next most important step is being able to get data. It is often the case that we can’t get the necessary data that would be required to answer a certain question.
   1. Remember that garbage in = garbage out
   2. It’s easier to predict things that have a clear connection such as predicting new movie ratings from old movie ratings. A harder task is to predict response to certain diseases based on gene expression data since there are many factors at play.
3. Features matter a lot!
   1. Good features have the following properties
      1. They lead to data compression
      2. Retain relevant information
      3. Are created based on expert application knowledge
   2. Common mistakes
      1. Trying to automate feature selection
      2. Not paying attention to data-specific quirks
      3. Throwing away information unnecessarily
4. Algorithms matter less than you might think

When making an algo or method, here are the 5 things we care about:

1. Interpertability
2. Simplicity
3. Accurate \*most important
4. Fast to train and test.
5. Scalability

Prediction is about accuracy tradeoffs

# In and out of Sample Errors

**In Sample Error:** The error rate you get on the same data set you used to build our predictor. Also called resubstitution error.

**Out of Sample Error:** The error rate you get on a new data set. Sometimes called generalization error.

Key Ideas

1. We care about out of sample error
2. In sample error < out of sample error
3. The cause is overfitting

Data is made up of two parts:

* Signal
* Noise

The issue with overfitting is that you capture too much noise in your data set, thus causing you to not tune your parameters well to general data, but rather only the testing set. In fact, you can usually get in sample error rate of zero. However, this is only because you’re making a function to exactly fit the test data.

Goal: Find the signal, ignore the noise!

# Prediction Study Design

1. Define your error rate.
2. Split data into: training, testing, validation (optional)
3. On the training set pick features
   * Use cross validation
4. On the training set pick a prediction function
   * Use cross-validaiton
5. If no validation set
   * Apply prediction one time and only one time to the test set
6. If we have a validation set
   * Apply our prediction function to test set and refine it
   * Then appliy it one time and only one time to our validation set

**\*Avoid small sample sizes!!! Small sample sizes allow probability to play a role in classicfication. If we’re predicting a binary outcome, a sample size of 1 gives us a 50% chance of 100% accuracy.**

## Rules of Thumb for Prediction Study Design

When we have a large sample size:

* 60% of data should go to training set
* 20% should go to test set
* 20% should go to validation set

If we have a medium sample size:

* 60% of data should go to training set
* 40% should go to test set

If we have a small sample size:

* Do cross validation
* Report that we have used a small sample size in order to let others know that the results are subject to this caveat.

## Principles to Remember:

* Set the test/validation set aside and don’t look at it at all
* In general, randomly sample training and test sets
* Data sets must reflect the structure of the problem

# Types of Errors

## Basic Terms

**True Positive: Correctly identified**

**False Positive: Incorrectly identified**

**True Negative: Correctly rejected**

**False Negative: Incorrectly rejected**

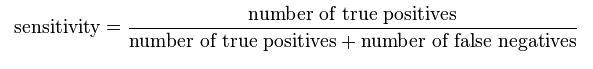
**Example:**

**True Positive: Sick people get correctly diagnosed as sick**

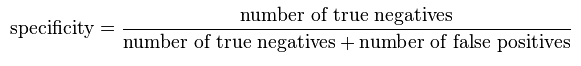
**False Positive: Healthy people get incorrectly identified as sick**

**True Negative: A healthy person is correctly identified as healthy**

**False Negative: A sick person is incorrectly identified as healthy**

**Sensitivity:** 

**Out of the people that actually have the disease, what percentage of them did we predict correctly, i.e how many tested positive (sick) for the disease?**

**Specificity:** 

Out of the people that actually don’t have the disease, what percentage of them did we predict correctly, i.e. how many tested negative (healthy) for the disease

## Worked Example

**A worked example**

A diagnostic test with sensitivity 67% and specificity 91% is applied to 2030 people to look for a disorder with a population prevalence of 1.48%

|  |  |  |  |
| --- | --- | --- | --- |
|  | | **Patients with**[**bowel cancer**](https://en.wikipedia.org/wiki/Bowel_cancer) **(as confirmed on**[**endoscopy**](https://en.wikipedia.org/wiki/Endoscopy)**)** | |
| Condition positive | Condition negative |
| [**Fecal occult blood**](https://en.wikipedia.org/wiki/Fecal_occult_blood) **screen test outcome** | Test outcome positive | **True positive** (TP) = 20 | **False positive** (FP) = 180 | [Positive predictive value](https://en.wikipedia.org/wiki/Positive_predictive_value)  = TP / (TP + FP) = 20 / (20 + 180) = **10%** |
| Test outcome negative | **False negative** (FN) = 10 | **True negative** (TN) = 1820 | [Negative predictive value](https://en.wikipedia.org/wiki/Negative_predictive_value)  = TN / (FN + TN) = 1820 / (10 + 1820) ≈ **99.5%** |
|  | | **Sensitivity**  = TP / (TP + FN) = 20 / (20 + 10) ≈ **67%** | **Specificity**  = TN / (FP + TN) = 1820 / (180 + 1820) = **91%** |  |

**Related calculations**

* False positive rate (α) = [type I error](https://en.wikipedia.org/wiki/Type_I_and_type_II_errors#Type_I_error) = 1 − specificity = FP / (FP + TN) = 180 / (180 + 1820) = 9%
* False negative rate (β) = [type II error](https://en.wikipedia.org/wiki/Type_I_and_type_II_errors#Type_II_error) = 1 − sensitivity = FN / (TP + FN) = 10 / (20 + 10) = 33%
* [Power](https://en.wikipedia.org/wiki/Statistical_power) = sensitivity = 1 − β
* [Likelihood ratio](https://en.wikipedia.org/wiki/Likelihood_ratios_in_diagnostic_testing) positive = sensitivity / (1 − specificity) = 0.67 / (1 − 0.91) = 7.4
* Likelihood ratio negative = (1 − sensitivity) / specificity = (1 − 0.67) / 0.91 = 0.37

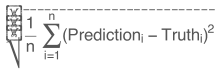
**So here’s some explanation and motivation behind all of this. Say we want to know what the probability is of us actually having the disease, given that we tested positive for it. What we do is take TP/(TP + FP) which is our positive predictive value.**

## **Continuous Data:**

**Mean Squared Error (MSE):**



**Root Mean Squared Error:**



## Common Measure Errors:

1. Mean squared errro
   * Continuous data, sensitive to outliers
2. Mean absolute deviation is used instead
   * Continuous data, often more robust
3. Sensitivity
   * If you want few missed positives
4. Specificity
   * If you want few negatives mislabeled as positives
5. Accuracy
   * Weights false positives/negatives equally
6. Concordance – multi class data
   * Kappa is an example

# ROC Curves

ROC (Receiver Operating Characteristic) Curves are used to measure the quality and the goodness of a prediction algo. They allow us to visualize the different values for sensitivity give a certain specificity depending on the cutoff point the algo uses.

X Axis: 1 – specificity

Y Axis: Sensitivity

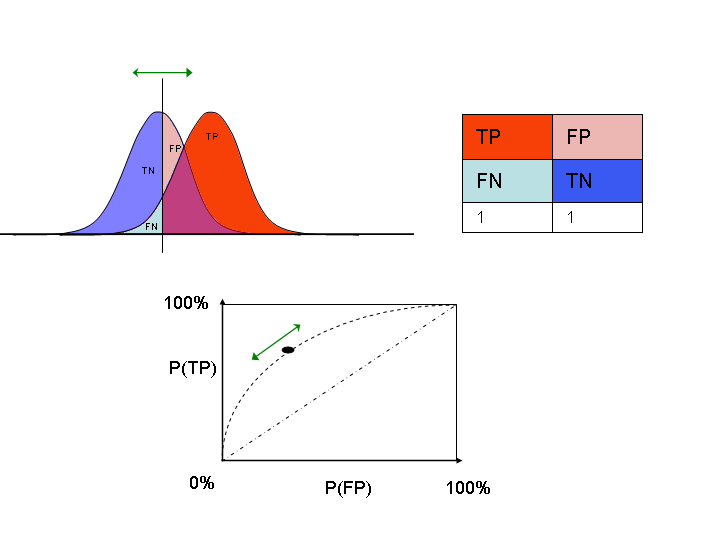
Comparing ROC Curves:

The measure is the area underneath a curve. The higher the area, the better the predictor is.

AUC = 0.5: random guessing

AUC = 1: perfect classifier

For example, imagine that the blood protein levels in diseased people and healthy people are [normally distributed](https://en.wikipedia.org/wiki/Normal_distribution) with means of 2 [g](https://en.wikipedia.org/wiki/Gram)/[dL](https://en.wikipedia.org/wiki/Decilitre) and 1 g/dL respectively. A medical test might measure the level of a certain protein in a blood sample and classify any number above a certain threshold as indicating disease. The experimenter can adjust the threshold (black vertical line in the figure), which will in turn change the false positive rate. Increasing the threshold would result in fewer false positives (and more false negatives), corresponding to a leftward movement on the curve. The actual shape of the curve is determined by how much overlap the two distributions have.



# Cross Validation

One of the most widely used tools for detecting relevant features, building models and estimating their paramters.

## Key Idea

1. Accuracy on training set is optimistic since it is in sample error
2. A better estimate comes from the test set.
3. Can’t use the test set when building the model otherwise it would become part of the training set.
4. So we need to somehow estimate the test set accuracy with the training set.

## Cross Validation

**Approach:**

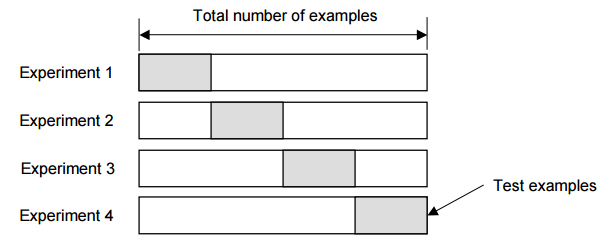
1. Use the training set
2. Split it into training/test sets
3. Build a model on the training set
4. Evaluate on the test set
5. Repeat and average the estimated errors

**Used For:**

1. Picking variables to include in a model based on which model has the best performance
2. Pick the type of prediction function to use. This is based on trying a few and choosing the one with best performance.
3. Picking parameters in the prediction function
4. Comparing different predictors

## K-Fold Cross Validation

* Break our dataset up into k equal-sized datasets



## Leave One Out Cross Validation

* Leave out only one observation for the testing set and use the rest of the observations to build a predictor and then apply that predictor to the one testing observation.
* For example, if we have 5 observation in our data set, the process would look like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration 1 | Testing | Training | Training | Training | Training |
| Iteration 2 | Training | Testing | Training | Training | Training |
| Iteration 3 | Training | Training | Testing | Training | Training |
| Iteration 4 | Training | Training | Training | Testing | Training |
| Iteration 5 | Training | Training | Training | Training | Testing |

Then we average out the error over each iteration

## Considerations

* For time series data, data must be used in chunks
* For k-fold cross validation
  + Larger k = less bias, more variance. Less bias means more accurate estimate of out of sample error rate
  + Smaller k = more bias, less variance. More bias means less accurate estimate of out of sampole error rate
* Random sampling must be done **without** replacement.
* If you do the random sampling with replacement, it’s called bootstrap
  + Understimates the error rate
  + Some samples appear more than once. So if say sample A appears more than once, and your method got sample A correct the first time, obviously it will get it correct the 2nd and 3rd times.
  + Can be corrected, but it can be complicated
* If you cross-validate to pick predictors, remember that in the end, you must estimate errors on independent data 1 time.

# What Data Should You Use

## Key Idea

* If you want to predict something about x, use something that’s as closely related as possible to x
* Looser the connection, the harder the prediction
* Data properties matter. Knowing how the data actually connects to the thing you’re trying to predict is very important
* Unrelated data is the most common mistake

# Week 2

# Caret Package

## Commands

**inTrain <- createDataPartition(dataset, percentage, list) #inTrain gets assigned an indicator function**

**training <- spam[inTrain, ]**

**test <- spam[-inTrain, ]**

**modelFit <- train(type ~., data=training, method=”glm)**

**predictions <- predict(modelFit, newData=testing)**

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

# Data Slicing

## Uses

* Building training and testing sets right at the beginning of our prediction function creation
* Performing cross validation or boostrapping within our training set in order to evaluate our models.

## Examples

> folds <- createFolds(y=spam$type, k=10, list=TRUE, returnTrain=TRUE)

> sapply(folds, length)

Fold01 Fold02 Fold03 Fold04 Fold05 Fold06 Fold07 Fold08 Fold09 Fold10

4141 4141 4141 4141 4142 4141 4140 4141 4140 4141

* Explantion:
* k specifies the number of folds we want
* returnTrain lets us know if we want the training set or the testing set
* Our dataset is about 4600 elements long. So if we want 10 folds, we want our testing set to be about 1/10 of that on each iteration/fold. Therefore 4600 - 0.1\*4600 = 4600-460 = 4140 which is about the length of each fold give or take one or two.

**Time Slices:**

|  |
| --- |
| > tme <- 1:1000  > folds <- createTimeSlices(y=tme, initialWindow = 20, horizon = 10)  > names(folds)  [1] "train" "test" |
| As an aside for R syntax, notice how  > folds$test[1]  $Testing001  [1] 21 22 23 24 25 26 27 28 29 30  > folds$test[[1]]  [1] 21 22 23 24 25 26 27 28 29 30  Which is because folds$test[1] is of type list and folds$test[[1]] is also of type list. So we have a list composed of a list.  \*To get the arguments for a function in R, use the args function. like so:  args(train.default) for example  > folds$train[[1]]  [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  **This code says that:** |
| * We want slices that have a window of about 20 samples in them. * Then the horizon says that we want to predict the next 10 samples after we take those initial 20. |

# Training Options

> args(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"), FALSE, TRUE), trControl = trainControl(),

tuneGrid = NULL, tuneLength = 3)

NULL

**Weights: Is used to set how much each observation should count for. Particularly useful for when you have an imbalance of observations for one type. For example, say in our spam data set that we have 1000 observations, 950 of which are nonspam while the rest are spam.**

**Metric Options:**

1. **Continuous outcomes**
   * **RMSE = Root Mean Squared Error**
   * **RSquared = R^2 from regression models**
2. **Categorical outcomes**
   * **Accuaracy = Fraction Correct**
   * **Kappa = A measure of concordance**

## **trainControl**

|  |
| --- |
| > args(trainControl)  function (method = "boot", number = ifelse(grepl("cv", method),  10, 25), repeats = ifelse(grepl("cv", method), 1, number),  p = 0.75, search = "grid", initialWindow = NULL, horizon = 1,  fixedWindow = TRUE, verboseIter = FALSE, returnData = TRUE,  returnResamp = "final", savePredictions = FALSE, classProbs = FALSE,  summaryFunction = defaultSummary, selectionFunction = "best",  preProcOptions = list(thresh = 0.95, ICAcomp = 3, k = 5),  sampling = NULL, index = NULL, indexOut = 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: bootstrap or cross validation**

## **trainControl Resampling**

**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**
  + **large number of repeats can slow things down**

## **Setting the seed**

* It is often useful to set an overall seed. There is a random draw created when doing a cross validation.
* If you set a seed, that means the same random numbers will be created each time
* You can set a seed for each resample. This is for parallel computation
* Seeding for each resample is useful for parallel fits, but isn’t very useful when not doing things in parallel

# Plotting Predictors

One of the most important parts of building a machine;earning algo is understanding how the data look and interact with each other.

The best way to do this is by plotting the data, and in particular, plotting the predictors.

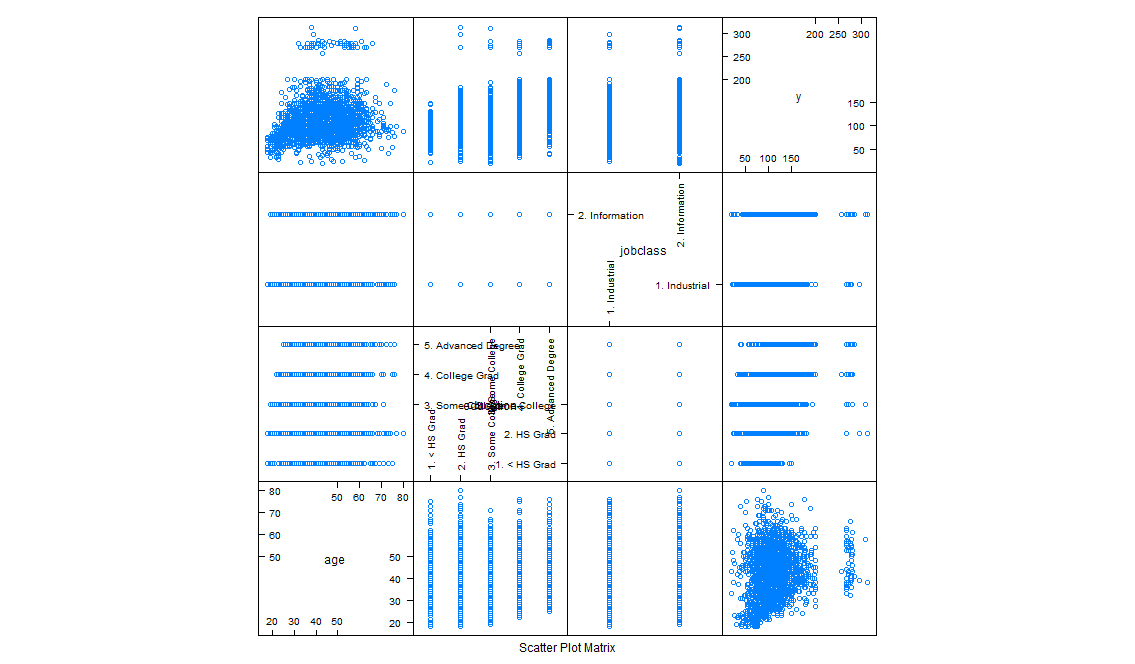
Exmaple: Predicting Wages

> featurePlot(x=training[, c("age", "education", "jobclass")],

+ y=training$wage,

+ plot="pairs")

This gives us a nice scatterplot matrix that plots every column we mentioned (age, education, jobclass) vs. the wage, and vs. each other.

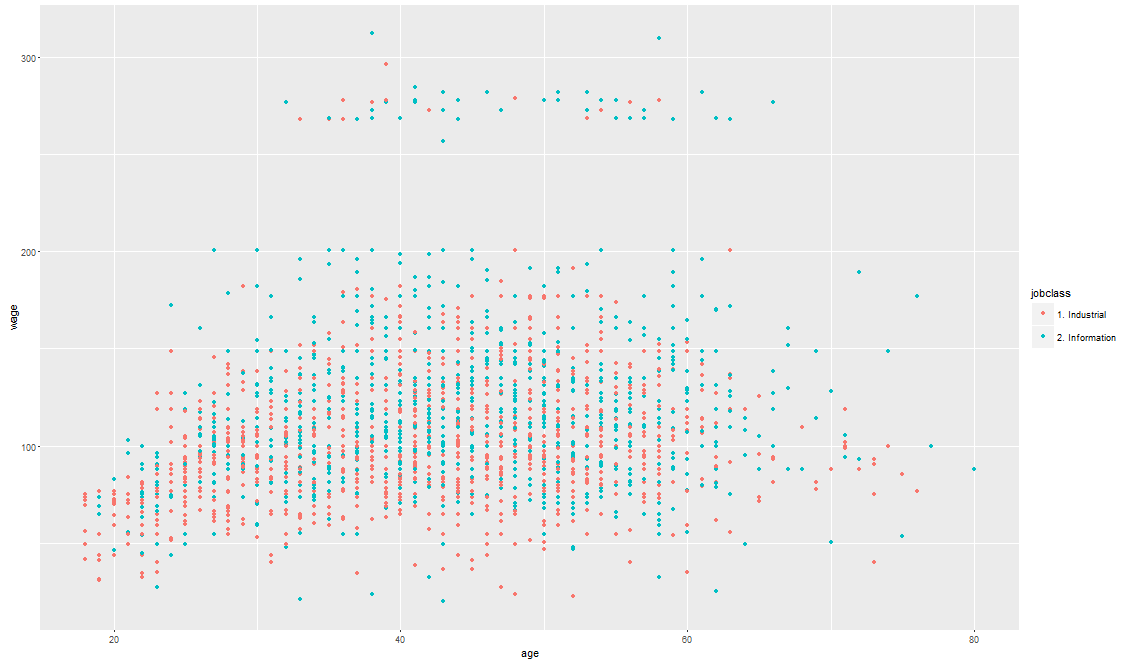


For example, the upper leftmost corner is a plot of age versus wage. Notice the area that I highlighted in red and how it’s narrower than the bottom area.

**Initial Hypothesis:** This indicates that there are no very old people or very young people making a lot of money. The ones making a lot seem to be middle aged.

So let’s investigate this even more using the following

> qplot(age, wage, color=jobclass, data=training)

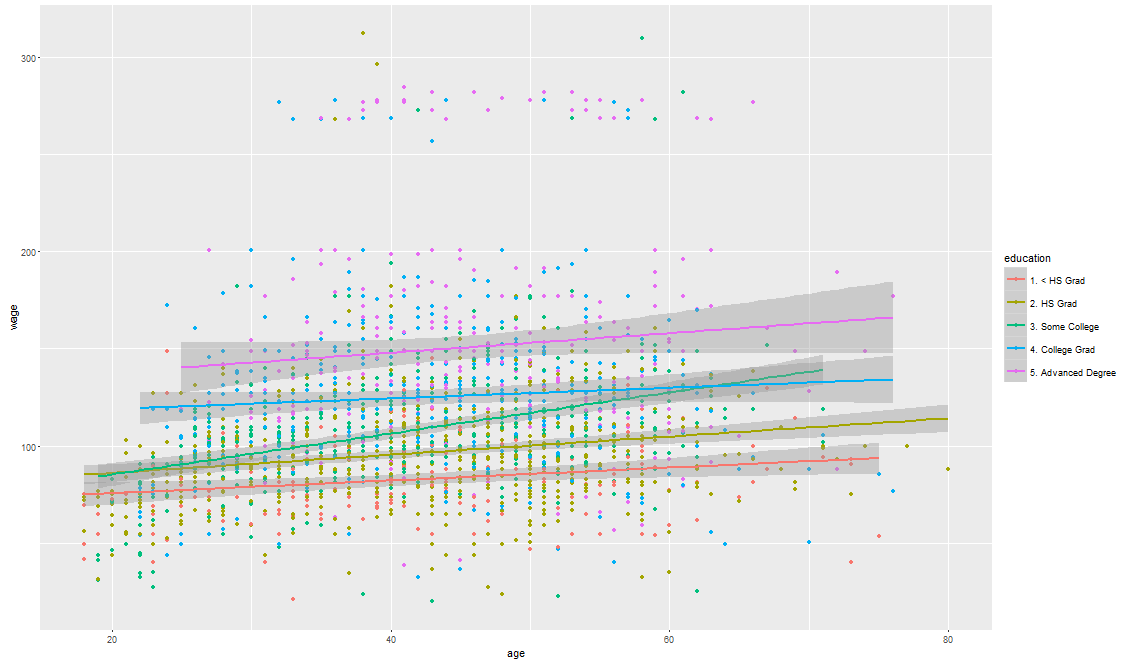


**Revised Hypothesis:** After examining jobclass as a potential underlying variable to this, we see that most of the people earning in the upper chunk come from the information based jobs. That could be a lingering variable in this case, had we only considered age.

## Regression Smoothers

> qq <- qplot(age, wage, colour=education, data=training)

> qq + geom\_smooth(method="lm", formula=y~x)



* This fits a linear model for each education level

## Making Factors

* If we want to split up the data into quantile groups, we can use the cut2 command

> cutWage = cut2(training$wage, g=3)

> table(cutWage)

cutWage

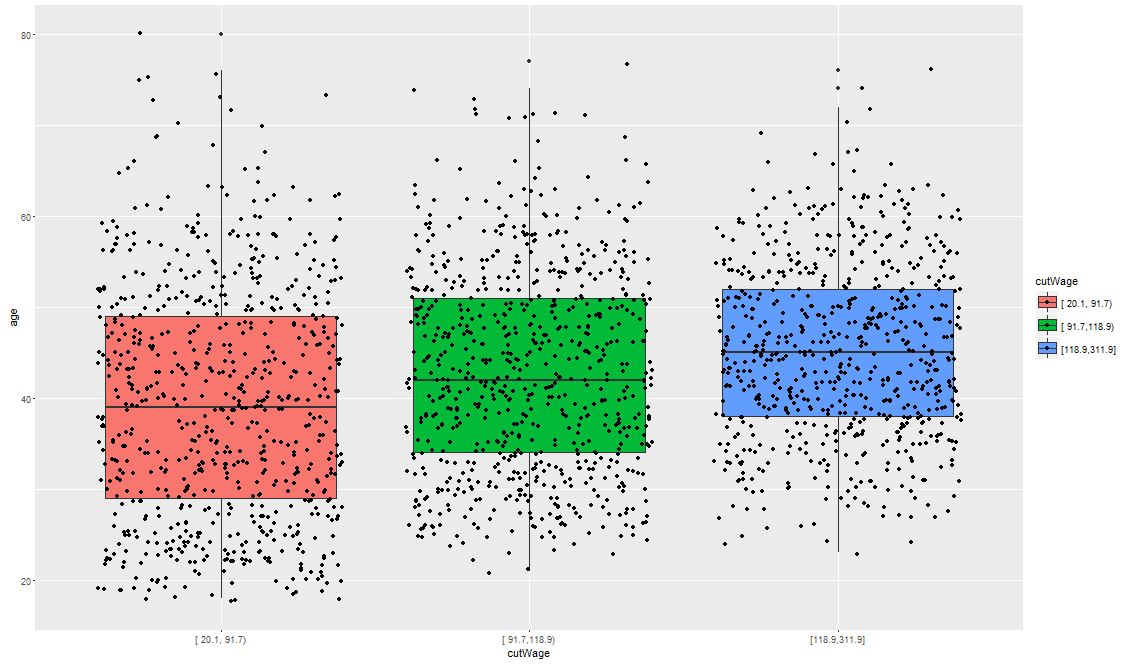
[ 20.1, 91.7) [ 91.7,118.9) [118.9,311.9]

707 718 677

## Boxplot With Factors and Points Overlayed

> p2 = qplot(cutWage, age, data=training, fill=cutWage, geom=c("boxplot", "jitter"))

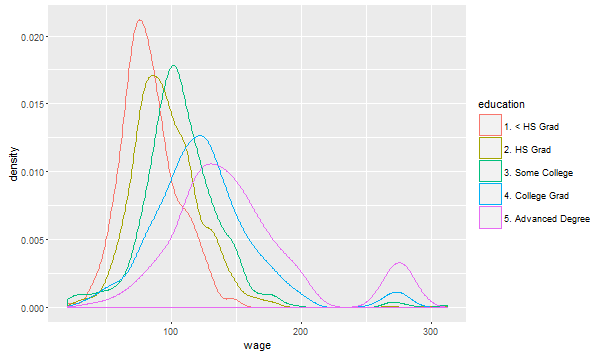
> p2



* This graph shows the spread of age for the 3 different income groups. As can be seen, the higher income wage group, which was the one at the top of our scatter plot, has a tightly grouped middle age population composing it.

## Density Plots

* Sometimes density plots can show things that boxplots simply can’t
* > qplot(wage, colour=education, data=training, geom="density")



* This plot shows a group outgroup for the advanced degree people near the highest wages.
* IT also shows how the peak of the advanced degree people is to the right of those without an education which means that they earn more on average. This can be verified by the following commands

|  |
| --- |
| > adv = tapply(training$wage, training$education, mean)  > adv  1. < HS Grad 2. HS Grad 3. Some College 4. College Grad  82.83874 96.46761 107.69513 125.08562  5. Advanced Degree  150.51457  > adv = tapply(training$wage, training$education, median)  > adv  1. < HS Grad 2. HS Grad 3. Some College 4. College Grad  81.28325 94.07271 104.92151 120.75807  5. Advanced Degree  141.77517 |
|  |
| |  | | --- | |  | |

## Noted and Further Reading

* Make plots ONLY in the training data!

**Keep an eye out for:**

* Imbalance in outcomes/predictors. I.e., if we have data favoring one outcome group heavily (a lot of SPAM and little HAM)
* Outliers. Outliers or groups of outliers could suggest that there are some variables we are missing/overlooking
* Groups of points that are not explained by any of the predictors
* Skewed variables

# Covariate Creation

## What is a Covariate?

* Covariates are sometimes called predictors or features
* They are variables that we will be using in our model and we will combine them to predict an outcome that we care about

## Levels of Covariates

* Level 1 Raw Data 🡪 Covariates
  + Taking the raw data that we have and turning it into a predictor that we can use
  + This raw data is usually an image or text file and we need to summarize that info somehow into either a quantitative or qualitative variable
  + So what we want to do is take that raw data and turn it into covariates that describe that data as much as possible while also compressing it
  + Example: Suppose we have an email. It’s hard to plug the email itself into a prediction function, so what we do is create some features (covariates) that describe the email. Like how many times certain words appear, how many capital letters there are on average, and so on.
  + Err on the side of MORE features NOT LESS
  + Sometimes you need to use automatic feature creation when dealing either things like images
* Level 2 Tidy Covariates 🡪 New Covariates
  + So tidy covariates are features you've already created on the data set, and then creating new covariates out of them
  + Usually involves transformations or functions of the covariates
  + This is sometimes more necessary for regression or support machine vectors (SVMs)
  + Less necessary for classification trees
  + Make sure we’re only working with the training set when creating/deciding on these functions. Otherwise we get OVERFITTING

## Dummy Variables

* Allow us to create quantitative variables out of qualitative variables thereby helping our machine learning algos since they no longer have to deal with qualitative variables

> inTrain <-createDataPartition(Wage$wage, p=0.7, list=FALSE)

> help("createDataPartition")

> training <- Wage[inTrain,]

> testing <- Wage[-inTrain,]

> table(training$jobclass)

1. Industrial 2. Information

1090 1012

> dummies <- dummyVars(wage ~ jobclass, data=training)

> head(predict(dummies, newdata=training))

jobclass.1. Industrial jobclass.2. Information

231655 1 0

161300 1 0

11443 0 1

377954 0 1

81404 0 1

302778 0 1

* So what we get from this is two quantitative variables (binary) that indicate yes/no for industrial job and yes/no for information job
* Covariates typically need to have some variability in them in order to be useful
  + For example, if we create a covariate indicating whether or not an email has any letters in it, it will pretty much always be true. This won’t help uswith any kind of analysis

## Removing Zero Covariates

* One thing we want to do is remove the covariates that have near zero variability since they’re not useful

|  |
| --- |
| > nsv <- nearZeroVar(training, saveMetric=TRUE)  > nsv  freqRatio percentUnique zeroVar nzv  year 1.011299 0.33301618 FALSE FALSE  age 1.157143 2.90199810 FALSE FALSE  sex 0.000000 0.04757374 TRUE TRUE  maritl 3.106291 0.23786870 FALSE FALSE  race 8.721393 0.19029496 FALSE FALSE  education 1.444915 0.23786870 FALSE FALSE  region 0.000000 0.04757374 TRUE TRUE  jobclass 1.077075 0.09514748 FALSE FALSE  health 2.468647 0.09514748 FALSE FALSE  health\_ins 2.263975 0.09514748 FALSE FALSE  logwage 1.012821 19.74310181 FALSE FALSE  wage 1.012821 19.74310181 FALSE FALSE |
|  |
| |  | | --- | | > | |

* From this we can see that sex and region are useless covariates on this dataset since everyone is male and comes from the same region

## Spline Basis

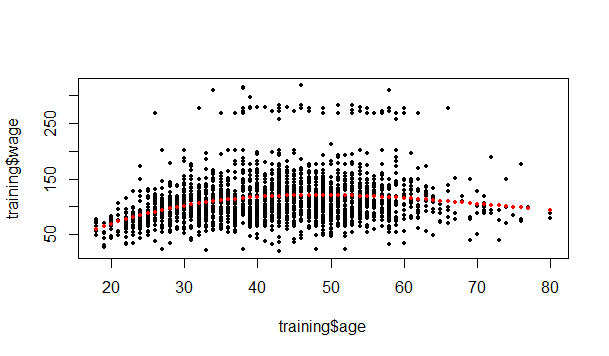
* Sometimes it is more useful to fit data with a curve rather than a straight line
* We can do this with basis functions via the spline package
* Consider this example:
* bsBasis <- bs(training$age, df=3)
* The bs() function simply creates a polynomial variable and what results is a 3-column matrix
* It generates a basis matrix for representing the family of piecewise polynomials with the specified interior knots and degree, evaluated at the values of x

## Fitting Curves with Splines

> lml = lm(wage ~ bsBasis, data=training)

> plot(training$age, training$wage, pch=19, cex=0.5)

> points(training$age, predict(lml, newdata=training), col="red", pch=19, cex=0.5)



* As can be seen, this fits a curve to the training data.
* In the lm() function, the wage ~ bsBasis part says that we are trying to predict wage using the bsBasis variable.
* So we are fitting a linear model, but our basis is polynomials so we have a linear combination of polynomials

## Important

* We need to keep in mind that when predicting with the test set, we have to use the same procedure that we used in the training set to come up with the covariates.

# Preprocessing with Principal Component Analysis

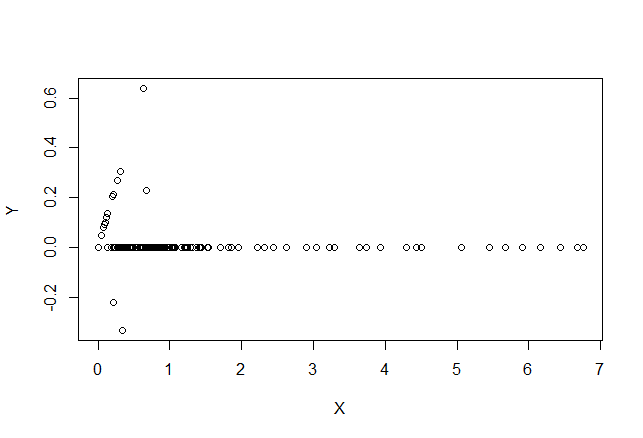
## Correlated Predictors

* The idea is that sometimes we’ll have many quantitative vairables, and they’ll be highly correlated with eacother
* In other words, they’ll be very similar to being pretty much the exact same variable
* In this case, tehre’s no need to include them all in our model
* How to determine which variables are correlated with eachother:

|  |
| --- |
| > M <- abs(cor(training[, -58]))  > diag(M) <- 0  > which(M > 0.8, arr.ind = T)  row col  num857 32 31  num415 34 31  telnet 31 32  num415 34 32  telnet 31 34  num857 32 34 |
|  |
| |  | | --- | | > | |

* + The preceding code does the following:
    - IT creates a correlation matrix M
    - It sets the diagonal of that matrix to 0, since we know that a variable is 100% correlated with itself so we don’t want to consider that case
    - Then we find cells with a value of correlation > 0.8
  + IF we dig deeper we see that
    - > M[32, 34]
    - [1] 0.9945068
  + And that corresponds to
    - > names(spam)[c(32,34)]
    - [1] "num857" "num415"
  + So this means that num857 and num415 are found together very often

## Basic PCA (Principal Components Analysis)

* How can we take the two variables and create just one variable out of them?
  + One idea is to take a weighted combination of these variables that explains the most of what’s going on.
  + Benefits:
    - Reduced number of predictors
    - Reduced Noise due to averaging
* We might not need every predictor
* Example Continued:
* > X <- 0.71 \* training$num415 + 0.71 \* training$num857
* > Y <- 0.71 \* training$num415 - 0.71 \* training$num857
* > plot(X, Y)
* 
* As we can see, most of the variability occurs on the X axis
* Therefore, adding the two variables together (X axis) captures more information than subtracting the two variables (Y axis)

## Related Problems

* 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 original data.

The first goal is statistical and the second goal is data compression

## Related Solutions – PCA/SVD (Singular Value Decomposition)

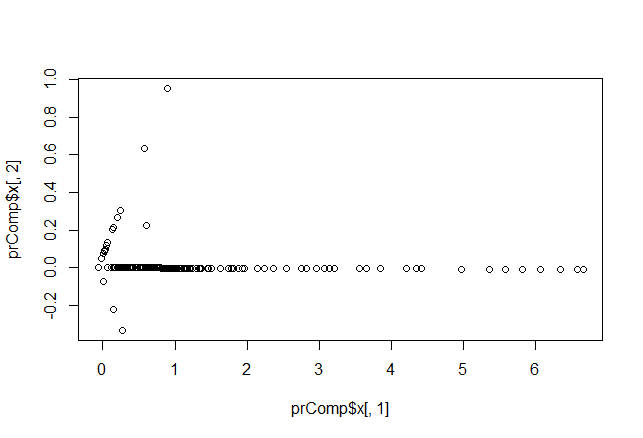
### SVD

* 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 = UDVT
* Where the columns of U are orthogonal (left singular vectors), the columns of V are orthogonal (right singular vectors) and D is a diagonal matrix (singular values)
* \*\*\* The variables in V are construcuted in such a way as to show the maximum number of variation in the data

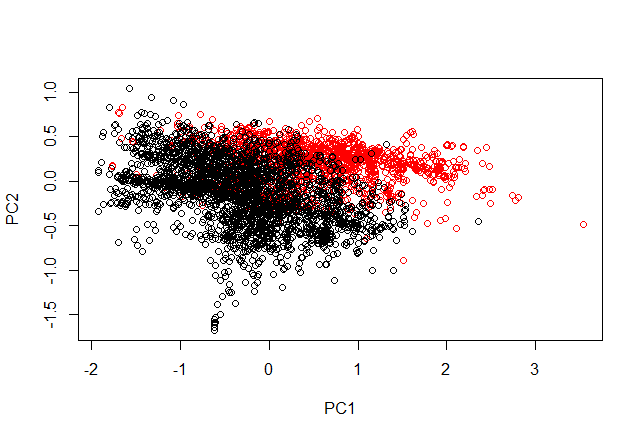
### PCA

* The principal components are equal to the right singular values if you first scale (subtract the mean, divide by the standard deviation) the variables

## Principal Components in R – prcomp

* Let’s take the variables that were highly correlated with each other in the spam dataset
* > smallSpam <- spam[,c(34,32)]
* > prComp <- prcomp(smallSpam)
* > plot(prComp$x[,1], prComp$x[,2])
* 
* We see a very similar plot to the earlier plot where the first principal component looks a lot like adding the two variables together, and the second looks a lot like subtracting the two variables
* So then why bother with principal components?
  + It allows you to perform this operation even if you have more than 2 variables
  + Lets you look at and reduce a large number of quantitative variables
* > prComp$rotation
* PC1 PC2
* num415 0.7080625 0.7061498
* num857 0.7061498 -0.7080625
* Now we see why he chose 0.71 earlier

## PCA on SPAM Data

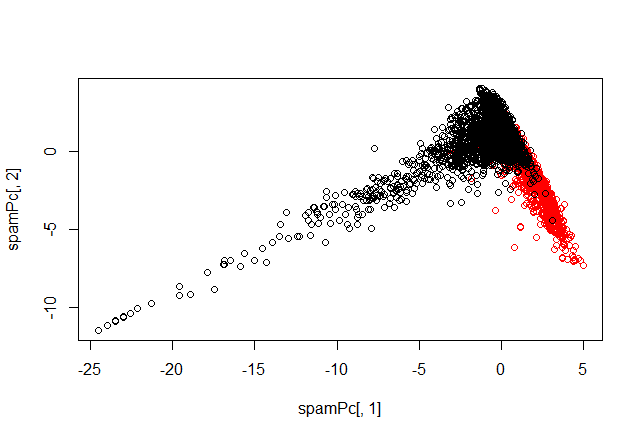
* Can do this for more variables
* > typeColor <- ((spam$type=="spam")\*1 + 1)
* > prComp <- prcomp(log10(spam[,-58]+1)) #the log10 and + 1 was done to make the data look a little more gaussian
* > plot(prComp$x[,1], prComp$x[,2], col=typeColor, xlab="PC1", ylab="PC2")
* 
* We often have to transform the data with functions like log10 in order to make the skewed variables look more sensible
* The rotation matrix of prComp now has 57 columns!
* PC1 explains the most variation in the data, then PC2 , then PC3 and so on
* Red is SPAM and black is HAM
* Keep in mind that PC1 is no longer a very simple addition of 2 vvaraibles. It could be a complicated combination of all the variables in the dataset

## 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)



* So the principal compoenents in this case are 2 new variables
* They are essentially a model that you fit to the data
* The idea is that if we get a new observation, we have to predict what the principal component will look like for that new observation

## Preprocessing with PCA

> plot(spamPc[,1], spamPc[,2], col=typeColor)

> trainPc <- predict(preProc, log10(training[,-58]+1))

> modelFit <- train(training$type ~ ., method="glm", data=trainPc)

* In the test data set, you have to use the same principal components as you calculated in the training data set

> testPc <- predict(preProc, log10(testing[,-58]+1)) # This simply gives us the value of the principal components we came up with before, but now for the test data set

> confusionMatrix(testing$type, predict(modelFit, testPc)) # The predict call in this line actually uses

Confusion Matrix and Statistics

Reference

Prediction nonspam spam

nonspam 658 39

spam 64 389

Accuracy : 0.9104

95% CI : (0.8924, 0.9263)

No Information Rate : 0.6278

P-Value [Acc > NIR] : < 2e-16

Kappa : 0.8106

Mcnemar's Test P-Value : 0.01804

Sensitivity : 0.9114

Specificity : 0.9089

Pos Pred Value : 0.9440

Neg Pred Value : 0.8587

Prevalence : 0.6278

Detection Rate : 0.5722

Detection Prevalence : 0.6061

Balanced Accuracy : 0.9101

'Positive' Class : nonspam

* Notice how high our accuaracy is, even though we only used 2 principal components

## Final Thoughts on PCs

* Most useful for linear-type models
* Can make it harder to interpret the predictors
* Watch out for outliers
  + Transform first with logs or Box Cox
  + Plot predictors to identify problems

# Predicting With Regression

## 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

## Eruption Duration Versus Waiting Time

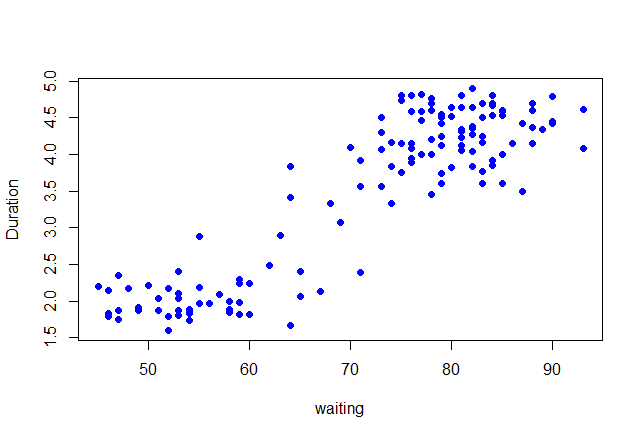
> set.seed(333)

> inTrain <- createDataPartition(y=faithful$waiting, p=0.5, list=FALSE)

> trainFaith <- faithful[inTrain,]

> testFaith <- faithful[-inTrain,]

> plot(trainFaith$waiting, trainFaith$eruptions, pch=19, col="blue", xlab="waiting", ylab="Duration")



* We can see that there is roughly a linear relationship for this data.
* We can draw a diagonal line that would predict duration based on waiting time

## Fit a Linear Model

> lml <- lm(eruptions ~ waiting, data=trainFaith)

> summary(lml)

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

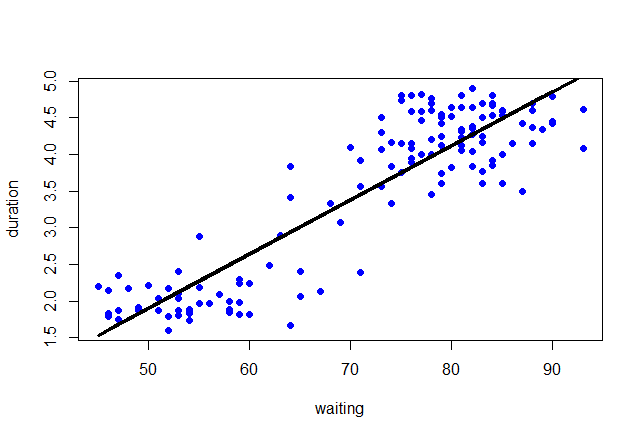
F-statistic: 551 on 1 and 135 DF, p-value: < 2.2e-16

* So the lm() function is saying to create a linear regression model that predicts the value of eruptions based on the value of waiting using the data trainFaith
* -1.792739 is our y intercept and 0.073901 is our slope

## Model Fit

> plot(trainFaith$waiting, trainFaith$eruptions, pch=19, col="blue", xlab="waiting", ylab="duration")

> lines(trainFaith$waiting, lml$fitted, lwd=3)



## Predict a new value

> coef(lml)[1] + coef(lml)[2]\*80

(Intercept)

4.119307

* This predicts the value of duration for an 80-second-long waiting time

> newdata <- data.frame(waiting=80)

> predict(lml, newdata)

1

4.119307

## Plot Predictions – Training and Test

* Keep in mind that we built this linear model on the training set, and now we want to see how it does on the test set

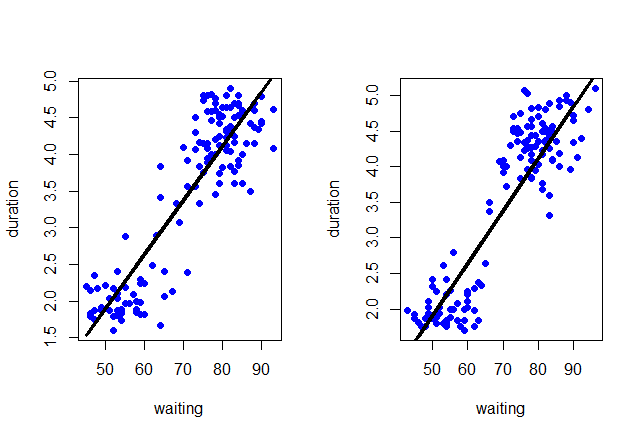
> par(mfrow=c(1,2))

> plot(trainFaith$waiting, trainFaith$eruptions, pch=19, col="blue", xlab="waiting", ylab="duration")

> lines(trainFaith$waiting, predict(lml), lwd=3)

> plot(testFaith$waiting, testFaith$eruptions, pch=19, col="blue", xlab="waiting", ylab="duration")

> lines(testFaith$waiting, predict(lml, newdata=testFaith), lwd=3)



* We can see that our model doesn’t fit the test set as well as it fit the training set. It looks a bit tilted down and to the right
* It still captures the overall trend though

## Get Training Set/Test Set Errors

* RMSE Root Mean Squared Error on training set

> sqrt(sum((lml$fitted-trainFaith$eruptions)^2))

[1] 5.75186

* RMSE on Test Set

> sqrt(sum((predict(lml, newdata=testFaith) - testFaith$eruptions)^2))

[1] 5.838559

## Prediction Intervals

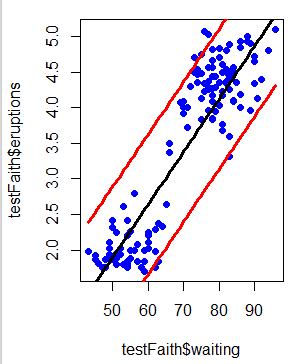
> matlines(testFaith$waiting[ord],pred1[ord,], type="l",,col=c(1,2,2), lty=c(1,1,1), lwd=3)

> pred1 <- predict(lml, newdata=testFaith, interval="prediction")

> ord <- order (testFaith$waiting)

> plot(testFaith$waiting, testFaith$eruptions, pch=19, col="blue")

> matlines(testFaith$waiting[ord],pred1[ord,], type="l",,col=c(1,2,2), lty=c(1,1,1), lwd=3)



## Linear Regression with Caret

> 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

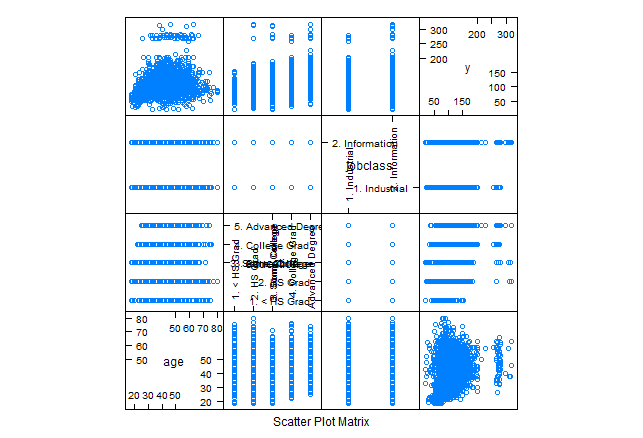
# Predicting With Regression, multiple covariates

## Example Predicting Wages

> Wage <- subset(Wage, select=-c(logwage))

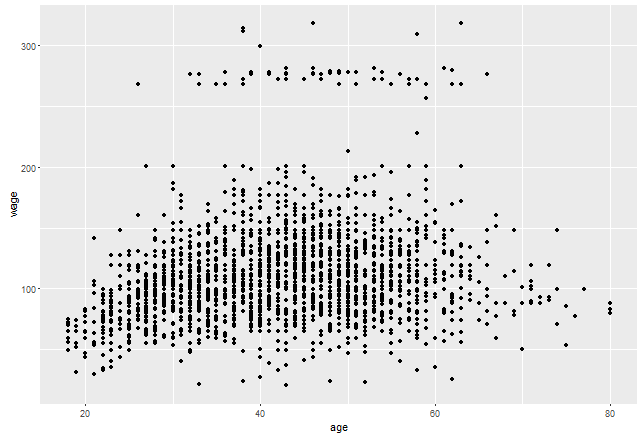
* So Wage now contains everything except for the logvalue of the wage

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



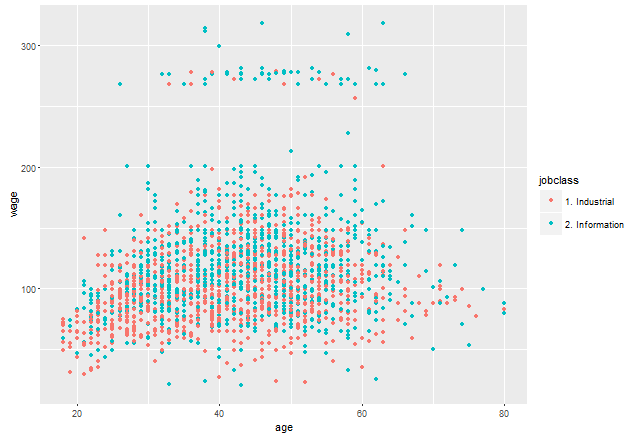
### Plot age versus wage

> qplot(age, wage, data=training)



### Plot age versus wage color by jobclass

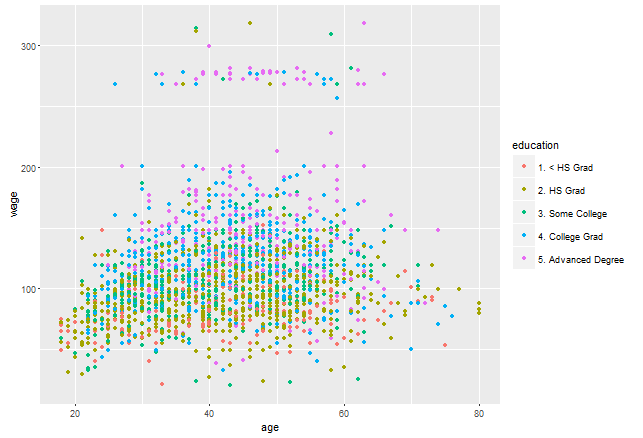
> qplot(age, wage, colour=jobclass, data=training)



* Notice how most of the observations in the top outlier group come from the information jobclass
* This indicates that the jobclass variable might be able to predict some of the variability that exists within the top group

### Plot age versus wage colour by education

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



* This plot indicates that the people having advanced degrees get paid more
* So some combination of education and jobclass could explain why the relationship between age and wage isn’t just a perfect relationship in one big cloud but has an outlier group too

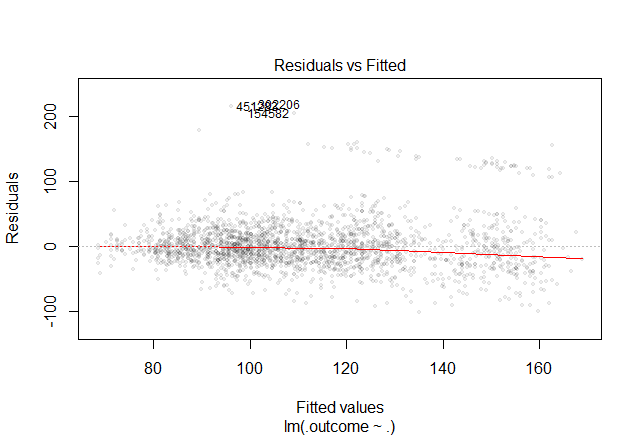
## Fit a Linear Model

|  |
| --- |
| > modFit <- train(wage ~ age + jobclass + education, method="lm", data=training)  > finMod <- modFit$finalModel  > print(modFit)  Linear Regression  2251 samples  10 predictor  No pre-processing  Resampling: Bootstrapped (25 reps)  Summary of sample sizes: 2251, 2251, 2251, 2251, 2251, 2251, ...  Resampling results  RMSE Rsquared RMSE SD Rsquared SD  35.33919 0.2769054 1.248185 0.02400764 |
|  |
| |  | | --- | |  | |

* Keep in mind that we use both age, jobclas, and education to predict wage
  + Jobclass and education are both factor variables in R, so by default indicator variables are created.
* Also note that the final model has 10 poredictors even though we only put 3 variable names into the formula
  + The reason being is that the education variable got more than one predictor because of the way we had to create the indicator functions

## Diagnostics

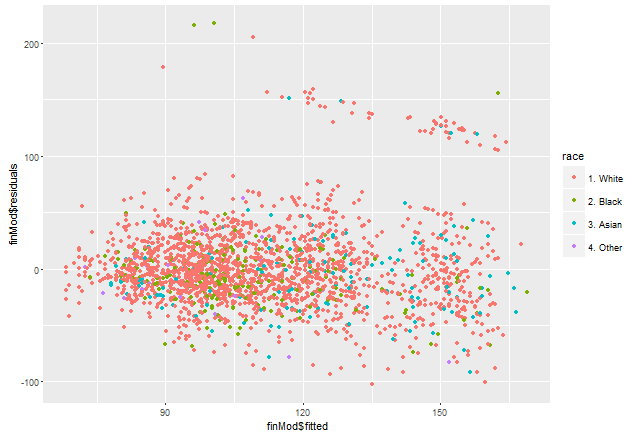
> plot(finMod, 1, pch=19, cex=0.5, col="#00000010")



* What we’d like to see in this plot is the red line flat at 0
* So the predictions are on the x axis
* Residual is the amount of variation left over after we fit our model
* We have some outliers labelled at the top of the plot

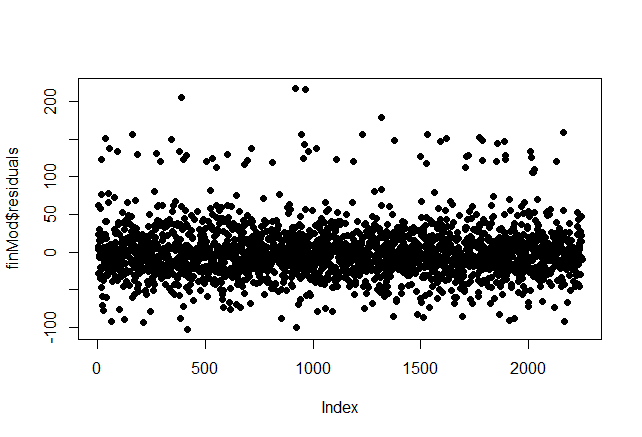
## Color by Variables not Used in Model

> qplot(finMod$fitted, finMod$residuals, color=race, data=training)



## Plot by Index

> plot(finMod$residuals, pch=19)



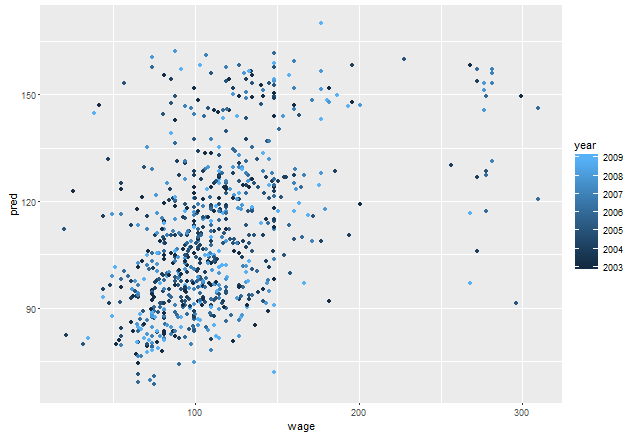
* In our case there doesn’t seem to be a relationship. But if there were a clear relationship or even a clear group of outliers in this plot, then that’s an indication that a variable could possibly be missing
  + This relationship is usually with respect to time or age or something like that

## Predicted versus Truth in Test Set

* Another thing we can do is plot the wage variable in the test set vs. the predicted values in the test set

> pred <- predict(modFit, testing)

> qplot(wage, pred, colour=year, data=testing)



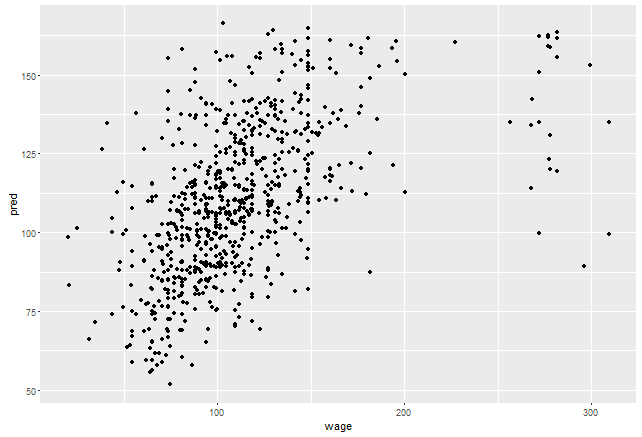
* Ideally these would all be very close to eachothers
* Ideally they would be on a 45 degree line
* Now if we find something here in the test set, we acna’t go back and update our model on the test set based on this new info
* This is more of a post mortem

## If We Want to Use All Covariates

> modFitAll <- train(wage ~ ., data=training, method="lm")

> pred <- predict(modFitAll, testing)

> qplot(wage, pred, data=testing)



* We can see that this model does a little bit better than our model that only used 3 variables

## Notes and Further Reading

* Linear regression is often useful in combination with other models
* It can capture a lot of variability if the relationship between the predictors and the outcome is in fact linear
* Exploratory Data analysis can be very useful with regression

# Week 3

# Predicting with Trees

## Key Ideas

* If you have a bunch of variables that you want to use to predict an outcome, you can take each of those variables and use it to split the outcome into different groups
* As you split the outcome into different groups, then you can evaluate the homogeneity of the outcome in each group
* Split again if necessary until you get outcomes split into groups that are homogenous enough or small enough that you need to stop

### Pros

* Easy to interpret
* Better performance in nonlinear settings

### Cons

* Without pruning/cross validation it can lead to over fitting
* Harder to estimate uncertainty
* Results may be variable

## Basic Algorithm

1. Start with all variables in one group
2. Find the first variable that best splits the outcome into 2 homogeneous groups
3. Then 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/homogeneous

## Measures of Impurity

### Misclassification Error

* In the Obama example, misclassification error would be 1 – probability that you would vote for Obama
* 0 means perfect purity, no misclassification error and all of the counties would vote for Obama
* 0.5 = no purity, when the leaf is totally balanced between the 2 outcomes

|  |  |
| --- | --- |
| 1down vote | If y^iy^i is your prediction for the iith observation then the misclassification rate is 1n∑iI(yi≠y^i)1n∑iI(yi≠y^i), i.e. it is the proportion of misclassified observations. In R you can easily calculate this by mean(y\_predicted != y\_actual). Note that this only applies to the case where yy is a categorical class label and not a continuous response.  As Christoph described in his comment, you don't directly get class labels from a logistic regression. You need to threshold the predicted posterior probabilities in order to get your y^iy^i. |

### Gini Index

* 0 perfect purity
* 0.5 no purity

### Deviance and Information Gain

0 = perfect purity

1 = no purity

### Example

* Suppose we had a variable trying to split 16 dots into blue or red dots
  + If the variable split it into 15 blue dots and 1 red dot
    - Misclassification would be 1/16 = 0.06
    - Gini: 1 – [(1/16)^2 + (15/16)^2] = 0.12
    - Information Gain = 0.34

## Example in R

> data("iris")

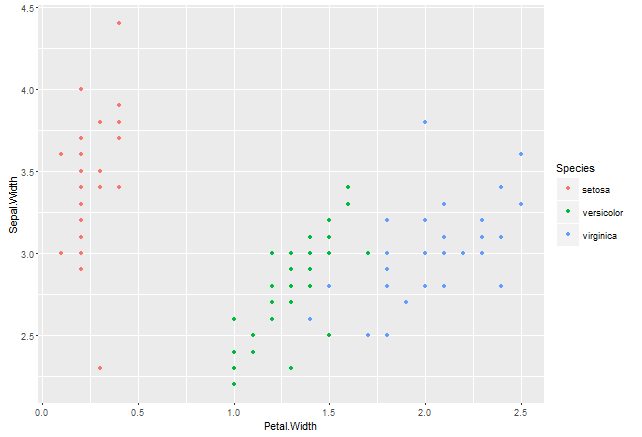
> names(iris)

|  |
| --- |
| [1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"  > table(iris$Species)  setosa versicolor virginica  50 50 50 |
|  |
| |  | | --- | | > | |

* So we have 4 variables to use in order to try and predict the species

### Petal Width vs Sepal Width

> qplot(Petal.Width, Sepal.Width, colour=Species, data=training)



* Notice that there are 3 distinct clusters
* This makes it an easy classification problem

|  |
| --- |
| > modFit <- train(Species ~ ., method="rpart", data=training)  Loading required package: rpart  > 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.Width< 1.75 38 3 versicolor (0.00000000 0.92105263 0.07894737) \*  7) Petal.Width>=1.75 32 0 virginica (0.00000000 0.00000000 1.00000000) \* |
|  |

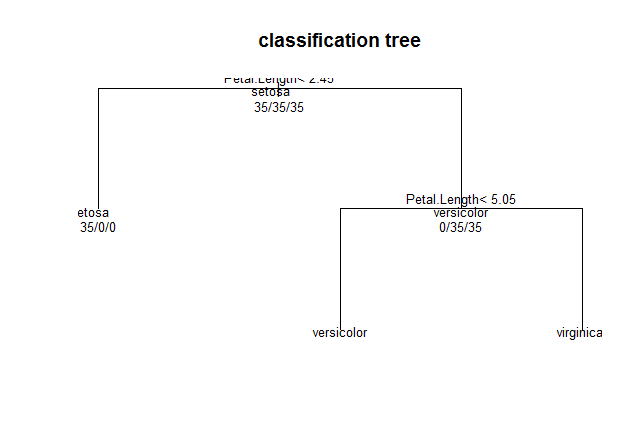
* The method=”rpart” is R’s package for doing regression with classification trees

2) says that every flower with Petal.Length < 2.45 belongs to Setosa

### Plot Tree

> plot(modFit$finalModel, uniform=TRUE, main="classification tree")

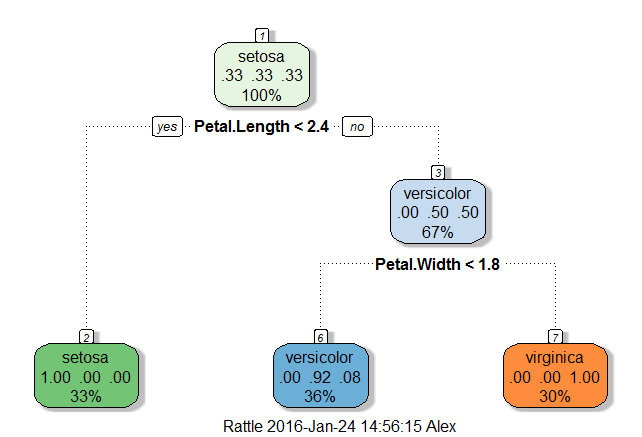
> text(modFit$finalModel, use.n=TRUE, all=TRUE, cex=0.8)



> library(rpart.plot)

> fancyRpartPlot(modFit$finalModel)

> fancyRpartPlot(modFit$finalModel)



### Predicting New Values

> predict(modFit, newdata=testing)

[1] setosa setosa setosa setosa setosa setosa setosa setosa

[9] setosa setosa setosa setosa setosa setosa setosa versicolor

[17] versicolor versicolor versicolor versicolor versicolor versicolor versicolor versicolor

[25] virginica versicolor versicolor versicolor versicolor versicolor virginica virginica

[33] virginica virginica virginica virginica versicolor virginica virginica virginica

[41] versicolor virginica virginica virginica virginica

Levels: setosa versicolor virginica

## Final Notes

* Classification trees are non linear models
  + They use interactions betyween variables
  + Data transformations may be les important
  + Tress can also be used for reression 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

# Bagging – Bootstrapping and aggregating

## Basic Idea

1. Resample cases and recalcuatlte predicitons. Take your data and take resamples of the data set. After you resample the cases with replacement, then you recalculate your precitions
2. Average or majority vote

## Notes

* You get a similar bias that you would get from fitting anyone of theose models indididually
* But you get a reduced variance
* Most useful for non linear funcitons

Given a standard [training set](https://en.wikipedia.org/wiki/Training_set) *D* of size *n*, bagging generates *m* new training sets D_i, each of size *n′*, by [sampling](https://en.wikipedia.org/wiki/Sampling_(statistics)) from *D* [uniformly](https://en.wikipedia.org/wiki/Probability_distribution#With_finite_support) and [with replacement](https://en.wikipedia.org/wiki/Sampling_(statistics)#Replacement_of_selected_units). By sampling with replacement, some observations may be repeated in each D_i. If *n*[*′*](https://en.wikipedia.org/wiki/Prime_(symbol))=*n*, then for large *n* the set D_i is expected to have the fraction (1 - 1/[*e*](https://en.wikipedia.org/wiki/E_(mathematical_constant))) (≈63.2%) of the unique examples of *D*, the rest being duplicates.[[1]](https://en.wikipedia.org/wiki/Bootstrap_aggregating#cite_note-1) This kind of sample is known as a [bootstrap](https://en.wikipedia.org/wiki/Bootstrap_(statistics)) sample. The *m* models are fitted using the above *m* bootstrap samples and combined by averaging the output (for regression) or voting (for classification).

Bagging leads to "improvements for unstable procedures" (Breiman, 1996), which include, for example, [artificial neural networks](https://en.wikipedia.org/wiki/Artificial_neural_networks), [classification and regression trees](https://en.wikipedia.org/wiki/Classification_and_regression_tree), and subset selection in [linear regression](https://en.wikipedia.org/wiki/Linear_regression) (Breiman, 1994). An interesting application of bagging showing improvement in preimage learning is provided here.[[2]](https://en.wikipedia.org/wiki/Bootstrap_aggregating#cite_note-2)[[3]](https://en.wikipedia.org/wiki/Bootstrap_aggregating#cite_note-3) On the other hand, it can mildly degrade the performance of stable methods such as K-nearest neighbors (Breiman, 1996).

## Ozone Data

> 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 want to predict temperature as a function of ozone

## Bagged Loess

> ll <- matrix(NA, nrow=10, ncol=155)

* So we create a matrix with 10 rows and 155 columns
* Then what we’re going to do is resample the dataset for 10 different samples
* Each time we’re gonna sample with replacement from the entire dataset
* Then we’re gonna create a new dataset ozone0 which is the resampled dataset for that particular element of the loop. Ust a subset of the dataset corresponding to our random sample
* Then we reorder ozone0 everytime by the ozone variable
* Then we fit a loess curve each time. Just a smooth curve we can fit through data. Very similar to spline from before
* Then for every single loess curve, we predict the outcome for a new data set

> for(i in 1:10) {

+ ss <- sample(1:dim(ozone)[1], replace=T)

+ ozone0 <- ozone[ss,]; ozone0 <- ozone0[order(ozone0$ozone),]

+ loess0 <- loess(temperature ~ ozone, data=ozone0, span=0.2) #span just says how smooth we want the curve to be

+ ll[i,] <- predict(loess0, newdata=data.frame(ozone=1:155))

+ }

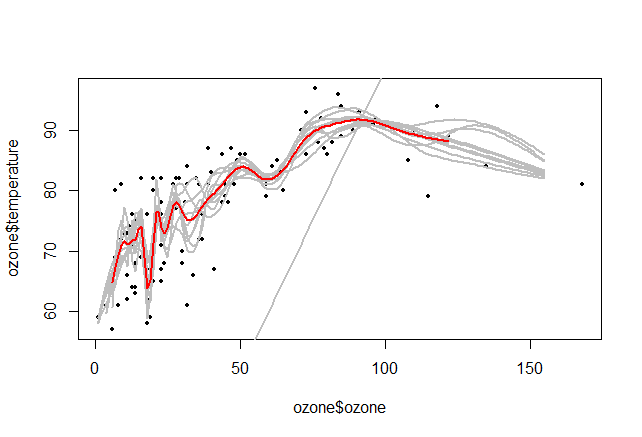
* To go in depth about the sample() method, it simply creates a set of dim(ozone) random whole numbers all in the range of 1 to dim(ozone)
* Okay so notice how we’re predicting with a different curve, but the same values of 1 to 155 each time. This is so that we can be show the difference between all the curves and the final averaged curve that we are about to draw. It wouldn’t make sense to have newdata set to something different for each predict call in this case

> plot(ozone$ozone, ozone$temperature, pch=19, cex=0.5)

> for(i in 1:10) {lines(1:155,ll[i,], col="grey", lwd=2)}

> lines(1:155, apply(ll,2,mean), col="red", lwd=2)

* Okay so let’s understand what’s going on with these plotting functions
* So the parameter pch in the plot() function specifies the type of point we want to see
* The parameter lwd technically stands for line width, but in this case it is the width of the symbol representing each point
* The lines() function is what draws the curves by connecting each point in the data that we provide to it.



* The red line is the bagged loess curve. It is averaged

To illustrate the basic principles of bagging, below is an analysis on the relationship between [ozone](https://en.wikipedia.org/wiki/Ozone) and temperature (data from [Rousseeuw](https://en.wikipedia.org/wiki/Peter_Rousseeuw) and Leroy (1986), available at [classic data sets](https://en.wikipedia.org/wiki/Classic_data_sets), analysis done in [R](https://en.wikipedia.org/wiki/R_(programming_language))).

The relationship between temperature and ozone in this data set is apparently non-linear, based on the scatter plot. To mathematically describe this relationship, [LOESS](https://en.wikipedia.org/wiki/Local_regression) smoothers (with span 0.5) are used. Instead of building a single smoother from the complete data set, 100 [bootstrap](https://en.wikipedia.org/wiki/Bootstrap_(statistics)) samples of the data were drawn. Each sample is different from the original data set, yet resembles it in distribution and variability. For each bootstrap sample, a LOESS smoother was fit. Predictions from these 100 smoothers were then made across the range of the data. The first 10 predicted smooth fits appear as grey lines in the figure below. The lines are clearly very *wiggly* and they overfit the data - a result of the span being too low.

By taking the average of 100 smoothers, each fitted to a subset of the original data set, we arrive at one bagged predictor (red line). Clearly, the mean is more stable and there is less[overfit](https://en.wikipedia.org/wiki/Overfitting).

## Bagging in Caret

* Will always have lower variability but similar bias
* Some models that already perform bagging for you if you use the train function and the following arguments for the “method” option
  + bagEarth
  + treeBag
  + bagFDA
* Alternatively, you can bag any model you choose using the bag function

## More Bagging in Caret

> predictors <- data.frame(ozone=ozone$ozone)

> temperature <- ozone$temperature

> treebag <- bag(predictors, temperature, B = 10,

+ bagControl = bagControl(fit = ctreeBag$fit,

+ predict = ctreeBag$pred,

+ aggregate = ctreeBag$aggregate))

* The idea is to first take our predictor variable and put it into one data frame
* Then we store our outcome variable in a vector called temperature
* Then to our bag() function we pass our predictors, the variable we’re trying to predict, B is the number of subsamples we want to use.
* bagControl tells us about how we’re going to fit the model
  + fit is the function that will be applied ot fit the model every time. This could be a call to the train function in the caret package
  + predict is the way that given a parituclar modelFit we are going to predict new values. This could be a call to the predict function in the caret package
  + aggregate is the way we’re going to put the predicitons together. I.e. how we want to average them out

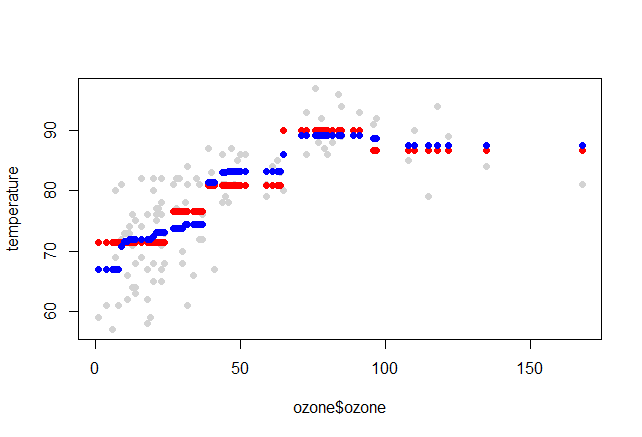
## Example of Custom Bagging

* Here the grey dots represent actual observe values
* The red dots represent the fit of a single conditional refression ttree
* The blue points are the fit from the bagged regression. As you can see this captures the shape and trend of the data much better than the red points

> plot(ozone$ozone, temperature, col="lightgrey", pch=19)

> points(ozone$ozone, predict(treebag$fits[[1]]$fit, predictors), pch=19, col="red")

> points(ozone$ozone, predict(treebag, predictors), pch=19, col="blue")



## Part of Bagging

### ctreeBag$fit

* For the fit part, it takes the data frame that we’ve passed and the outcome that we passed and it uses the ctree function to train a conitional regression tree over the dataset

### ctreeBag$pred

* The prediction takes in the object from the ctree modelfit, and a new dataset x and it will get a new prediction.
* It basically calucaltes each time the tree response

## Final Thoughts

* 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

# Random Forests

## Basic Idea

1. Bootstrap samples. Rebuild calassification or regression trees on each of those bootstrapped samples
2. At each split, bootstrap variables. Only a subset of the variables is considered at each potential split. Makes for a diverse set of potential trees that can be built
3. Grow multiple trees and vote or average those trees

### Pros

1. Accuracy

### Cons

1. Can be slow
2. Hard to interpret
3. Overfitting. Hard to understand which trees lead to that overfitting

## Algo

* Build a large number of trees where each tree is based on a bootstrapped sample
* At each node we allow a different subset of the variables to potentially contribute to the splits
* Then for a new input, we see the outcome that each tree generates for that input and we average those predictions in order to get the predicted probability of each class across all the different trees.

## Iris Example

> inTrain <- createDataPartition(y=iris$Species, p=0.7, list=FALSE)

> training <- iris[inTrain,]

> testing <- iris[-inTrain,]

|  |
| --- |
| > modFit <- train(Species ~ ., data=training, method="rf", prox=TRUE)  > 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 Accuracy SD Kappa SD  2 0.9426065 0.9119106 0.03003797 0.04719492  3 0.9473049 0.9191337 0.02939650 0.04630099  4 0.9475191 0.9194375 0.03150657 0.04953213  Accuracy was used to select the optimal model using the largest value.  The final value used for the model was mtry = 4. |
|  |
| |  | | --- | | > | |

### Explanation

* The prox=TRUE argument producres a little bit more extra information that we can use when building modelFits.
* So it says that we’ve built the model ,and we’ve done bototstrap resampling.
* We’ve alos tied a bunch of different tuning paramters. The tuning parameter in particular is the number of repeated trees it’s going to build.

### Getting a Single Tree

> getTree(modFit$finalModel, k=2)

left daughter right daughter split var split point status prediction

1 2 3 4 0.80 1 0

2 0 0 0 0.00 -1 1

3 4 5 4 1.70 1 0

4 6 7 3 5.25 1 0

5 0 0 0 0.00 -1 3

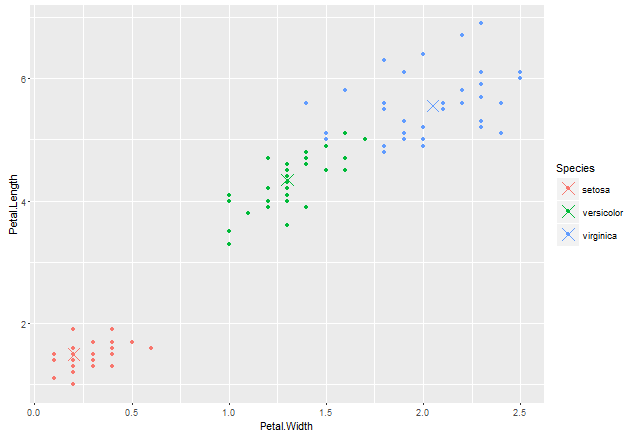
6 0 0 0 0.00 -1 2

7 0 0 0 0.00 -1 3

* K specifies which tree we want to look at
* Each row corresponds to a particular split
* The “var split” column shows us which variable we’re splitting on
* The “split point” column is the value where that variable is split
* The “prediction” column is what the prediction is going to be at that particular split

### Class “enters”

|  |
| --- |
| > irisP <- classCenter(training[,c(3,4)], training$Species, modFit$finalModel$proximity)  > irisP <- as.data.frame(irisP); irisP$Species <- rownames(irisP)  > p <- qplot(Petal.Width, Petal.Length, col=Species, data=training)  > p + geom\_point(aes(x=Petal.Width, y=Petal.Length, col=Species), size=5, shape=4,  data=irisP) |



* So here we’re looking only at petal length and petal width
* Each dot represents an observation
* Each X shows the observation center for each of the predicitons

### 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 15 1  virginica 0 0 14 |
|  |

* We can see that one of our predictions was wrong. We predicted a versicolor when the true value should have been virginica

|  |
| --- |
| > qplot(Petal.Width, Petal.Length, colour=predRight, data=testing, main="Newdata Predictions") |
|  |
| * We can see in this graph the value we didn’t predict properly, as well as what the petal length and width of this observation * Un surprisingly, this value is between two different classes if we look back at the previous graph |

## Final Notes

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

# Boosting

## Basic Idea

* Take lots of (possibly) weak predictors
* Weight them in a way that takes advantage of their strength and add them up
* By averaging them together, we get a stronger predictor.

## Algo

1. Start with a set of classifiers h1 to hk
   1. Examples: all possible trees, all possible regression models, all possible cutoffs
2. Create a classifier that combines classification functions
   1. Goal is to minimize error (on training set)
   2. Iterative, select one h at each step
   3. Calculate weights based on errors we get from that h
   4. Upweight missed classifications and select next h

Most famous boosting algorithm is **adaboost**

## 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 classifications functions and combination rules
  + gbm – does boosting with trees
  + mboost – model based boosting
  + ada – statistical boosting based on additive logistic regression
  + gamBoost – for boosting generalized additive models
* Most are available in the caret package

## Wage Example

> Wage <- subset(Wage, select=-c(logwage))

> inTrain <- createDataPartition(y=Wage$wage, p=0.7, list=FALSE)

> training <- Wage[inTrain,]

> testing <- Wage[-inTrain,]

>

> modFit <- trian(wage ~ ., data=training, method="gbm", verbose=FALSE)

> print(modFit)

Stochastic Gradient Boosting

2102 samples

10 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 RMSE SD Rsquared SD

1 50 35.36706 0.3148331 1.556055 0.03034328

1 100 34.78179 0.3240294 1.490142 0.02962657

1 150 34.69932 0.3251985 1.448145 0.03013448

2 50 34.80974 0.3243022 1.471688 0.02987964

2 100 34.68260 0.3255075 1.431039 0.02936904

2 150 34.78145 0.3216457 1.461068 0.02919576

3 50 34.81825 0.3213591 1.499781 0.03424497

3 100 34.90380 0.3170521 1.488759 0.03283339

3 150 35.13635 0.3087901 1.465999 0.03099938

Tuning parameter 'shrinkage' was held constant at a value of 0.1

Tuning parameter 'n.minobsinnode' was

held constant at a value of 10

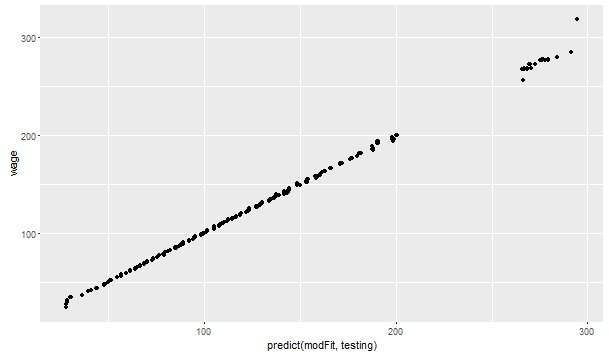
RMSE was used to select the optimal model using the smallest value.

The final values used for the model were n.trees = 100, interaction.depth = 2, shrinkage = 0.1

and n.minobsinnode = 10.

* Here we’re using gbm which does boosting with trees
* Verbose=FALSE gets rid of the output we don’t cara about
* So when we print the modelFit, we can see that there’s a different number of trees that are used and different interaction depths. These are used together to build a boosted verison of regression trees

> qplot(predict(modFit, testing), wage, data=testing)



* From this graph we can see that our predictions are pretty good since they lie along a 45 degree line.

# Model Based Prediction

## Basic Idea

1. Assume the data follow a probabilistic model
2. Use Baye’s theorem to find optimal classifiers

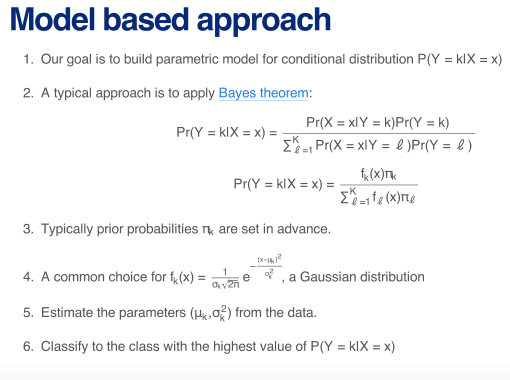
### Pros

* Can take advantage of the structure of the data
  + For example, if the data follow a specific distribution
* May be computationally convenient
* Are reasonably accurate on real problems

### Cons

* Make additional assumptions about the data
* When the model is incorrect you may get reduced accuracy

## Algo



Assume some parametric model for

the distribution of the features given the class

The denominator in this equation is just the law of total probability for X=x

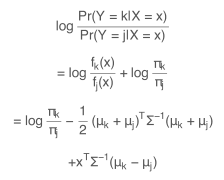
* The prior probabilities, pi k, are usually set in advance from the data
* A common choice for fk of x is a Gaussian distribution. It may be a multivariate Gaussian distribution if there are multiple x variables.
* And then we might estimate the parameters, mu k and sigma squared k, from the data. Then once we have these parameters estimated, we can calculate the probability y belongs to any given class, as soon as we observe the predictor variables

## Classifying Using the Model

* Linear Discriminant Analysis assumes f\_k(x) is multivariate guassian with **same** covariances
  + Draws lines through the data
* Quadratic discriminant analysis assumes f\_k(x) is multivariate Gaussian with **different** covariances
  + Draws quadratic curves through the data
* Model based prediction assumes more complicated versions of the covariance matrix
* Naïve Bayes assumes independence between features for model building

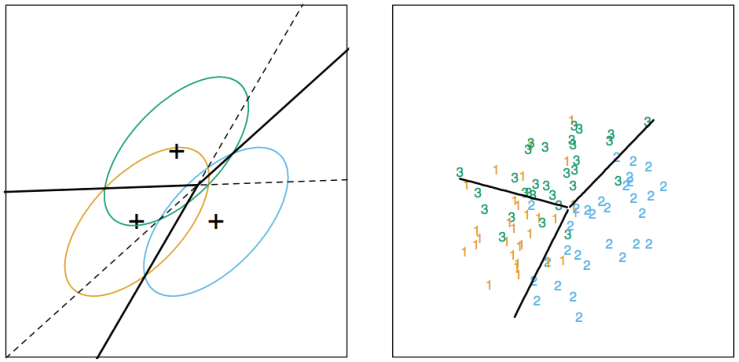
## Why Linear Discriminant Analysis

* This basically draws lines between the data, and covariance space



* Why it’s called “linear”
* If we consider the ratio of the probabilities of the two classes, and we take the log of that quantitiy, log is a monotone function which means that as this ratio increased, so will its log.

## Decision Boundaries



So here’s what’s happening in the diagrams above

* So there are 3 different classes that we care about
* We have two variables which we are using for classification, in this case that’s the x and y axis
* What we end up doing in the diagram on the left is fitting one Gaussian distribution to each class
* Then we draw lines where the probability switches from being higher for one calss to another class
* In the diagram on the right, if you are to the right of the purple line, you will get classified as a 2. If you are to the left of it, you will be classified as a 3, and so on.

## Discriminant Function



* In general the discriminant fnction is what gets used.
* Uk is the eman oof class k for all our features
* Sigma inverse is the inverse of the covariance matrix
* So here is what we want to do with this function
  + So xT is the value of our newly observed features. We plug those into the function, then we pick the value of k that maximixes this function
  + We usually estimate the paramters with maximum likelihood

## Naïve Bayes

* Does a little bit more to simplify the problem

**1.** 

* In this formula, pi\_k is just P(Y=k).
* We can consider the bottom part of the formula to just be some constant so we can rewrite it as follows
* **2.** 
* So **1** Is proportional to **2** Which is to say that maximizing **2** would be the same as maximizing **1**
* If we assume that all of the predictors are independent, we can write

**3.** 

* This is a naïve assumption, hence the name Naïve Bayes
* It’s particularly useful for when we have a very large number of features that are binary or categorical variables
* Comes up in text classification

## Iris Example

> inTrain <- createDataPartition(iris$Species, p=0.7, list=FALSE)

> training <- iris[inTrain,]

> testing <- iris[-inTrain,]

> modlda <- train(Species ~ ., data=training, method="lda")

> modnb <- train(Species ~ ., data=training, method="nb")

> plda <- predict(modlda,testing)

> pnb <- predict(modnb, testing)

> table(plda, pnb)

pnb

plda setosa versicolor virginica

setosa 15 0 0

versicolor 0 16 0

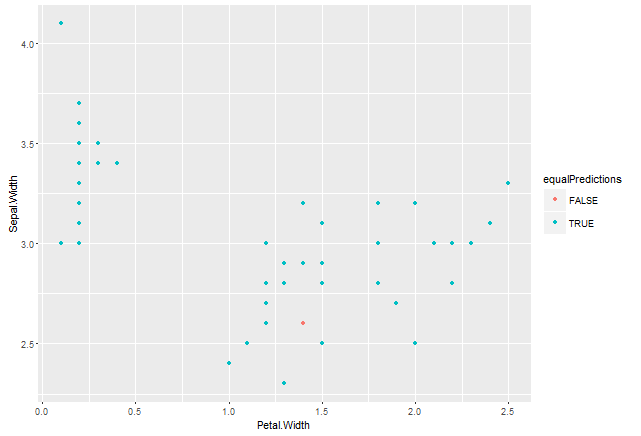
virginica 0 1 13

* Notice how the two models agree on all but one of the classifications

### Comparison of Results

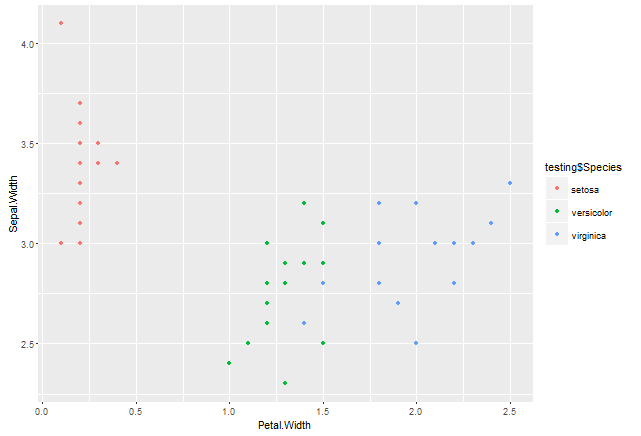
> equalPredictions <- (plda==pnb)

> qplot(Petal.Width, Sepal.Width, colour=equalPredictions, data=testing)



* So let’s try to figure out what happened with this one point that didn’t get classified the same way by the two algos
* Let’s try to plot the points colored by species instead

> qplot(Petal.Width, Sepal.Width, colour=testing$Species, data=testing)



* As we can see in this graph, the point that was classified differently by the two algos is basically on the boundary between versicolor and virginica.

# Week 4

# Regularized Regression

## Basic Idea

1. Fit a regression model
2. Penalize or shrike the large coefficients

## Pros

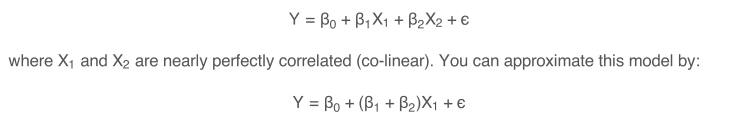
* Can help with the bias/variance tradeoff
  + If certain variables are highly correlated with eachother, we might not wanna include them both in the linear regression model as they will have a very high variance. Leaving them out might slightly bias your model (might lose a bit of prediction capability), but you’ll save a lot on the variance and therefore improve rror
* Can help with model selectio.

## Cons

* May be compuitationally demanding on large data sets
* Doesn’t perform quite as well as random forests or boosting

## Motivating Example

* Suppose we fit a simple regression model as so:



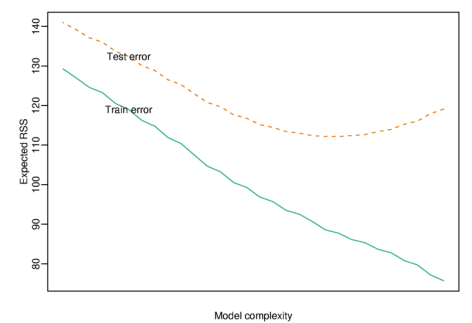
The result is:

* You will get a good estimate of Y
* The estimate of Y will be biased
* We may reduce variance in this estimate

## Prostate Cancer Example

## Subset Selection

* For the training set (in-sample error), the error always goes down as the number of predictors increases.
* However, for the test set, it tends to go down, then plateu, then INCREASe
* This is because we’re overfitting the data in the training set
* The following graph illustrates the phenomenon



## Model Selection Approach: Split Samples

* In general, the best approach might be when you have enough data and computation time to split samples

### Approach

1. Divide data into training/test/validation sets
2. Treat validation as test data, train all competing models on the train data and pick the best one on validation
3. To properly assess performance on new data, apply to test set
4. You may re-split and reperform steps 1-3

### Common Problems

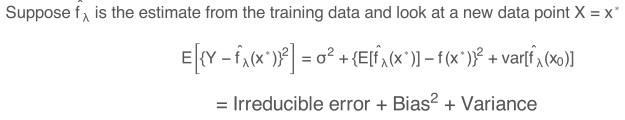
* We have limited amouints of data
* Computational Complexity

## Decomposing Expected Prediction Error



Then





The goal when producing a prediction model is to reduce the circled term

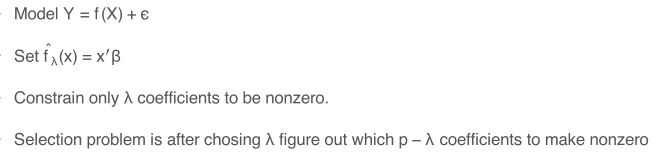
* The irreducible error obviously usually can’t be reduced
* We can though tradeoff bias and variance

## Another Issue for High-Dimensional Data

|  |
| --- |
| > small = prostate[1:5,]  > lm(lpsa ~ ., data=small)  Call:  lm(formula = lpsa ~ ., data = small)  Coefficients:  (Intercept) lcavol lweight age lbph svi lcp  9.60615 0.13901 -0.79142 0.09516 NA NA NA  gleason pgg45 trainTRUE  -2.08710 NA NA |
|  |
| |  | | --- | | > | |

* We take a small subset of only 5 obserfcations from the prostate data
* However, there are more than 5 predictor variables
* Some predictor variables get estiamtes, but some will be NA
* R won’t be able to estimate spome of them because you have more preidcotrs than samples
* This is due to having a design matrix that cannot be inverted

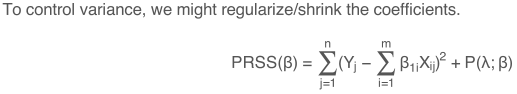
## Hard Thresholding



The question is , after we pick lambda, suppose let’s say that there are only 3 nonzero coefficients, then we have to try all possible combinations of 3 coefficicients that are not 0, and then fit the best model.

## Regularization for Regression

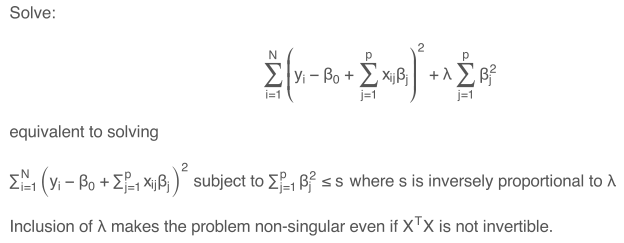
* If our coeffcicients aren’t constrained, they can explode!
* Hence they are succeptible to very high variance



Where PRSS is a penalized form of the sum of squares. Things that are commonly looked for are:

* Penalty reduces complexity
* Penalty reduces variance
* Penalty respects structure of the problem

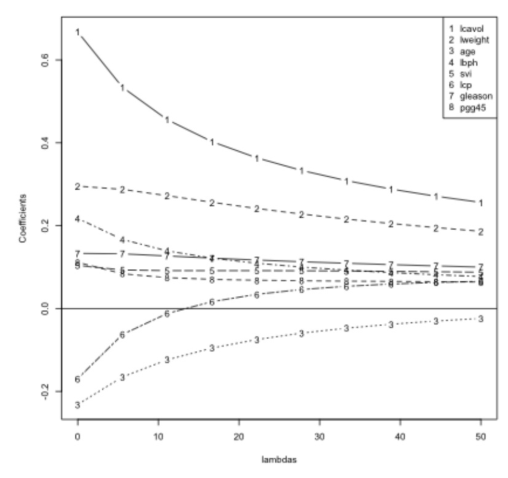
## Ridge Regression



Notice the circled term. This term implies that if our coefficicents are very large, the entire quantity gets large. This will keep the coeffcients small

The red part implies that ridge regeression will work even for that once scenario where we had more predictors than observations

## Ridge Coefficient Paths



* As lambda increases, that means we penalize the big coefficients more and more
* We start off with the coefficients being equal to a certain set of values when lambda is equal to 0
* As you increase lambda, all of the coefficients get closer to zero
* This is because we’re penalizing the coefficients and making them smaller

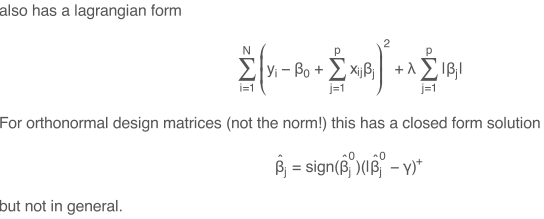
## Tuning Parameter λ

* λ controls the size of the coefficients
* λ controls the amount of regularization
* As λ approaches 0 we obtain the least square solution
* As λ approaches infinite we have our coefficients approaching zero

## Lasso

* A similar approach can be done with a slight change in penalty





* Notice the difference between this and the earlier approach. Here we are taking the absolute value of the coefficients as opposed to the square of the coefficients
* The lasso shrinks some of the coefficients and sets some of them to exactly 0 in the case that beta – gamma < 0
* Some people like this approach because it both shrinks coefficients and performs model selection for you in advance by setting some of them exactly to 0

## Notes

* In Caret, the methods are:
  + ridge
  + lasso
  + relaxo

# Combining Predictors

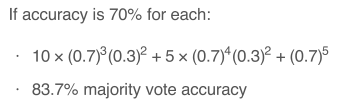
* Sometimes called ensembling methods

## Key Ideas

* You can combine classifiers by averaging/voting
  + You can combine a boosting classifier with a random forest with a linear regresison
* Combining classifiers improves accuracy
* Combining classifiers reduces interpretability
  + Be careful that the gain we get in accuracy is worth the loss in interpretability
* Boosting, bagging, and random forests are variants on this theme
  + However, these are all examples where it’s the same kind of classifier being averaged

## Basic Intuition – Majority Vote

Suppose we have 5 completely independent classifiers



This is saying that the accuracy of majority vote would be the probability of 3 of the classifiers getting it right, plus the probability of 4 of the classifiers getting it right, plus the probability of 5 of the classifiers getting it right.

\*The 10 and 5 in the equation above are results of the binomial coefficient, 5 choose 3 and 5 choose 4 respectively

With 101 independent classifiers

* You get 99.9% majority vote accuracy

## Approaches for Combining Classifiers

1. Bagging boosting, random forests
   1. Usually combine similar classifiers
2. Combining different classifiers
   1. Model stacking
   2. Model ensembling

## Example with Wage Data

> Wage <- subset(Wage, select=-c(logwage)) #Leave out the logwage variable since it is a fantastic predictor

>

> inBuild <- createDataPartition(y=Wage$wage, p=0.7, list=FALSE)

> validation <- Wage[-inBuild,]

> buildData <- Wage[inBuild,]

> inTrain <- createDataPartition(y=buildData$wage, p=0.7, list=FALSE)

> training <- buildData[inTrain,]

> testing <- buildData[-inTrain,]

So we have separated into 3 different datasets

> dim(training)

[1] 1474 11

> dim(testing)

[1] 628 11

> dim(validation)

[1] 898 11

## Build Two Different Models

* In the training set we are going to build 2 different models

> mod1 <- train(wage ~ ., method="glm", data=training)

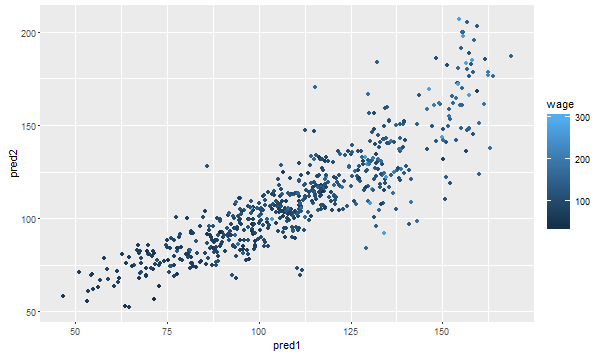
> mod2 <- train(wage ~ ., method="rf", data=training, trControl = trainControl(method="cv"), number=3)

* Now let’s see what these predictions look like

> pred1 <- predict(mod1, testing)

> pred2 <- predict(mod2, testing)

> qplot(pred1, pred2, colour=wage, data=testing)



* Here we can see that thew predicitons are close to eachother, but they don’t exactly agree with eachother
* Also, neither of them perfectly correlates with the wage variable which is the colour of the dots

## Fit a Model That Combines Predictors

### Example Predicting on Test Set

> predDF <- data.frame(pred1, pred2, wage=testing$wage)

> combModFit <- train(wage ~ ., method="gam", data=predDF)

> combPred <- predict(combModFit, data=predDF)

* Now instead of just fitting a model that realtes the covariants to the outcome, we’ve firt 2 separate models, and now we’re fitting a regression model that relating the outcome to the predicitons from those 2 models. Then we can predict from the combined dataset on new samples

### Testing Errors

> sqrt(sum((pred1-testing$wage)^2))

[1] 857.8234

> sqrt(sum((pred2-testing$wage)^2))

[1] 892.7873

> sqrt(sum((combPred-testing$wage)^2))

[1] 848.0532

* \*\*\*Important, we used the test set to blend the two models together, so we fit the model on the test set. Therefore, this is not a good representation of thee out of sample error
* Hence, we need to try this model out on the validation set now.

### Example Predicting on Validation Set

> pred1V <- predict(mod1, validation)

> predVF <- data.frame(pred1=pred1V, pred2 = pred2V)

> combPredV <- predict(combModFit, predVF)

* The covariants being passed to the model are the predicitions from the two different models

### Error on Validation Set

> sqrt(sum((pred1V - validation$wage)^2))

[1] 1048.556

> sqrt(sum((pred2V - validation$wage)^2))

[1] 1085.455

> sqrt(sum((combPredV - validation$wage)^2))

[1] 1042.032

## Final Notes

* Even simple blending like the type we just did in our example can be very useful
* The typical model for binary/multiclass data involves
  + Building an odd number of models
  + Predict the outcome with each model
  + Assigning the ultimate class label based on majority vote

### Scalability Matters!

* Ensembling can lead to computational complexity
* Think back to the netflix example
* There are tradeoffs in scalability vs accuracy thast you have to pay attention ton when making these prediction models

# Forecasting

## Basic Info

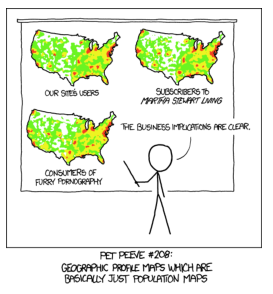
* A very specific kind of prediction problem
* Typically applied to time series data
* The introduced a dependences structure and certain things hthat must be taken into account when performing prediction

## What is Different?

* The data are dependent over time and that alone makes prediction more challenging.
* There are specific pattern types that should be paid attention to
  + Trends - long term increase or decrease
  + Seasonal patterns - pattern related to time of week, month, year
  + Cycles - patterns that rise and fall periodically over a period that’s longer than a year for example
* Subsampling into training and test sets is more complicated. You can’t just take random observations since observations are dependent on time
* Similar Issues arise in spatial data
  + Dependency between nearby observations
  + Location specific effects that have to be modelled when doing prediction
* Typically the goal is to predict one or more observations into the future
* All standard prdicitons can be used, BUT YOU MUST BE CAUTIOUS

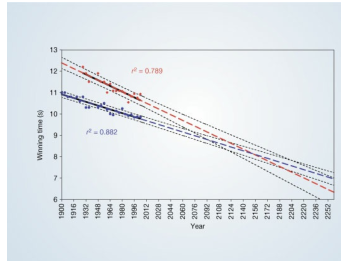
## Beware Spurious Correlations

* Time series can often be correlated for reasons that do not make them good for predicting one from the other
* Can be common in geographical analysis
* Here is an example with a population based heat maps



Notice how they have very similar shapes. This is due to the population density in the U.S.

## Beware Extrapolation

* The following graph says that women will be faster tehn men at springing by the year 2156
* 
* Furthermore, this extrapolation is dangerous because according to these lines, at some point in the future, both men and women will be predicted to run negative times
* You have to always be careful with how far out you extrapolate from your data

## Google Stock Data

> from.dat <- as.Date("01/01/08", format="%m/%d/%y")

> to.dat <- as.Date("12/31/13", format="%m/%d/%y")

> getSymbols("GOOG", src="google", from=from.dat, to=to.dat)

[1] "GOOG"

> head(GOOG)

GOOG.Open GOOG.High GOOG.Low GOOG.Close GOOG.Volume

2008-01-02 346.09 348.34 338.53 342.25 NA

2008-01-03 342.29 343.08 337.92 342.32 NA

2008-01-04 339.51 340.14 327.17 328.17 NA

2008-01-07 326.64 330.81 318.36 324.30 NA

2008-01-08 326.17 329.65 315.18 315.52 NA

2008-01-09 314.70 326.34 310.94 326.27 NA

### Summarized Monthly and Store as Time Series

> tempGOOG <- GOOG[, -5] #Simply removing the volume column since it contains only NA values

> head(tempGOOG)

GOOG.Open GOOG.High GOOG.Low GOOG.Close

2008-01-02 346.09 348.34 338.53 342.25

2008-01-03 342.29 343.08 337.92 342.32

2008-01-04 339.51 340.14 327.17 328.17

2008-01-07 326.64 330.81 318.36 324.30

2008-01-08 326.17 329.65 315.18 315.52

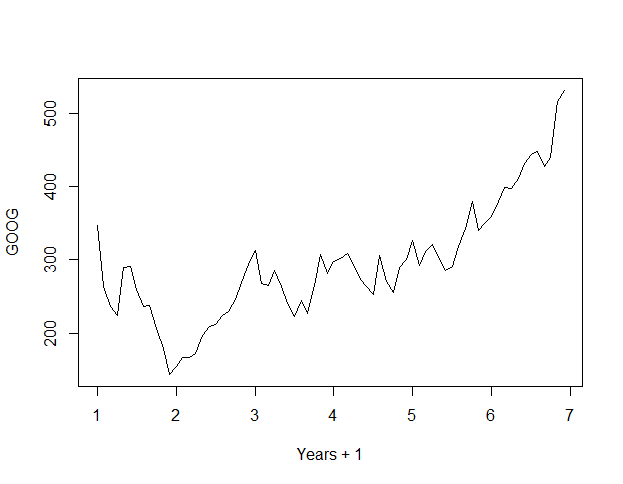
2008-01-09 314.70 326.34 310.94 326.27

> mGoog <- to.monthly(tempGOOG)

> googOpen <- Op(mGoog)

> ts1 <- ts(googOpen, frequency=12)

> plot(ts1, xlab="Years + 1", ylab="GOOG")



* The to.monthly() function converts our data to a monthly time series
  + Let’s take a quick look at what mGoog looks like

tempGOOG.Open tempGOOG.High tempGOOG.Low tempGOOG.Close

Jan 2008 346.09 348.34 259.24 281.87

Feb 2008 264.07 270.25 223.20 235.35

Mar 2008 235.52 236.12 205.85 220.01

Apr 2008 223.65 292.14 220.28 286.86

May 2008 288.87 300.92 268.64 292.61

Jun 2008 290.96 293.73 257.29 262.95

* + AS we can see, this is basically like doing a group by in SQL. For example, for Jan 2008, it took the first day in January for Open, the max of the high price for High, the min of tempGOOG.Low for Low, and the last day in January as Close
  + This is confirmed by the following table

> janGoog <- GOOG[1:21,]

> janGoog

GOOG.Open GOOG.High GOOG.Low GOOG.Close GOOG.Volume

2008-01-02 346.09 348.34 338.53 342.25 NA

2008-01-03 342.29 343.08 337.92 342.32 NA

2008-01-04 339.51 340.14 327.17 328.17 NA

2008-01-07 326.64 330.81 318.36 324.30 NA

2008-01-08 326.17 329.65 315.18 315.52 NA

2008-01-09 314.70 326.34 310.94 326.27 NA

2008-01-10 322.18 328.27 319.73 323.04 NA

2008-01-11 321.03 324.41 314.74 318.81 NA

2008-01-14 325.24 328.37 322.30 326.58 NA

2008-01-15 322.78 324.20 317.37 318.51 NA

2008-01-16 314.17 319.68 300.66 307.67 NA

2008-01-17 310.07 312.56 298.71 300.09 NA

2008-01-18 303.88 304.69 298.93 299.82 NA

2008-01-22 280.73 298.45 280.32 291.88 NA

2008-01-23 280.07 283.72 259.24 274.04 NA

2008-01-24 279.12 289.56 276.79 286.96 NA

2008-01-25 295.61 297.20 282.81 282.92 NA

2008-01-28 285.20 285.83 274.03 277.71 NA

2008-01-29 279.95 280.38 270.06 274.98 NA

2008-01-30 274.32 279.93 271.48 273.86 NA

2008-01-31 269.24 286.21 266.88 281.87 NA

* Then we just take the opening price into googOpen
* Then we create a time series object that looks like so

> ts1

Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov

1 346.09 264.07 235.52 223.65 288.87 290.96 259.53 236.02 238.15 205.86 178.61

2 154.15 166.98 166.50 171.72 197.32 209.16 211.89 224.15 229.61 246.25 268.27

3 313.16 267.03 264.34 285.16 262.99 239.97 222.42 244.25 227.26 264.74 307.56

4 297.94 301.94 308.58 294.09 272.58 263.76 253.12 305.30 270.10 254.67 289.76

5 326.14 292.18 310.82 320.06 301.59 285.61 290.62 318.33 341.93 379.15 339.41

6 359.35 378.70 398.50 397.11 411.22 431.34 442.78 447.05 426.75 439.68 515.38

Dec

1 143.20

2 293.77

3 281.22

4 299.70

5 350.77

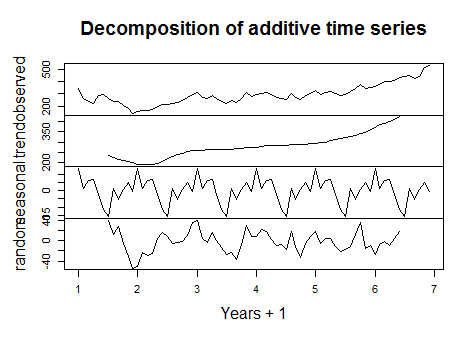
6 531.22

* What results from plotting ts1 is a graph of the monthly opening prices of google’ stock for 7 years

## Time Series Decomposition

* **Trend:** Consistently increasing pattern over time
* **Seasonal:** When there is a pattern over a fixed period of time that recurs
* **Cyclic:** When data rises and falls over non fixed periods

> plot(decompose(ts1), xlab="Years + 1")



* WE can see that there is an upward trend based on the 2nd graph from the top
* There also appears to be a seasonal pattern based on the 3rd graph from the top
* Additionally, there is a random cyclical pattern in the dataset based on the graph at the bottom

## Training and Test Sets

* We have to build training and test sts having consecutive time points

> ts1Train <- window(ts1, start=1, end=5)

> ts1Test <- window(ts1, start=5, end=(7-0.01))

> ts1Train

Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

1 346.09 264.07 235.52 223.65 288.87 290.96 259.53 236.02 238.15 205.86 178.61 143.20

2 154.15 166.98 166.50 171.72 197.32 209.16 211.89 224.15 229.61 246.25 268.27 293.77

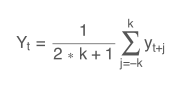
3 313.16 267.03 264.34 285.16 262.99 239.97 222.42 244.25 227.26 264.74 307.56 281.22

4 297.94 301.94 308.58 294.09 272.58 263.76 253.12 305.30 270.10 254.67 289.76 299.70

5 326.14

* So notice how our training set window has consecutive time points, and how our test set window has consecutive time points immediately after the training set

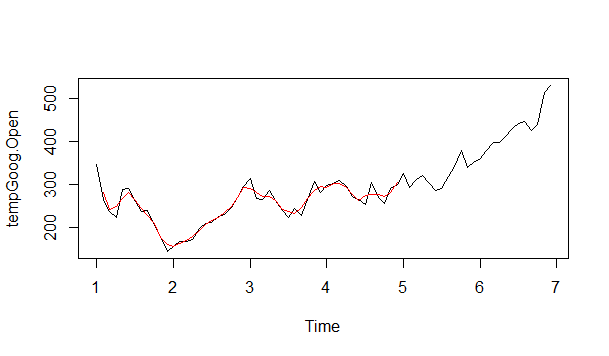
## Simple Moving Average



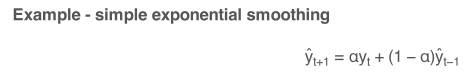
* Averages all of the values for a particulartime point

> plot(ts1)

> lines(ma(ts1Train,order=3), col="red")



## Exponential Smoothing





* Basically we weight nearby time points more heavily than time points that are further away
* You can fit a model where you have a choice for the different types of trends that you might want to fit
* When you forecast, you can get prediction bounds for the possible values that you could get from that prediction

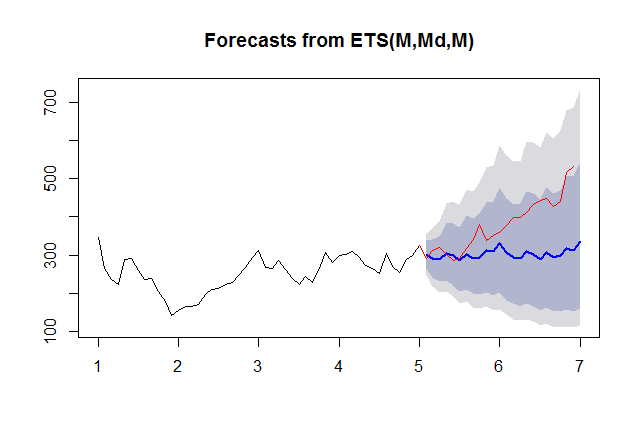
### Example

> ets1 <- ets(ts1Train, model="MMM") #This actually fits a model

> fcast <- forecast(ets1)

> plot(fcast)

> lines(ts1Test, col="red")



* The shaded area is the prediction bounds of the possible values that we could get from our prediction
* So the blue line is the forecast created by the model and the

## Getting the Accuracy

> accuracy(fcast, ts1Test)

ME RMSE MAE MPE MAPE MASE ACF1

Training set -0.4052602 24.07031 19.40497 -0.69944 7.867479 0.3685844 0.09043751

Test set 77.2247187 101.55663 79.50341 18.01131 18.796156 1.5101138 0.77662386

Theil's U

Training set NA

Test set 3.704502

## Final Notes

* What we’ve covered in this lecture is a tiny fraction of what is an entire field
* Cautions
  + Be wary of spurious correlations
  + Be careful how far into the future you predict
  + Be ware of dependencies over time
* quantmod and quandl are very useful for finance related problems

# Unsupervised Prediction

* In all other lectures in this course, we always knew what the labels were. I.e. we were trying to predict an outcome when we know what it is supposed to be

## Key Ideas

* Sometimes it is the case that we don’t know the labels for prediction, so you have to discover those labels in advance
* To build a predictor
  + Create clusters from the data you have observed
    - First of all creating the clusters in not a perfectly noiseless process
  + Add names to those clusters
    - Interpreting the clusters well in an incredibly challenging problem
  + Build predictor for clusters
    - Basically using the algos that we havwe learned so far in this course on the clusters
* In a new data set
  + Predict clusters
* This adds several layers of difficulty to the prediction problem

## Iris Example Ignoring Special Labels

> data("iris")

> inTrain <- createDataPartition(y=iris$Species, p=0.7, list=FALSE)

> training <- iris[inTrain,]

> testing <- iris[-inTrain,]

> dim(training)

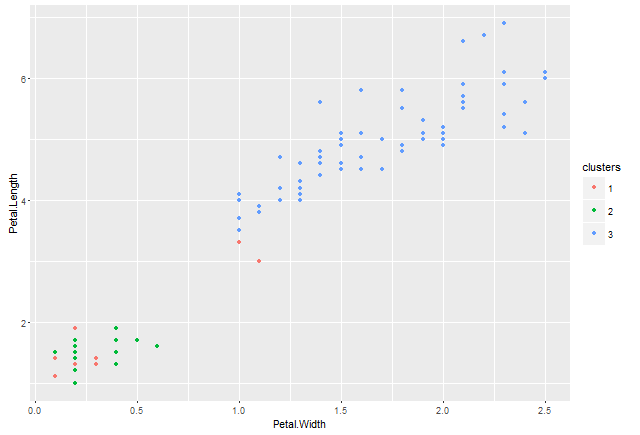
[1] 105 5

> dim(testing)

[1] 45 5

### Cluster with K-Means

* The basic idea here is to create 3 different clusters while ignoring the species information from the iris data frame



* For some reason, the clustering didn’t work very well this time since there really should be 3 distinct groups but cluster 3 in this case just takes up way too much space.